

C# با CLR

سوم
ویرایش

جفری ریچر



جلد اول

مترجم
محسن افشین

صلى الله عليه وسلم

C# با CLR

ویرایش سوم

جفری ریچر

مترجم

محسن افشین

تقدیم بہ روح پدرم

و وجود مادرم

...

فهرست مطالب

پیشگفتار نویسنده	۱
پیشگفتار مترجم	۱
مقدمه	۱
فصل ۱: مدل اجرایی CLR	۱
کامپایل سورس کد به ماژول مدیریت شده	۱
ترکیب ماژول های مدیریت شده به اسمبلی ها	۳
بارگذاری اجرایی زبان مشترک	۴
اجرای کد اسمبلی شما	۷
IL و بازبینی	۱۲
کد ناامن	۱۳
ابزار تولید کد اصلی: Native Code Generator (NGen.exe)	۱۴
کتابخانه کلاس فریمورک	۱۶
سیستم مشترک نوع	۱۷
مشخصات مشترک زبان	۱۹
تقابل با کد مدیریت نشده	۲۳
فصل ۲: ساخت، بسته بندی، نصب و مدیریت برنامه ها و نوع ها	۲۴
اهداف راه اندازی دات نت فریمورک	۲۴
ساخت و تبدیل نوع ها به یک ماژول	۲۵
فایل های جواب	۲۶
نگاهی اجمالی بر متادیتا	۲۷
ترکیب ماژول ها و ساخت یک اسمبلی	۳۳
افزودن اسمبلی ها به یک پروژه در ویژوال استودیو	۳۸
استفاده از Assembly Linker	۳۹
افزودن فایل های منبع به یک اسمبلی	۴۱
اطلاعات منبع نسخه اسمبلی	۴۲
شماره های نسخه (Version Numbers)	۴۵
فرهنگ Culture	۴۶
نصب آسان برنامه (اسمبلی های نصب شده شخصی)	۴۷
کنترل های مدیریتی ساده (تنظیمات)	۴۸
فصل ۳: اسمبلی های اشتراکی و اسمبلی های قوی نام	۵۱

۵۱ دو نوع اسمبلی، دو نوع نصب
۵۲ اختصاص یک نام قوی برای یک اسمبلی
۵۶ کش سراسری اسمبلی The Global Assembly Cache
۵۸ ساخت یک اسمبلی که به یک اسمبلی قوی نام ارجاع می کند
۵۹ اسمبلی های قوی نام ضد نفوذ هستند
۶۰ امضای تاخیری
۶۲ نصب اسمبلی های قوی نام به صورت خصوصی
۶۲ چگونه CLR ارجاع به نوع ها را تحلیل میکند
۶۵ کنترل های مدیریتی پیشرفته (تنظیمات)
۶۷ سیاست های کنترلی سازنده
۷۰ فصل ۴: مبانی نوع
۷۰ همه ی نوع ها از System.Object مشتق می شوند
۷۱ تبدیل میان نوع ها
۷۳ تبدیل با عملگرهای is و as سی شارپ
۷۵ فضاهای نام (Namespace) و اسمبلی ها
۷۸ چگونه چیزها در زمان اجرا به هم ربط پیدا می کنند
۸۸ فصل ۵: نوع های اصلی، ارجاعی و مقداری
۸۸ نوع های اصلی زبان برنامه نویسی
۹۱ عملیات های کنترل شده و کنترل نشده بر روی نوع اصلی
۹۳ نوع های ارجاعی و نوع های مقداری
۹۹ بسته بندی و باز کردن نوع های مقداری
۱۰۹ تغییر فیلدها در یک نوع مقداری بسته بندی شده به کمک رابط ها (و چرا شما نباید این کار را انجام دهید)
۱۱۲ برابری و هویت شی
۱۱۴ کدهای هش شی
۱۱۶ نوع اصلی dynamic
۱۲۱ فصل ۶: مبانی نوع و عضو
۱۲۱ گونه های مختلف اعضای نوع
۱۲۳ پدیداری نوع
۱۲۴ اسمبلی های دوست
۱۲۵ دسترس پذیری عضو
۱۲۶ کلاس های استاتیک
۱۲۷ کلاس های جزئی، ساختارها و رابط ها

۱۲۸ کامپوننت ها، چندریختی و نسخه بندی
۱۳۰ چگونه CLR متدها، ویژگی ها و رویدادهای مجازی را فراخوانی می کند
۱۳۳ استفاده هوشمندانه از پدیداری نوع و دسترس پذیری نوع
۱۳۶ کار با متدهای مجازی هنگام نسخه بندی نوع ها
۱۴۱ فصل ۷: ثابت ها و فیلدها
۱۴۱ ثابت ها
۱۴۲ فیلدها
۱۴۵ فصل ۸: متدها
۱۴۵ سازنده های نمونه و کلاس ها (نوع های ارجاعی)
۱۴۸ سازنده های نمونه و ساختارها (نوع های مقداری)
۱۵۰ سازنده های نوع
۱۵۳ کارایی سازنده ی نوع
۱۵۶ متدهای سرپارگذاری عملگرها
۱۵۷ عملگرها و تقابل زبان برنامه نویسی
۱۵۸ متدهای عملگر تبدیل
۱۶۱ متدهای گسترشی
۱۶۳ قوانین و راهنمایی ها
۱۶۴ گسترش نوع های مختلف با متدهای گسترشی
۱۶۵ خاصیت گسترشی
۱۶۶ متدهای جزئی
۱۶۸ قوانین و راهنمایی ها
۱۶۹ فصل ۹: پارامترها
۱۶۹ پارامترهای نامی و انتخابی
۱۷۰ قوانین و راهنمایی ها
۱۷۱ Optional و DefaultValue صفت های
۱۷۲ متغیرهای محلی با نوع ضمنی
۱۷۳ ارسال پارامتر با ارجاع به یک متد
۱۷۸ ارسال تعداد متغیری از آرگومان ها به یک متد
۱۸۰ راهنمایی های مربوط به پارامتر و نوع برگشتی
۱۸۲ ثابت بودن
۱۸۳ فصل ۱۰: ویژگی ها
۱۸۳ ویژگی های بدون پارامتر

۱۸۶	ویژگی هایی که به صورت خودکار پیاده سازی می شوند
۱۸۷	تعریف هوشمندانه ویژگی ها
۱۸۹	مقداردهی کننده های شی و مجموعه
۱۹۱	نوع های ناشناس
۱۹۴	نوع System.Tuple
۱۹۶	ویژگی های پارامتردار
۲۰۰	کارایی در فرخوانی متدهای دستیابی ویژگی
۲۰۰	دسترس پذیری متدهای دستیابی ویژگی
۲۰۱	متدهای جنریک دستیابی ویژگی
۲۰۲	فصل ۱۱: رویدادها
۲۰۳	طراحی یک نوع که یک رویداد را ارائه می کند
۲۰۳	قدم اول: تعریف نوعی که اطلاعات اضافی که می بایست به دریافت کننده ی خبر رویداد، ارسال شود را نگهداری می کند
۲۰۴	قدم دوم: تعریف عضو رویداد
۲۰۵	قدم سوم: تعریف یک متد که مسئول فعال کردن رویداد جهت اطلاع به اشیاء ثبت شده درباره ی رخداد رویداد است
۲۰۸	قدم چهارم: تعریف یک متد که ورودی را به رویداد مطلوب ترجمه میکند
۲۰۹	کامپایلر چگونه یک رویداد را پیاده سازی می کند
۲۱۱	طراحی یک نوع که از یک رویداد استفاده می کند
۲۱۲	پیاده سازی صریح یک رویداد
۲۱۶	فصل ۱۲: جنریک ها
۲۲۰	جنریک ها در کتابخانه کلاس فریمورک
۲۲۱	کتابخانه ی Power Collections از Wintellect
۲۲۱	زیر ساختار جنریک ها
۲۲۲	نوع های باز و بسته
۲۲۴	نوع های جنریک و وراثت
۲۲۵	هویت یک نوع جنریک
۲۲۶	انفجار کد
۲۲۷	رابط های جنریک
۲۲۸	نماینده های جنریک
۲۲۸	آرگومان های نوع جنریک Contravariant و Covariant رابط ها و نماینده ها
۲۳۰	متدهای جنریک
۲۳۱	متدهای جنریک و استثنا نوع
۲۳۲	جنریک ها و دیگر اعضا

۲۳۲	قابلیت بازبینی و محدودیت ها
۲۳۴	محدودیت های اصلی
۲۳۵	محدودیت های ثانویه
۲۳۷	محدودیت های سازنده
۲۳۷	دیگر مسائل قابلیت بازبینی
۲۴۰	فصل ۱۳: رابط ها
۲۴۰	وراثت کلاس و رابط
۲۴۰	تعریف یک رابط
۲۴۲	به ارث بردن یک رابط
۲۴۴	اطلاعات بیشتر درباره فراخوانی متدهای رابط
۲۴۵	پیاده سازی صریح و ضمنی متد رابط (در پشت صحنه چه رخ می دهد)
۲۴۶	رابط های جنریک
۲۴۸	جنریک ها و محدودیت های رابط
۲۴۹	پیاده سازی چندین رابط که نام و امضای متد یکسانی دارند
۲۵۰	بهبود امنیت نوع در زمان کامپایل در پیاده سازی های صریح متد رابط
۲۵۲	هنگام پیاده سازی های صریح متد رابط مراقب باشید
۲۵۴	طراحی: کلاس پایه یا رابط ؟
۲۵۶	فصل ۱۴: کاراکترها، رشته ها و کار با متن
۲۵۶	کاراکترها
۲۵۸	نوع System.String
۲۵۸	ساختن رشته ها
۲۶۰	رشته ها تغییر ناپذیرند
۲۶۱	مقایسه رشته ها
۲۶۶	وارد کردن رشته (String Interning)
۲۶۸	ادغام رشته ها
۲۶۹	بررسی کاراکترهای یک رشته و عناصر متنی
۲۷۱	دیگر عملیات های رشته
۲۷۱	ساخت یک رشته به صورت کارا
۲۷۲	ساخت یک شی StringBuilder
۲۷۲	اعضای StringBuilder
۲۷۴	بدست آوردن نمایش رشته ای از یک شی ToString
۲۷۵	فرمت های خاص و فرهنگ ها

۲۷۸	فرمت کردن چندین شی به یک رشته
۲۷۹	فراهم کردن فرمت کننده ی سفارشی خودتان
۲۸۱	تجزیه یک رشته برای بدست آوردن یک شی : Parse
۲۸۳	Encoding : تبدیل بین کاراکترها و بایت ها
۲۸۸	اینکدینگ و دیکدینگ استریم های کاراکترها و بایت ها
۲۸۸	اینکدینگ و دیکدینگ رشته مبنای ۶۴
۲۸۹	رشته های امن
۲۹۲	فصل ۱۵: نوع های شمارشی و پرچم های بیتی
۲۹۲	نوع های شمارشی
۲۹۷	پرچم های بیتی
۳۰۰	افزودن متد به نوع های شمارشی
۳۰۲	فصل ۱۶: آرایه ها
۳۰۴	مقداردهی اولیه عناصر آرایه
۳۰۵	تبدیل آرایه ها
۳۰۷	تمام آرایه ها به صورت ضمنی از System.Array مشتق شده اند
۳۰۸	تمام آرایه ها به صورت ضمنی IEnumerable, ICollection, و IList را پیاده سازی می کنند
۳۰۹	ارسال و برگرداندن آرایه ها
۳۰۹	ساخت آرایه هایی با حد پایین غیر صفر
۳۱۰	کارایی دسترسی به آرایه
۳۱۵	دسترسی ناامن به آرایه ها و آرایه های با اندازه ثابت
۳۱۷	فصل ۱۷: نماینده ها
۳۱۷	نگاه ابتدایی به نماینده ها
۳۱۹	استفاده از نماینده ها برای کالک کردن (Call Back) متدهای استاتیک
۳۲۰	استفاده از نماینده ها برای کالک کردن (Call back) متدهای نمونه
۳۲۱	روشن کردن موضوع نماینده ها
۳۲۵	استفاده از نماینده ها برای فراخوانی چند متد (زنجیربندی)
۳۲۹	پشتیبانی سی شارپ برای زنجیرهای نماینده
۳۲۹	کنترل بیشتر برای فراخوانی زنجیر نماینده
۳۳۱	قبلا به اندازه کافی نماینده ها را معرفی کرده ایم (نماینده های جنریک)
۳۳۲	شکر نحوی سیشارپ برای نماینده ها
۳۳۳	میانبر نحوی شماره ۱: نیاز به ساخت یک شی نماینده نیست
۳۳۳	میانبر نحوی شماره ۲: نیاز به تعریف یک متد کالک نیست

۳۳۶ میانبر نحوی شماره ۳: نیاز نیست متغیرهای محلی در یک کلاس را جهت ارسال به یک متد کالیک، به صورت دستی بیوشانید
۳۳۹ نماینده ها و رفلکشن
۳۴۳ فصل ۱۸: صفت های سفارشی
۳۴۳ استفاده از صفت های سفارشی
۳۴۶ تعریف کلاس صفت خودتان
۳۴۹ سازنده صفت و نوع های داده ای فیلد/ویژگی
۳۵۰ شناسایی استفاده از یک صفت سفارشی
۳۵۴ بررسی تطابق دو نمونه صفت در مقابل هم
۳۵۶ شناسایی استفاده از یک صفت سفارشی بدون ساخت اشیاء مشتق شده از Attribute
۳۶۰ کلاس های صفت شرطی
۳۶۱ فصل ۱۹: نوع های مقداری تهی پذیر
۳۶۲ پشتیبانی سیشارپ برای نوع های مقداری تهی پذیر
۳۶۵ عملگر ترکیب گر تهی سی شارپ
۳۶۶ CLR پشتیبانی ویژه برای نوع های مقداری تهی پذیر دارد
۳۶۶ بسته بندی نوع های مقداری تهی پذیر
۳۶۶ بازکردن نوع های مقداری تهی پذیر
۳۶۷ فراخوانی GetType از طریق یک نوع مقداری تهی پذیر
۳۶۷ فراخوانی متدهای رابط از طریق یک نوع مقداری تهی پذیر
۳۶۸ فصل ۲۰: اکسپشن ها و مدیریت وضعیت
۳۶۸ تعریف "اکسپشن"
۳۶۹ مکانیک مدیریت اکسپشن
۳۷۰ بلوک try
۳۷۰ بلوک catch
۳۷۲ بلوک finally
۳۷۵ کلاس System.Exception
۳۷۸ کلاس های اکسپشن تعریف شده در FCL
۳۸۰ تولید یک اکسپشن
۳۸۱ تعریف کلاس اکسپشن خودتان
۳۸۳ معامله ی قابلیت اطمینان در برابر بهره وری
۳۹۰ راهنمایی ها و بهترین تجربه ها
۳۹۱ از بلوک های finally آزادانه استفاده کنید

۳۹۲	هرچیزی را نگیرید
۳۹۳	احیای آرام از یک اکسپشن
۳۹۴	برگشت از یک عملیات نیمه کامل وقتی یک اکسپشن غیرقابل احیا رخ می دهد - نگهداری وضعیت
۳۹۴	مخفی سازی جزئیات پیاده سازی برای حفظ یک "قرارداد"
۳۹۷	اکسپشن های مدیریت نشده
۴۰۱	خطایابی اکسپشن ها
۴۰۳	ملاحظات عملکردی مدیریت اکسپشن
۴۰۵	نواحی اجرایی محدود شده (CERs) Constrained Execution Regions
۴۰۸	قراردادهای کد

پیشگفتار نویسنده

در آغاز وقتی جف از من خواست که برای این کتاب پیشگفتار بنویسم خیلی خوشحال شدم؛ فکر می‌کنم او برای من خیلی احترام قائل است. من در لیست نویسندگان پیشگفتار در مکان چهارم جای داشتم. واضح است که قضیه برای هیچ یک از دیگر نویسندگان (بیل گیتس، استیو بالمر، کارترین زتا و دیگران) مثل من نبود. حداقل او برای من یک شام خرید.

اما هیچ‌کس به آن میزان که من می‌توانم بگویم، قادر به حرف زدن پیرامون این کتاب نیست. منظورم این است که کاترین می‌توانست یک توصیف خوب ارائه کند اما من همه چیز راجع به فلکشن و اکسپشن‌ها و آپدیت‌های زبان سی‌شارپ را می‌دانم چرا که سال‌ها و سال‌ها پیرامون آن بحث کرده‌ام. این بحث استاندارد بر سر میز شام در خانه‌ی ماست. دیگران پیرامون آب و هوا صحبت می‌کنند اما ما درباره‌ی دانت. حتی آیدان، پسر شش ساله ما، درباره‌ی کتاب جف می‌پرسد. اغلب درباره‌ی اینکه چه زمانی کتاب به پایان می‌رسد تا بتوانند حسابی بازی کنند. گرنت (دو ساله) هنوز صحبت نمی‌کند، اما اولین کلمه‌ای که او بر زبان خواهد آورد احتمالاً "سریالی" خواهد بود.

در واقع اگر می‌خواهید بدانید چگونه همه‌ی این‌ها آغاز شد، داستان به ۱۰ سال پیش باز می‌گردد. تقریباً ۱۰ سال پیش، جف به یک جلسه‌ی سری و مهم در مایکروسافت رفت.

آن‌ها تعدادی افراد خبره (واقعا، شما چگونه این مقام را گرفتید؟ باور کنید که این به خاطر مدرک دانشگاهی جف نبود) را جمع کردند و COM جدید را بیرون دادند. همان روزها بود که جف گفت COM مرده است. تا چند روز اطراف خیابان ۴۲ مایکروسافت پرسه می‌زد؛ به این امید که چیز بیشتری درباره‌ی دانت یاد بگیرد. این کار تمام نشد و این کتاب چیزی است که او می‌خواهد برای این کار نشان دهد.

برای سال‌ها جف به من درباره‌ی تردینگ می‌گفت. او واقعا این موضوع را دوست دارد. یکبار در نیواورلان، در پیاده‌روی ۲ ساعته او گفت که مطالب کافی برای یک کتاب تردینگ را دارد؛ هنر تردینگ، اینکه تا چه حد تردینگ در ویندوز به اشتباه درک می‌شود قلبش را شکست. (تمام تردهای آن بیرون). آن‌ها کجا می‌روند؟ وقتی کسی برای آن‌ها برنامه‌ای ندارد، آن‌ها کجا می‌روند؟ این‌ها سوالات دنیا از جف است؛ معانی عمیق‌تر در زندگی. سرانجام او در این کتاب آن‌ها را نوشت.

همه اینجاست. باور کنید اگر می‌خواهید پیرامون تردینگ بدانید، هیچ‌کس به اندازه‌ی جف با آن کار و به آن فکر نکرده است. و تمام ساعات هدر رفته‌ی زندگی او که غیر قابل برگشت است، اینجا در این کتاب جمع شده است. لطفاً آن را بخوانید. سپس به او ایمیل بزنید که چقدر این اطلاعات زندگی شما را تغییر داد. در غیر این صورت او ادبی تراژیک دیگری است که زندگی‌اش بدون تکامل و بی‌معنی پایان یافته است. او آنقدر نوشابه رژیمی خواهد نوشید تا بمیرد.

این ویرایش کتاب، فصل جدیدی پیرامون سریالی‌کننده‌های زمان اجرا هم دارد. این کتاب صبحانه‌ی جدیدی برای کودکان نیست. وقتی فکر کردم دیدم که بحث بیشتر کامپیوتری است و نه چیزی که در لیست خرید روزانه جای گیرد، پس آن را بیرون انداختم. من نمی‌دانم که کتاب چه می‌گوید، اما آن اینجاست تا شما آن را (همراه با یک لیوان شیر) بخوانید.

امید من این است اکنون که بحث درباره‌ی جمع‌آوری زباله در تئوری به پایان رسیده است، می‌تواند زباله‌های خودمان را جمع کند و بیرون بگذارد. جدا، چقدر این کار سخت است؟

مردم، این شاهکار جفری ریچر است. کتاب دیگری در کار نخواهد بود. گرچه پس از پایان هر کتاب، این جمله را تکرار می‌کنیم، اما این بار واقعی است. از میان ۱۳ کتاب، این بهترین آنهاست. و آخرین آن‌ها. سریعا آن را تهیه کنید که تنها تعداد اندکی موجود است. برگردیم به زندگی واقعی خودمان. بحث کنیم پیرامون مسائل مهمی همچون اینکه امروز بچه‌ها چه چیزی را شکستند و نوبت کیست که کهنه‌ی بچه را عوض کند.

کریستین تریس (همسر جف)

۲۴ نوامبر ۲۰۰۹



یک صبحانه‌ی معمولی در خانواده ریچر

پیشگفتار مترجم

سیزدهمین روز از بهار سال ۸۹ بود که تصمیم گرفتم این کتاب را ترجمه کنم. هدفی محکم و استوار داشتم و چون از یک ماه پیش کتاب را مطالعه کرده بودم مطمئن بودم که می توانم با وقت گذاشتن برای ترجمه کتاب، اثر ماندگار جفری را برای هموطنانم به ارمغان آورم. اگر چه این نخستین ترجمه بلند من بود اما با بهره گیری از تجربیات گذشته و عزم راسخ، ترجمه این گنج تقریباً ۱۰۰۰ صفحه ای را در ۳۱ تیر ۸۹ به پایان رساندم. ۷۷ روز ترجمه مستمر و تمام وقت به هزاران برگ دستنویس انجامید. کار تایپ و ویرایش را بلافاصله آغاز نمودم و بیشتر کتاب آماده چاپ شد اما به دلایلی تا این زمان به تاخیر افتاد. به دلیل عدم پیروی از سبک ترجمه های موجود در بازار این اثر به مذاق ناشران خوش نیامد در نتیجه به جای چاپ کاغذی به چاپ دیجیتال روی آوردم تا سریعتر، آسانتر و بدون هرگونه سانسوری این گنجینه را به دستان شما برسانم.

امیدوارم آن که باید، بپسندد. و شما هم خطاهای موجود در ترجمه را به بزرگی اندیشه خود نه به خردی نگاه من بر من بخشیده و مرا از وجود خطاهای احتمالی در اثر مطلع نمایید. همه سعی خودم را کردم تا ضمن انتقال مفاهیم مدنظر نویسنده، شما را از نوشته های اصلی نویسنده محروم نکنم.

این اثر با کسب اجازه از نویسنده کتاب ترجمه و عرضه می شود، امیدوارم پل خوبی بین جف و شما بوده باشم.

محسن افشین

۱ شهریور ۱۳۹۰

۲۳ رمضان

مقدمه

اکتبر ۱۹۹۹ بود که افرادی در مایکروسافت، داتنت فریمورک، CLR (اجزایی زبان مشترک) و زبان برنامه‌نویسی سی‌شارپ را به من نشان دادند. وقتی همه-ی آن‌ها را دیدم، تحت تاثیر قرار گرفتم و دانستم روش برنامه‌نویسی برای من به کلی تغییر می‌کند. از من خواستند برای تیم مشاوره انجام دهم و من قبول کردم. در ابتدا تصور کردم داتنت یک لایه‌ی انتزاعی بر روی Win32 API و COM است. هر چه بیشتر زمان گذاشتم، دیدم که بسی وسیع تر از اینست. به طریقی، آن، سیستم عامل خودش است. مدیریت حافظه، امنیت سیستم، بارگذاری کننده‌ی فایل، مکانیزم مدیریت خطا، ایزوله کردن محیط برنامه (AppDomain)، مدل تردینگ، همه و همه را خود انجام می‌دهد. این کتاب همه‌ی این موارد را توضیح می‌دهد تا بتوانید به نحو بهینه نرم افزارها و کامپوننت‌هایی برای این پلتفرم طراحی کنید.

من بخش مهمی از زندگی خود را بر روی تردینگ، اجرای همروند، موازی سازی، همزمان سازی و ... گذرانده ام. امروزه، با گسترش کامپیوترهای چند هسته‌ای این مباحث بسیار مهم شده اند. چند سال پیش تصمیم گرفتم کتابی پیرامون تردینگ بنویسم. اما هر چیزی به چیز دیگری منجر می‌شد و من هرگز این کتاب را ننوشتم. وقتی زمان تجدیدنظر کتاب فرا رسید، تصمیم گرفتم تمام اطلاعات تردینگ را در آن جای دهم. این کتاب، CLR، داتنت فریمورک و زبان سی‌شارپ را پوشش داده و کتاب تردینگ من نیز در آن تعبیه شده است (بخش پنجم "تردینگ").

اکتبر ۲۰۰۹ است که این مطالب را می‌نویسم، درست در دهمین سالی که با داتنت و سی‌شارپ کار می‌کنم. در طی ۱۰ سال، من همه نوع برنامه ساخته ام و به عنوان یک مشاور، فراوان در خود داتنت همکاری کرده ام. به عنوان شریک در شرکت خودم (Wintellect (<http://Wintellect.com>) با مشتریان فراوانی کار کرده ام و به آن‌ها در طراحی، خطایابی و حل مشکلاتشان در داتنت کمک کرده ام. تمام این تجربه‌ها به من در یادگیری مباحثی که دیگران در داتنت با آن مشکل دارند، کمک کرده است. من سعی کرده‌ام تا دانش تمام این تجربه‌ها را در مباحث کتاب پوشش دهم.

این کتاب مناسب چه افرادی است:

هدف این کتاب بیان چگونگی طراحی برنامه‌ها و کلاس‌ها با قابلیت استفاده مجدد برای داتنت است. به خصوص، قصد دارم توضیح دهم CLR چگونه کار می‌کند و چه امکاناتی را فراهم می‌آورد. همچنین پیرامون بخش‌های مختلف کتابخانه کلاس‌های فریمورک (FCL) بحث خواهم کرد.

هیچ کتابی نمی‌تواند FCL را کاملاً توضیح دهد چرا که شامل هزاران نوع می‌باشد و این تعداد در حال افزایش است. در نتیجه بر نوع‌های اصلی که هر برنامه‌نویس بدان‌ها نیاز دارد، تکیه می‌کنم. و این کتاب مخصوص Windows Forms، Windows Presentation Foundation (WPF)، Silverlight، XML Web Services، Web Forms و غیره نیست. پس تکنولوژی‌های معرفی شده در کتاب برای تمام این نوع برنامه‌ها کاربرد دارد.

کتاب با ویژوال استودیو ۲۰۱۰ و داتنت نسخه ۴.۰ و نسخه‌ی ۴.۰ از زبان برنامه‌نویسی سی‌شارپ پیش می‌رود. به این دلیل که مایکروسافت همواره سعی در سازگاری نسخه‌های جدید با قدیم دارد، بسیاری از مباحث کتاب درباره‌ی نسخه‌های قبلی نیز صادق است. تمامی نمونه کدها در زبان سی‌شارپ برای بیان رفتار امکانات مختلف نوشته شده است. اما از آنجا که CLR توسط زبان‌های مختلف قابل استفاده است، مطالب کتاب حتی برای برنامه‌نویس غیر سی‌شارپ نیز مفید است.

نکته

شما می‌توانید کدهای این کتاب را از سایت Wintellect به آدرس <http://Wintellect.com> دریافت کنید. در بخش‌هایی از کتاب کلاس‌هایی از Power Threading Library خودم را توضیح می‌دهم که این کتابخانه نیز از سایت فوق به صورت رایگان قابل دریافت است.

امروزه میکروسافت نسخه‌های مختلفی از CLR ارائه می‌دهد. نسخه‌ی دسکتاپ/سروری وجود دارد که بر روی ویندوز x86 ۳۲ بیتی و همچنین نسخه‌ی ۶۴ بیتی x64 و IA64 ویندوز عمل می‌کند. نسخه‌ی Silverlight موجود است که از روی نسخه‌ی دسکتاپ/سروری تهیه شده است. پس تمام مطالب کتاب درباره‌ی ساخت برنامه‌های Silverlight نیز کار می‌کند فقط با کمی تفاوت در چگونگی بارگذاری اسمبلی‌ها. همچنین نسخه‌ی سبکی از دات‌نت فریمورک به نام .NET Compact Framework وجود دارد که برای گوشی‌های Windows Mobile و Windows CE قابل استفاده است. اغلب مطالب برای .NET Compact Framework نیز قابل استفاده است، اما این پلتفرم هدف اصلی کتاب نیست.

در ۳۱ دسامبر ۲۰۰۱، ECMA International (<http://www.ECMA-International.org>) زبان سی‌شارپ، بخش‌هایی از CLR و بخش‌هایی از FCL را به عنوان استاندارد ثبت کرد. مستندات این استاندارد اجازه داد دیگر شرکت‌ها نسخه‌هایی منطبق بر ECMA از این تکنولوژی‌ها را برای پردازنده و سیستم عامل‌های دیگر طراحی کنند. در واقع Novell، پیاده‌سازی منبع باز از Silverlight را با نام این تکنولوژی‌ها را برای پردازنده و سیستم عامل‌های دیگر طراحی کردند. در واقع Moonlight (<http://www.mono-project.com/Moonlight>) ارائه کرد که مخصوص لینوکس و دیگر سیستم عامل‌های مبتنی بر UNIX/X11 می‌باشد. Moonlight منطبق بر مشخصات ECMA است. بخش‌های زیادی از کتاب پیرامون این استانداردهاست. پس این کتاب برای کار با کتابخانه‌هایی که با استاندارد ECMA تطابق دارند نیز مفید است.

نکته

ویراستاران و شخص خود من تلاش فراوانی کردیم تا دقیق‌ترین، بروزترین و عمیق‌ترین کتاب را با نثر روان و درک آسان و خالی از اشتباه فراهم کنیم. حتی با این تیم بسیار عالی، باز هم اشتباه ممکن است. اگر شما اشتباهی در کتاب یافتید و یا فیدبکی سازنده داشتید، بسیار از شما تشکر می‌کنم اگر با ایمیل من (JefferyR@Wintellect.com) تماس بگیرید.

تقدیم به

کریستین : کلمات قاصر از بیان این است که من چه حسی از زندگی مشترک با وی دارم. من خانواده ام و تمام سرگذشتان را گرامی می‌دارم. من هر روز با عشق تو سرشار می‌شوم.

آیدان (۶ ساله) و گرت (۲ ساله) : هر دوی شما به من الهام می‌دهید و به من یاد داده اید که بازی کنم و شاد باشم. تماشای رشد کردن و بزرگ شدن شما بسیار برایم لذت بخش است. من خوش شانسم که در زندگی شما سهیم هستم. بیش از آنکه تصور کنید شما را دوست داشته و برایتان احترام قائل هستم.

تشکر و قدردانی

من بدون کمک و یاری فنی بسیاری از افراد، قادر به نوشتن این کتاب نبودم. خصوصا از خانواده ام تشکر می‌کنم. تلاش و زمان مورد نیاز برای نوشتن این کتاب قابل اندازه گیری نیست. تنها این را می‌دانم بدون حمایت همسرم کریستین و دو پسر (آیدان و گرنت)، نمی‌توانستم این کتاب را تولید کنم. زمان‌های زیادی بود که کتاب مانع از کنار هم بودن ما می‌شد. حال که کتاب پایان یافته است، واقعا به دنبال ادامه‌ی ماجراجویی‌های زندگی مشترک‌مان می‌روم. برای این ویرایش از کتاب، افراد مهمی به من کمک کردند. کریستوف نازاری، که با هم در پروژه‌ی کتاب‌های مختلف کار کرده ایم، کتاب را بررسی کرد و مطمئن شد هر آنچه گفته ام به بهترین وجه ممکن بیان شده باشد. او اثر مهمی بر کیفیت کتاب داشته است. همچون همیشه، تیم ویراستاران انتشارات میکروسافت نیز برای همکاری عالی بودند. تشکر مخصوص خود را نیز شامل بن ریان، ولری وولی و دون سوسگریو می‌کنم. همچنین از جین فیندلی و سو مک کلانگ برای ویرایش و پشتیبانی تولید تشکر می‌کنم.

پشتیبانی از کتاب

تمام تلاش ممکن برای صحت کتاب انجام شده است. تغییرات و اصلاحات احتمالی به پایگاه دانش میکروسافت اضافه می‌شود که از طریق وب سایت پشتیبانی و کمک میکروسافت در دسترس است. پشتیبانی از کتاب و دسترسی به مقالات در وب سایت زیر:

<http://www.microsoft.com/learning/support/books>

سوالات خود پیرامون کتاب را به آدرس msinput@microsoft.com ارسال کنید.

ما منتظر نظرات شما هستیم

هر گونه نظری پیرامون کتاب را ارزش می‌نهم. لطفا نظرات خود را از طریق نظر سنجی زیر با ما در میان بگذارید:

<http://www.microsoft.com/learning/booksurvey>

نکته

امیدواریم شما نظرات دقیق و جزیی برای ما ارسال کنید. اگر سوالی پیرامون برنامه‌های انتشارات، عناوین در حال چاپ و ... دارید از طریق توییت <http://twitter.com/MicrosoftPress> با ما در ارتباط باشید.

فصل ۱: مدل اجرایی CLR

دانت میکروسافت مفاهیم، تکنولوژی و واژه‌های جدیدی را معرفی می‌کند. هدف من در این فصل این است که خلاصه‌ای از طراحی دانت برای شما بیان کرده و برخی از تکنولوژی‌های جدید فریمورک را برای شما معرفی کنم، و واژه‌های جدیدی را که با آنها کار خواهیم کرد را برایتان تعریف کنم. همچنین پیرامون فرآیند ساخت برنامه از سورس کد و یا ایجاد کامپوننت‌های قابل توزیع (فایل‌ها) که شامل نوع‌ها (کلاس، ساختار و ...) می‌شوند، و چگونگی اجرای برنامه‌هایتان توضیح خواهم داد.

کامپایل سورس کد به ماژول مدیریت شده

شما تصمیم گرفته اید دانت فریمورک را به عنوان پلتفرم برنامه‌نویسی خود انتخاب کنید. بسیار عالی است. اولین قدم شما تعیین نوع برنامه یا کامپوننتی است که قصد دارید بسازید. تصور کنید همه چیز را طراحی کرده اید، مشخصات برنامه آماده‌اند و شما آماده‌ی برنامه‌نویسی هستید.

اکنون باید انتخاب کنید از چه زبان برنامه‌نویسی استفاده کنید. این انتخاب مشکلی است چراکه زبان‌های مختلف، قابلیت‌های متفاوتی ارائه می‌دهند. برای نمونه در C/C++ مدیریت نشده، شما کنترل بسیار سطح پایینی از سیستم دارید. شما می‌توانید حافظه را آنطور که می‌خواهید مدیریت کنید، تردها را به آسانی بسازید و ویژوال بیسیک C# به شما اجازه می‌دهد برنامه‌های گرافیکی را به سرعت تولید کنید و به سادگی اشیاء COM و پایگاه داده را کنترل کنید.

اجرای زبان مشترک Common Language Runtime (CLR) دقیقاً چیزی است که نامش بیان می‌کند: یک اجرایی که توسط زبان‌های برنامه‌نویسی مختلف و متفاوتی قابل استفاده است. ویژگی‌های اصلی CLR (همچون مدیریت حافظه، بارگذاری اسمبلی، امنیت، مدیریت اکسپشن و همزمانی تردها) برای تمام زبان‌های برنامه‌نویسی که آن را هدف قرار می‌دهند، در دسترس می‌باشد. برای نمونه، CLR از اکسپشن‌ها برای گزارش خطاها استفاده می‌کند، پس هر زبانی که با CLR کار می‌کند خطاها را از طریق اکسپشن‌ها دریافت می‌کند. نمونه دیگر اینکه CLR به شما اجازه می‌دهد ترد بسازید، پس هر زبانی که با CLR کار می‌کند می‌تواند ترد بسازد.

در واقع، در زمان اجرا، CLR درباره‌ی اینکه از چه زبان برنامه‌نویسی برای نوشتن سورس کد استفاده شده است، چیزی نمی‌داند. این بدان معنی است که شما باید زبانی را انتخاب کنید که بتوانید خواسته‌های خود را به آسانی در آن بیان کنید. شما می‌توانید کد خود را به هر زبانی بنویسید مشروط به اینکه زبان مورد نظر با CLR کار کند.

پس اگر آنچه گفته شد صحیح است، مزیت استفاده از یک زبان نسبت به دیگری چیست؟ خوب، من به کامپایلرها به عنوان بررسی‌کننده‌ی نحو دستورات و آنالیزر "کد صحیح" نگاه می‌کنم. آن‌ها کد شما را بررسی می‌کنند و مطمئن می‌شوند آنچه نوشته اید صحیح باشد و سپس کد خروجی که هدف شما را می‌رساند را تولید می‌کنند. زبان‌های برنامه‌نویسی به شما اجازه می‌دهند با نحوهای متفاوتی کد خود را بنویسید. ارزش این انتخاب را دست کم نگیرید. برای نمونه در برنامه‌های محاسباتی و مالی بیان اهداف شما به کمک نحو APL زمان کدنویسی و توسعه‌ی آن را نسبت به نحو Perl کاهش می‌دهد.

میکروسافت کامپایلرهای مختلفی ساخته است که با CLR کار می‌کنند: C++/CLI، C#، Visual Basic، F#، Iron Python، Iron Ruby و یک اسمبلر زبان میانی Intermediate Language (IL) Assembler. علاوه بر میکروسافت، شرکت‌ها و دانشگاه‌های مختلف هم کامپایلرهایی تولید کرده‌اند که CLR را هدف قرار می‌دهد، من از وجود کامپایلرهایی برای Ada، APL، Caml، CoBol، Eiffel، Fortran، Haskell، Logo، LISP، Lexico، ML، Mercury، Oberon، Mondorian، Pascal، Perl، PHP، Prolog، RPG، Scheme، Smalltalk و Tcl/Tk مطلع هستم.

شکل ۱-۱ فرآیند کامپایل فایل‌های سورس کد را نشان می‌دهد. همانطور که شکل نشان می‌دهد، شما می‌توانید سورس کدهای خود را در هر زبان برنامه‌نویسی که CLR را پشتیبانی می‌کند، بنویسید. سپس با کامپایلر متناظر با زبان خود، نحو دستورات را بررسی کرده و سورس کد را آنالیز کنید. بدون توجه به زبانی که انتخاب می‌کنید، نتیجه یک ماژول مدیریت شده است. یک ماژول مدیریت شده یک فایل استاندارد اجرایی قابل حمل (PE32) تحت ویندوز ۳۲ بیتی یا یک فایل استاندارد اجرایی قابل حمل (PE32+) تحت ویندوز ۶۴ بیتی می‌باشد که برای اجرا نیاز به CLR دارد. در واقع، اسمبلی‌های مدیریت شده همواره از ویژگی‌های جلوگیری از اجرای داده (Data Execution Prevention (DEP) و تصادفی سازی قالب فضای آدرس Address Space Layout Randomization (ASLR) در ویندوز بهره می‌برند؛ این دو ویژگی امنیت کل سیستم را افزایش می‌دهد.



شکل ۱-۱ کامپایل سورس کد به ماژول های مدیریت شده

جدول ۱-۱ بخش های یک ماژول مدیریت شده را نشان می دهد.

جدول ۱-۱ بخش های یک ماژول مدیریت شده

بخش	توضیح
هدر PE32 یا PE32+	هدر استاندارد Windows PE از فایل که شبیه هدر Common Object File Format (COFF) است. اگر هدر از فرمت PE32 استفاده کند فایل می تواند در ویندوز ۳۲ بیتی یا ۶۴ بیتی اجرا شود. اگر هدر از فرمت PE32+ استفاده کند، فایل برای اجرا نیاز به ویندوز ۶۴ بیتی دارد. این هدر نوع فایل را نیز مشخص می کند: GUI، .CUI، DLL و شامل برچسب زمان ساخت فایل نیز می باشد. برای ماژول هایی که فقط شامل کد IL هستند اطلاعات هدر PE32(+) نادیده گرفته می شود. برای ماژول هایی که شامل کد اصلی پردازنده هستند، این هدر اطلاعات مرتبط با کد اصلی پردازنده را داراست.
هدر CLR	شامل اطلاعاتی است که این را به یک ماژول مدیریت شده تبدیل می کند. (اطلاعات توسط ابزارها و CLR درک می شوند). هدر شامل نسخه ی CLR مورد نیاز، برخی پرچم ها، علامت متادیتای MethodDef از متد آغازین ماژول مدیریت شده (متد Main) و مکان و اندازه ی متادیتای ماژول، منابع، نام قوی، برخی پرچم ها و دیگر موارد غیر مهم می باشد.
متادیتا (فرداده) ^۱	هدر ماژول مدیریت شده شامل جدول های متادیتا می باشد. دو نوع اصلی از این جدول ها وجود دارد: جدول هایی که نوع ها و اعضای تعریف شده در کد شما را توصیف می کنند و جدول هایی که نوع ها و اعضای ارجاع داده شده توسط سورس کدتان را توصیف می کنند.
کد IL	کدی است که کامپایلر در نتیجه ی کامپایل سورس کد تولید می کند. در زمان اجرا، CLR کد IL را به دستورات اصلی پردازنده کامپایل می کند.

کامپایلرهای کد اصلی، کدی را تولید می کنند که ویژه ی یک معماری خاص از پردازنده ها همچون x86، x64 یا IA64 می باشد. تمام کامپایلرهای مطابق با CLR به جای کد اصلی کد IL تولید می کنند. (در این فصل بیشتر وارد جزئیات کد IL می شوم). گاهی به کد IL، کد مدیریت شده نیز گفته می شود چرا که CLR اجرای آن را مدیریت می کند.

علاوه بر کد IL، هر کامپایلری که با CLR کار می کند لازم است متادیتای کاملی در هر ماژول مدیریت شده قرار دهد. به طور خلاصه، متادیتا مجموعه جدول های اطلاعاتی است که چیزهایی که در ماژول تعریف شده اند همچون نوع ها و اعضای آن ها را توصیف می کند. به علاوه، متادیتا دارای جدول هایی است که چیزهایی که ماژول به آن ها ارجاع داده است، همچون نوع های وارد شده و اعضای آن ها را تعیین می کند. متادیتا مجموعه ای برتر از تکنولوژی های قدیمی تر مثل TypeLibrary از COM و فایل های Interface Definition Language (IDL) است. نکته مهم این است که متادیتای CLR بسیار کاملتر است و بر خلاف آن ها، متادیتا همواره همراه با فایلی است که حاوی کد IL می باشد. در واقع، متادیتا همیشه درون همان فایل EXE/DLL که کد در آن قرار دارد، تعبیه شده است و جداسازی آن غیر ممکن است. به دلیل آنکه کامپایلر متادیتا و کد را همزمان تولید می کند و در ماژول مدیریت شده خروجی قرار می دهد، متادیتا و کد IL همواره با همدیگر هماهنگ هستند. متادیتا کاربردهای بسیاری دارد که برخی از آن ها عبارتند از:

- متادیتا نیاز به فایل های اصلی کتابخانه و هدر C/C++ را هنگام کامپایل حذف می کند چرا که تمام اطلاعات در مورد نوع ها و اعضای ارجاع داده شده درون همان فایلی است که کد IL، آن نوع ها و اعضا را به کار می گیرد. کامپایلرهای می توانند متادیتا را مستقیماً از ماژول مدیریت شده بخوانند.

¹ Metadata

- ویژوال استودیو برای کمک به شما در نوشتن کد از متادیتا استفاده می‌کند. ویژگی IntelliSense متادیتا را تحلیل می‌کند تا به شما بگوید یک نوع چه متدها، ویژگی‌ها، رویدادها و فیلدهایی را ارائه می‌دهد و در خصوص متدها، چه پارامترهایی را به عنوان ورودی می‌پذیرد.
- فرآیند بازبینی که توسط CLR انجام می‌شود از متادیتا استفاده می‌کند تا اطمینان حاصل کند که شما تنها عملیات‌های "نوع-امن"^۲ را انجام می‌دهید. (فرآیند بازبینی توضیح داده خواهد شد).
- متادیتا اجازه می‌دهد که فیلدهای یک شی به فرم بلوکی از حافظه سریالی شده^۳، به ماشین دیگری ارسال گردد و در آنجا غیرسریالی شده^۴ و حالت شی در ماشین مقصد بازسازی شود.
- متادیتا به جمع آوری کننده زباله^۵ (GC) اجازه می‌دهد دوره حیات اشیاء را پیگیری کند. برای هر شی، GC می‌تواند نوع شی را تعیین کند و از روی متادیتا بفهمد کدام فیلدها درون این شی به اشیاء دیگر ارجاع می‌کنند.

در فصل ۲، "ساخت، بسته‌بندی، نصب و مدیریت برنامه‌ها و نوع‌ها" متادیتا را به تفصیل توضیح خواهیم داد.

سی‌شارپ، ویژوال بیسیک، اف‌شارپ و اسمبلر IL همواره ماژول‌هایی حاوی کد مدیریت‌شده (IL) و داده مدیریت‌شده (نوع‌های داده‌ای که توسط GC جمع آوری می‌شوند) تولید می‌کنند. کاربران نهایی برای اجرای هر ماژولی که حاوی کد مدیریت‌شده و/یا داده مدیریت‌شده است باید CLR (که در حال حاضر به عنوان بخشی از دات‌نت عرضه می‌شود) را بر روی ماشین خود نصب داشته باشند درست همانطور که باید کتابخانه Microsoft Foundation Class (MFC) یا DLL‌های ویژوال بیسیک نصب شده باشند تا بتوانند برنامه‌های MFC یا ویژوال بیسیک ۶ را اجرا کنند.

به صورت پیشفرض، کامپایلر C++ مایکروسافت، ماژول‌های EXE/DLL تولید می‌کند که شامل کد مدیریت نشده (اصلی^۶) است و داده‌های مدیریت نشده (حافظه اصلی) را در زمان اجرا دستکاری می‌کنند. این ماژول‌ها برای اجرا به CLR نیاز ندارند. هر چند با تعیین سویچ خط فرمان /CLR کامپایلر C++ ماژول‌هایی را تولید می‌کند که حاوی کد مدیریت‌شده است و البته برای اجرای این کد CLR نیاز است. در میان تمام کامپایلرهای مایکروسافت که نام بردیم، C++ تنها کامپایلری است که اجازه‌ی تولید هر دو کد مدیریت‌شده و نشده را به برنامه‌نویس داده و ترکیب این دو را در یک ماژول ممکن می‌سازد. این کامپایلر البته تنها کامپایلری است که اجازه‌ی تعریف نوع داده‌ای مدیریت‌شده و نشده در سورس برنامه را به برنامه‌نویس می‌دهد. این انعطاف پذیری که کامپایلر C++ مایکروسافت ارائه می‌دهد به صورت غیرموازی با دیگر کامپایلرهاست چرا که به برنامه‌نویسان اجازه می‌دهد از کدهای کنونی اصلی C/C++ خود درون کد مدیریت‌شده استفاده کنند و همزمان از نوع‌های مدیریت‌شده برای کار خود بهره گیرند.

ترکیب ماژول‌های مدیریت شده به اسمبلی‌ها

CLR در واقع با ماژول‌ها کار نمی‌کند، بلکه با اسمبلی‌ها کار می‌کند. یک اسمبلی^۷ مفهومی انتزاعی است که درک آن در ابتدا مشکل می‌باشد. اولاً، یک اسمبلی، گروه بندی منطقی یک یا چند ماژول یا فایل‌های منبع است. دوماً، یک اسمبلی کوچکترین واحد با قابلیت استفاده مجدد، امنیت و نسخه‌بندی است. بسته به انتخاب‌هایی که با کامپایلرها و ابزارها می‌کنید، شما می‌توانید یک اسمبلی تک فایلی یا چندفایلی تولید کنید. در دنیای CLR یک اسمبلی چیزی است که به آن یک کامپوننت^۸ می‌گوییم.

در فصل ۲، اسمبلی‌ها را با جزئیات توضیح می‌دهم، پس در اینجا وقت زیادی بر روی آن‌ها صرف نمی‌کنیم. فقط در این جا این را نشان می‌دهم که مفاهیم دیگری نیز وجود دارد که روشی را برای شناخت گروهی از فایل‌ها به عنوان یک تک موجودیت، ارائه می‌کند.

شکل ۱-۲ نشان می‌دهد اسمبلی‌ها چیستند. در این شکل، تعدادی ماژول مدیریت‌شده و فایل‌های منبع (یا داده) توسط یک ابزار پردازش می‌شوند. این ابزار یک تک فایل PE32(+) تولید می‌کند که گروه بندی منطقی فایل‌ها را نشان می‌دهد. اتفاقی که رخ می‌دهد اینست که این فایل PE32(+) شامل بلوکی از داده به نام مانیفست^۹ است. مانیفست مجموعه‌ی دیگری از جدول‌های متادیتاست. این جدول‌ها، فایل‌های سازنده اسمبلی، نوع‌های صادرشده‌ی عمومی که توسط فایل‌های اسمبلی پیاده‌سازی شده‌اند و فایل‌های منبع و داده که با اسمبلی همراه هستند را توصیف می‌کنند.

² Type-safe

³ Serialize

⁴ Deserialize

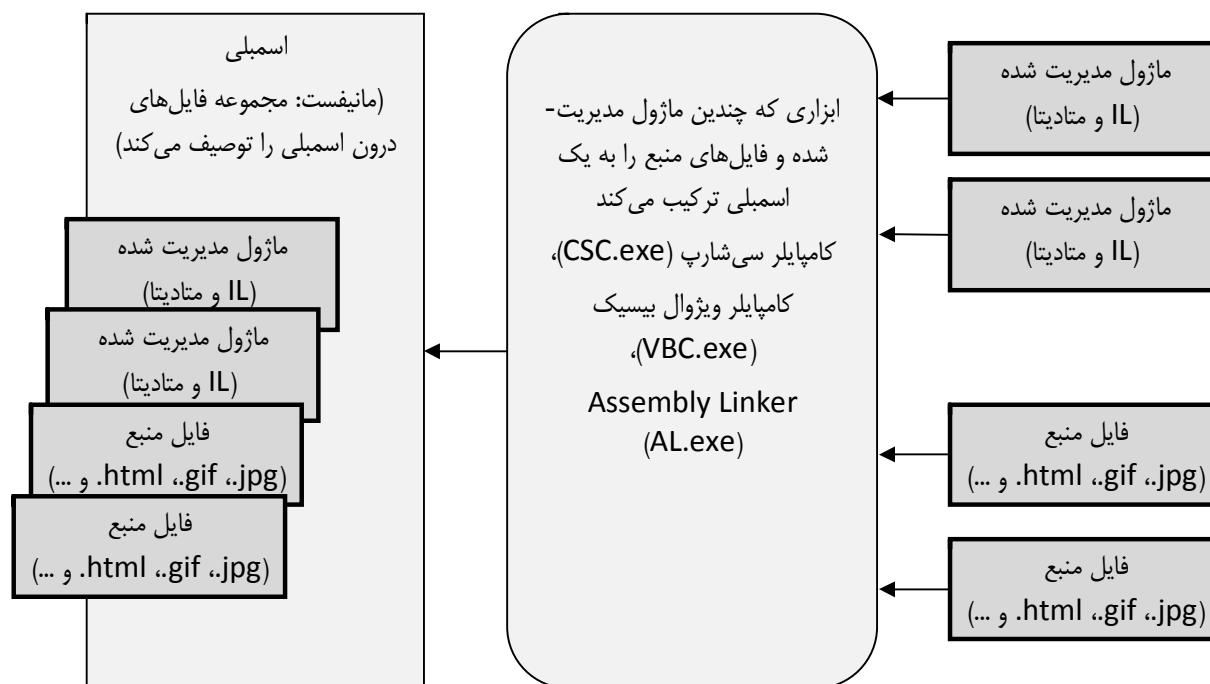
⁵ Garbage Collector

⁶ Native

⁷ Assembly

⁸ Component

⁹ Manifest



شکل ۲-۱ ترکیب ماژول‌های مدیریت‌شده به اسمبلی‌ها

به صورت پیشفرض، کامپایلرها کار تبدیل ماژول مدیریت‌شده به یک اسمبلی را انجام می‌دهند. کامپایلر سی‌شارپ، ماژول مدیریت‌شده‌ای تولید می‌کند که حاوی یک مانیفست است. این مانیفست بیان می‌کند که اسمبلی فقط از یک فایل تشکیل شده است. پس، برای پروژه‌هایی که تنها یک ماژول مدیریت‌شده دارند و فایل منبع (یا داده) ندارند، اسمبلی همان ماژول مدیریت‌شده است و شما مرحله اضافی دیگری در فرآیند ساخت خود ندارید. اگر شما می‌خواهید گروهی از فایل‌ها را درون یک اسمبلی قرار دهید باید با ابزارهای دیگر (همانند **Assembly Linker (AL.exe)**) و سوییچ‌های خط فرمان آن‌ها آشنا باشید. این ابزارها را در فصل ۲ توضیح می‌دهم.

یک اسمبلی به شما اجازه می‌دهد که مفاهیم فیزیکی و منطقی یک کامپوننت با قابلیت استفاده مجدد، امنیت و نسخه‌بندی را از یکدیگر تفکیک کنید. این که چگونه کد و منابع را تقسیم کنید، به شما بستگی دارد. برای نمونه شما می‌توانید منابع و نوع‌هایی که کمتر استفاده می‌شوند را در فایلی جداگانه درون اسمبلی قرار دهید. این فایل‌های جداگانه می‌توانند بر طبق درخواست، وقتی در زمان اجرا به آن‌ها نیاز است از وب دانلود شوند. اگر فایل‌ها اصلاً استفاده نشدند، هرگز دانلود نمی‌شوند که منجر به کاهش فضای مصرف شده و زمان نصب برنامه می‌شود. اسمبلی‌ها به شما اجازه می‌دهند که فایل‌هایتان را تفکیک کنید در حالیکه به فایل‌ها به عنوان یک تک مجموعه نگاه می‌کنید.

ماژول‌های یک اسمبلی همچنین حاوی اطلاعاتی پیرامون اسمبلی‌های ارجاع داده شده شامل نسخه‌ی آن‌ها هستند. این اطلاعات، یک اسمبلی را خود توصیف گر می‌سازد. به بیان دیگر، CLR می‌تواند وابستگی‌های آنی اسمبلی برای اجرای کد را تعیین کند. هیچ اطلاعات اضافی دیگری در رجیستری یا در سرویس‌های دامنه اکتیو‌دایرکتوری¹⁰ (AD DS) نیاز نیست. به دلیل آنکه هیچ اطلاعات اضافی دیگری نیاز نیست، استفاده از اسمبلی‌ها بسیار آسانتر از کامپوننت‌های مدیریت نشده است.

بارگذاری اجرایی زبان مشترک

هر اسمبلی‌ای که می‌سازید می‌تواند یک برنامه اجرایی یا یک فایل DLL شامل نوع‌هایی برای استفاده توسط یک برنامه اجرایی باشد. CLR مسئول مدیریت اجرای کدهای درون این اسمبلی‌ها می‌باشد. به این معنی که دات‌نت فریمورک باید روی ماشین نصب شده باشد. مایکروسافت یک بسته قابل توزیع ارائه کرده است که شما می‌توانید به کمک آن دات‌نت را به صورت رایگان بر روی ماشین‌های مشتریان خود نصب کنید. برخی از نسخه‌های ویندوز همراه با دات‌نت نصب شده فروخته می‌شوند.

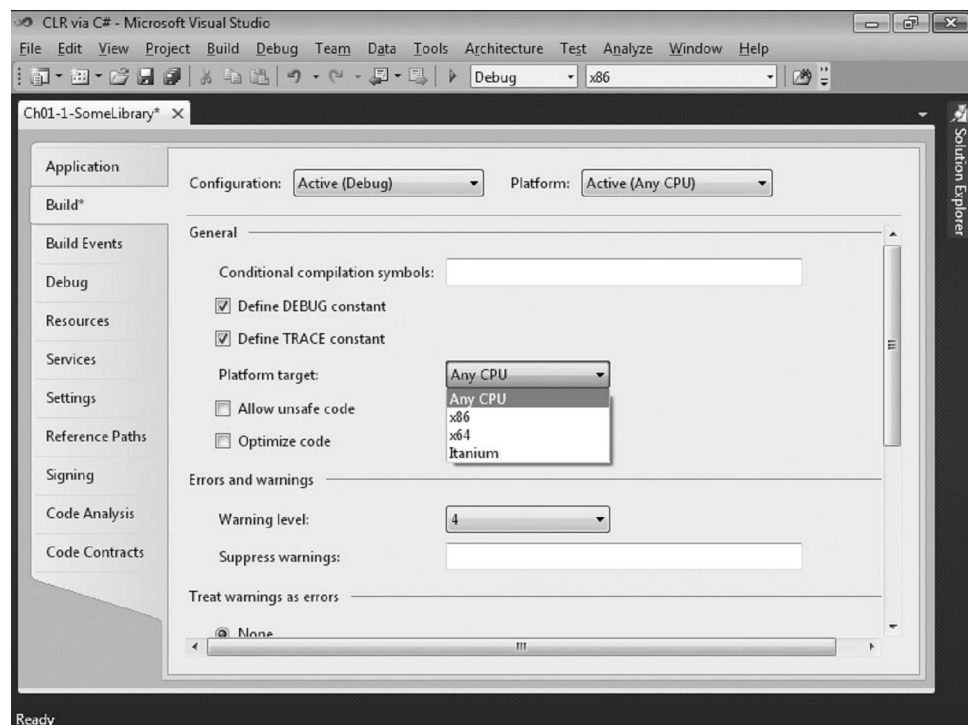
¹⁰ Active Directory Domain Services

شما می‌توانید با جستجوی فایل `MSCorEE.dll` در پوشه `%SystemRoot%\System32` تعیین کنید آیا دات‌نت فریمورک نصب شده است یا خیر. وجود این فایل به معنی اینست که دات‌نت فریمورک نصب شده است. هر چند، نسخه‌های مختلف دات‌نت فریمورک می‌توانند به صورت همزمان در یک ماشین نصب شوند. برای تعیین نسخه‌های دات‌نت فریمورک نصب شده، زیرکلیدهای رجیستری زیر را بررسی کنید:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP

SDK¹¹ دات‌نت شامل یک برنامه خط فرمان به نام `CLRVer.exe` است که تمامی نسخه‌های CLR نصب شده بر روی ماشین را نشان می‌دهد. این برنامه همچنین می‌تواند به کمک سویچ `-all` یا با ارسال شناسه‌ی پردازنده مورد نظر شما نشان دهد پردازنده‌های در حال اجرا در ماشین از چه نسخه‌ی CLR استفاده می‌کنند. پیش از آنکه درباره بارگذاری CLR صحبت کنیم نیاز است کمی پیرامون نسخه‌های ۳۲ و ۶۴ بیتی ویندوز صحبت کنیم. اگر اسمبلی شما فقط شامل کد مدیریت‌شده‌ی نوع-امن باشد، شما کدی نوشته اید که می‌بایست در هر دو نسخه ۳۲ بیتی و ۶۴ بیتی ویندوز کار کند. در واقع، فایل تولیدی `EXE/DLL` که کامپایلر آن را ایجاد کرده است بر روی ویندوز ۳۲ بیتی و همچنین نسخه‌های `x64` و `IA64` از ویندوز ۶۴ بیتی به خوبی اجرا می‌شود. به بیان دیگر، این تک فایل بر روی هر ماشینی که یک نسخه از دات‌نت فریمورک (البته با شماره نسخه صحیح) بر روی آن نصب است، اجرا می‌شود.

در مواقع بسیار نادری، برنامه‌نویسان می‌خواهند کدی بنویسند که بر روی نسخه‌ی خاصی از ویندوز کار کند. برنامه‌نویسان چنین کاری را احتمالاً در موقع نوشتن کد نا امن^{۱۲} و یا کار با کد مدیریت‌نشده که یک معماری پردازنده خاص را هدف قرار داده است، انجام می‌دهند. برای کمک به برنامه‌نویسان، کامپایلر سی‌شارپ سویچ خط فرمان `/platform` را ارائه می‌دهد. این سویچ به شما اجازه می‌دهد که تعیین کنید اسمبلی خروجی بتواند فقط بر روی ماشین‌های `x86` که ویندوز ۳۲ بیتی دارند، فقط ماشین‌های ۶۴ بیتی که ویندوز ۶۴ بیتی دارند یا فقط ماشین‌های `Intel Itanium` که ویندوز ۶۴ بیتی دارند، اجرا بشود. اگر شما پلتفرمی را تعیین نکنید، پیش فرض `anycpu` (هر پردازنده‌ای) انتخاب می‌شود که به این معنیست که اسمبلی می‌تواند در هر نسخه‌ای از ویندوز اجرا شود. کاربران ویژوال استودیو می‌توانند پلتفرم هدف در پروژه خود را با نمایش صفحات مشخصات پروژه از طریق منوی `Project` و سپس گزینه `Properties` پروژه و با کلیک بر تب `Build` و انتخاب یکی از گزینه‌های لیست `Platform Target` (شکل ۳-۱)، تعیین کنند.



شکل ۳-۱ تنظیم پلتفرم هدف به کمک ویژوال استودیو

¹¹ Sample Document Kit

¹² Unsafe code

بسته به سویچ پلتفرم، کامپایلر سی شارپ، یک اسمبلی که شامل هدر PE32 یا PE32+ می‌باشد تولید می‌کند و نیز کامپایلر، معماری پردازنده مورد نظر (و یا بی تفاوت به معماری) را درون هدر قرار می‌دهد. مایکروسافت ۲ برنامه خط فرمان SDK به نام‌های DumpBin.exe و CorFlags.exe را ارائه می‌دهد که شما می‌توانید برای بررسی اطلاعات هدر موجود در مازول‌های مدیریت‌شده از آن‌ها استفاده کنید.

وقتی شما یک فایل اجرایی را اجرا می‌کنید، ویندوز، هدر فایل EXE را بررسی می‌کند تا تعیین کند که برنامه به فضای آدرس ۳۲ بیتی و یا ۶۴ بیتی نیاز دارد. یک فایل با هدر PE32 می‌تواند با فضای آدرس ۳۲ بیتی یا ۶۴ بیتی اجرا شود و یک فایل با هدر PE32+ به فضای آدرس ۶۴ بیتی نیاز دارد. ویندوز همچنین اطلاعات معماری پردازنده که درون هدر تعبیه شده است را بررسی می‌کند تا مطمئن شود که با پردازنده‌ی موجود بر روی ماشین منطبق باشد. در پایان، ویندوزهای ۶۴ بیتی تکنولوژی‌ای ارائه می‌دهند که به برنامه‌های ۳۲ بیتی اجازه‌ی اجرا می‌دهد. این تکنولوژی (WoW64 (Windows On Windows 64) نامیده می‌شود. این تکنولوژی حتی اجازه می‌دهد برنامه‌های ۳۲ بیتی با کد اصلی x86 بر روی ماشین‌های Itanium اجرا شوند، علت آنست که WoW64 می‌تواند مجموعه دستورات x86 را با کاهش قابل توجه سرعت، شبیه سازی کند.

جدول ۱-۲ دو چیز را نشان می‌دهد. اول اینکه نشان می‌دهد وقتی شما سویچ‌های مختلف خط فرمان platform/ را برای کامپایلر سی شارپ تعیین می‌کنید چه نوع مازول مدیریت‌شده‌ای حاصل می‌شود. دوم اینکه، نشان می‌دهد آن برنامه چگونه بر روی نسخه‌های مختلف ویندوز اجرا می‌شود.

جدول ۱-۲ اثرات platform/ بر روی مازول تولید شده و در زمان اجرا

سویچ platform/	مازول حاصل	مدیریت‌شده	ویندوز x86	ویندوز x64	ویندوز IA64
anycpu (پیشفرض)	مستقل از معماری/PE32	به عنوان برنامه ۳۲ بیتی اجرا می‌شود	به عنوان برنامه ۶۴ بیتی اجرا می‌شود	به عنوان برنامه ۶۴ بیتی اجرا می‌شود	
x86	PE32/x86	به عنوان برنامه ۳۲ بیتی اجرا می‌شود	به عنوان برنامه ۳۲ بیتی اجرا می‌شود	به عنوان برنامه WoW64 اجرا می‌شود	به عنوان برنامه WoW64 اجرا می‌شود
x64	PE32+/x64	اجرا نمی‌شود	اجرا نمی‌شود	به عنوان برنامه ۶۴ بیتی اجرا می‌شود	اجرا نمی‌شود
Itanium	PE32+/Itanium	اجرا نمی‌شود	اجرا نمی‌شود	اجرا نمی‌شود	به عنوان برنامه ۶۴ بیتی اجرا می‌شود

بعد از آنکه ویندوز هدر فایل EXE را بررسی نمود و تعیین کرد که یک پردازنده ۳۲ بیتی، یک پردازنده ۶۴ بیتی و یا یک پردازنده WoW64 بسازد، ویندوز نسخه‌ی x86، x64 و یا IA64 از MSCorEE.dll را در فضای آدرس پردازنده بارگذاری می‌کند. در نسخه x86 ویندوز، نسخه x86 از MSCorEE.dll در پوشه‌ی C:\Windows\System32 قرار دارد. در نسخه x64 یا IA64 از ویندوز، نسخه x86 از MSCorEE.dll در پوشه‌ی C:\Windows\SysWow64 می‌باشد در حالیکه نسخه‌ی ۶۴ بیتی (x64 یا IA64) را می‌توان در پوشه C:\Windows\System32 (به دلایل سازگاری با نسخه‌های قدیمی) پیدا کرد. سپس، ترد اصلی پردازنده، متدی را درون MSCorEE.dll فراخوانی می‌کند. این متد، CLR را مقداردهی اولیه کرده، اسمبلی EXE را بارگذاری می‌کند و سپس متد آغازین (Main) آن را فراخوانی می‌نماید. در این لحظه برنامه مدیریت‌شده در حال اجراست.^{۱۳}

نکته اسمبلی‌هایی که با نسخه‌ی ۱.۰ یا ۱.۱ از کامپایلر سی شارپ مایکروسافت ساخته شده اند، شامل هدر PE32 بوده و مستقل از معماری پردازنده می‌باشند. هرچند که CLR این اسمبلی‌ها را "فقط x86" در نظر می‌گیرد. برای فایل‌های اجرایی، این کار احتمال اینکه برنامه در سیستم ۶۴ بیتی اجرا شود را به دلیل آنکه فایل اجرایی در WoW64 بارگذاری می‌شود افزایش خواهد داد و برای پردازنده، محیطی بسیار شبیه به آنچه در ویندوزهای ۳۲ بیتی x86 خواهد داشت، فراهم می‌کند.

اگر یک برنامه مدیریت‌نشده، LoadLibrary را فراخوانی کند تا یک اسمبلی مدیریت‌شده را بارگذاری کند، ویندوز می‌داند که برای پردازش کد درون اسمبلی باید CLR را بارگذاری و مقداردهی اولیه کند (اگر زودتر بارگذاری نشده باشد). البته در این سناریو، پردازنده قبلا در حال اجراست و این ممکن است

^{۱۳} کد شما می‌تواند ویژگی Is64BitOperatingSystem از Environment را برای تعیین اینکه ویندوز در حال اجرا ۶۴بیتی است یا خیر، بررسی کند. کد شما همچنین می‌تواند ویژگی Is64BitProcess از Environment را بررسی کند تا تعیین نمایید آیا کد در فضای آدرس ۶۴بیتی در حال اجراست یا خیر.

کاربرد اسمبلی را محدود کند. برای نمونه، یک اسمبلی مدیریت شده که با سویچ `/platform:x86` کامپایل شده است اصلا قادر نخواهد بود که درون یک پردازنده ۶۴ بیتی بارگذاری شود، در حالیکه فایل اجرایی‌ای که با همان سویچ کامپایل شده است می‌تواند در `Wow64` بر روی یک ویندوز ۶۴ بیتی بارگذاری شود.

اجرای کد اسمبلی شما

همانطور که قبلا گفته شد، اسمبلی‌های مدیریت شده هم شامل متادیتا و هم شامل کد IL هستند. IL یک زبان ماشین مستقل از پردازنده است که توسط مایکروسافت و پس از مشاوره با چندین نویسنده‌ی زبان/کامپایلرهای آکادمیک و تجاری، ساخته شده است. IL دارای سطحی بسیار بالاتر از اغلب زبان‌های ماشین است. IL می‌تواند به اشیاء دسترسی داشته و آن‌ها را دستکاری کند و دارای دستورات خاصی برای ایجاد و مقداردهی اولیه اشیاء، فراخوانی متدهای مجازی بر روی اشیاء و دستکاری مستقیم عناصر آرایه می‌باشد. آن حتی دستوراتی برای تولید و گرفتن اکسپشن‌ها به منظور مدیریت خطا دارد. شما می‌توانید به IL به عنوان یک زبان ماشین شی‌گرا نگاه کنید.

معمولا، برنامه‌نویسان در یک زبان سطح بالا همچون سی‌شارپ، `C++/CLI` و یا ویژوال بیسیک کد می‌نویسند. کامپایلرهای این زبان‌های سطح بالا، کد IL تولید می‌کنند. هر چند همچون هر زبان ماشین دیگری، IL می‌تواند به زبان اسمبلی نوشته شود و مایکروسافت یک `IL Assembler` به نام `ILAsm.exe` ارائه کرده است. مایکروسافت همچنین یک `IL Disassembler` به نام `ILDasm.exe` فراهم کرده است.

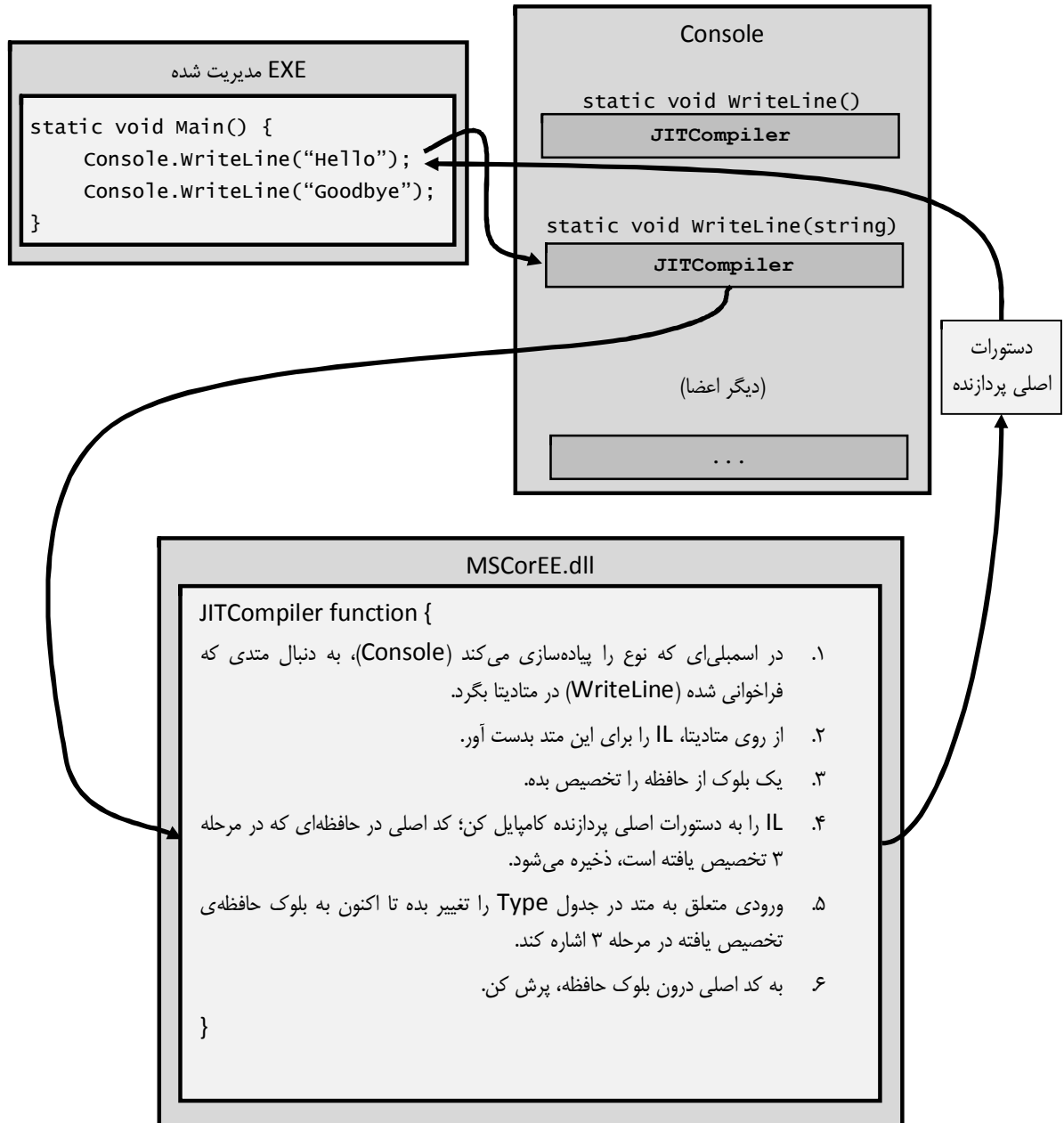
به خاطر داشته باشید که هر زبان سطح بالا، اغلب تنها یک زیر مجموعه از امکاناتی که CLR ارائه می‌دهد را فراهم می‌کند. هرچند، زبان اسمبلی IL به یک برنامه‌نویس اجازه می‌دهد به تمام امکانات CLR دسترسی داشته باشد. پس اگر زبان برنامه‌نویسی مورد انتخاب شما، امکانی از CLR که شما به آن نیاز دارید را پوشش نمی‌دهد، شما می‌توانید آن قطعه از کد خود را به زبان اسمبلی IL و یا یک زبان برنامه‌نویسی دیگر که آن امکان را پوشش می‌دهد، بنویسید.

تنها راهی که از امکانات CLR مطلع شوید مطالعه مستندات مخصوص به خود CLR است. در این کتاب من سعی می‌کنم بر ویژگی‌های CLR و چگونگی پوشش یا عدم پوشش این ویژگی‌ها توسط زبان سی‌شارپ تمرکز کنم. من شک دارم که دیگر کتاب‌ها و مقالات؛ CLR را از دیدگاهی زبانی مورد بررسی قرار دهند و اغلب برنامه‌نویسان تصور می‌کنند که CLR تنها ویژگی‌هایی را فراهم می‌کند که زبان مورد نظرشان آن‌ها را پوشش می‌دهد. تا زمانی که زبان شما به شما اجازه می‌دهد آنچه را می‌خواهید پیاده‌سازی کنید؛ این دیدگاه کدر و تار چیز بدی نیست.

مهم من فکر می‌کنم این قابلیت سویچینگ ساده بین زبان‌های برنامه‌نویسی همراه با یکپارچگی عالی بین زبان‌ها یک ویژگی بسیار عالی از CLR است. متاسفانه، همچنین فکر می‌کنم اغلب برنامه‌نویسان از این ویژگی غفلت می‌کنند. زبان‌های برنامه‌نویسی همچون سی‌شارپ و ویژوال بیسیک زبان‌های عالی برای انجام پروژه‌های ورودی خروجی هستند. APL یک زبان عالی برای انجام محاسبات پیشرفته مهندسی و مالی است. از طریق CLR، شما می‌توانید بخش‌های مربوط به ورودی خروجی برنامه خود را در سی‌شارپ نوشته و سپس بخش‌های محاسبات مهندسی را در APL بنویسید. CLR سطحی از یکپارچگی میان زبان‌ها ارائه می‌دهد که بی سابقه بوده و واقعا برنامه‌نویسی با زبان‌های مختلف را برای بسیاری از پروژه‌های برنامه‌نویسی با ارزش می‌سازد.

برای اجرای یک متد، ابتدا باید کد IL آن به دستورات اصلی پردازنده تبدیل شود. این وظیفه‌ی کامپایلر فقط-در-لحظه (JIT) از CLR است.

شکل ۴-۱ نشان می‌دهد وقتی برای اولین بار یک متد فراخوانی می‌شود چه اتفاقی رخ می‌دهد.



شکل ۴-۱ فراخوانی یک متد برای اولین بار

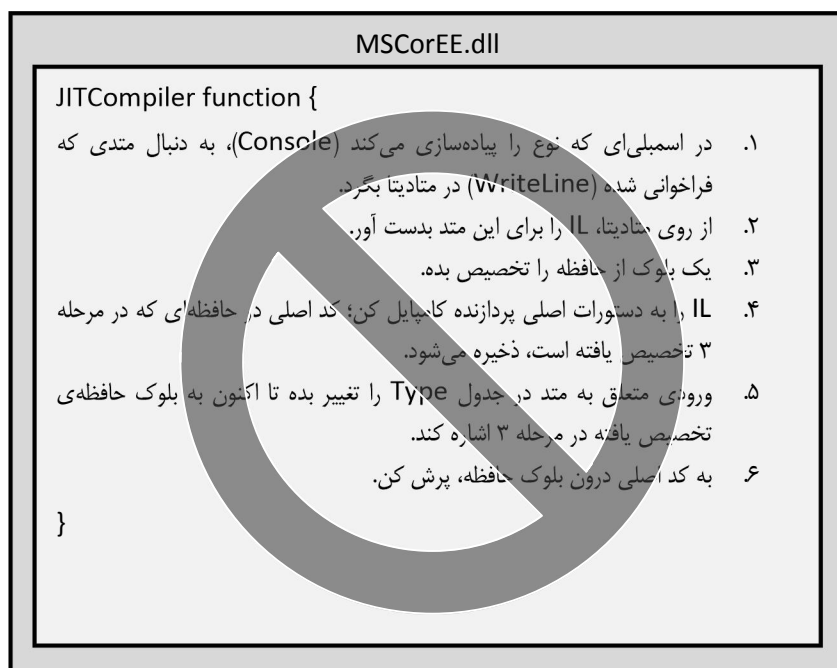
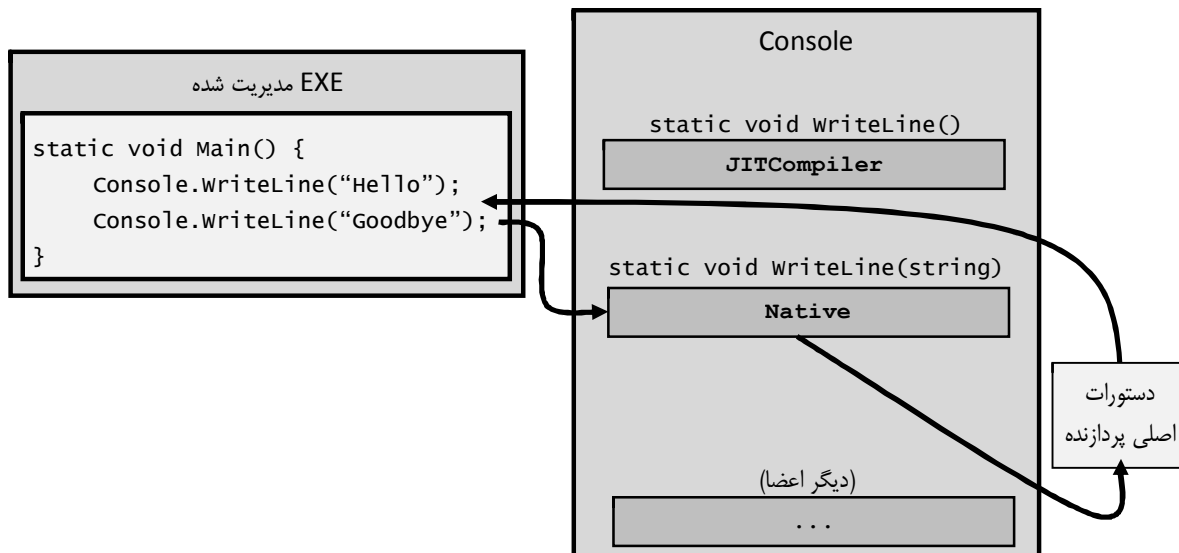
درست قبل از آنکه متد `Main` اجرا شود، تمام نوع‌هایی که توسط کد `Main` ارجاع داده شده‌اند را شناسایی می‌کند. این کار باعث می‌شود که CLR یک ساختمان داده‌ی داخلی برای مدیریت دسترسی به نوع‌های ارجاعی ایجاد کند. در شکل ۴-۱، متد `Main` به یک تک نوع یعنی `Console` ارجاع می‌کند، که باعث می‌شود CLR یک ساختمان داده‌ی داخلی را تخصیص دهد. این ساختمان داده‌ی داخلی شامل یک ورودی^{۱۴} برای هر متد تعریف شده توسط نوع `Console` است. هر ورودی، آدرسی را در خود نگه می‌دارد که در آن آدرس، پیاده‌سازی متد یافت می‌شود. در هنگام مقارده‌ی اولیه‌ی این ساختمان داده، CLR هر ورودی را به یک تابع داخلی و مستند نشده که داخل خود CLR است، تنظیم می‌کند. من این تابع را `JITCompiler` می‌نامم.

زمانی که **Main** اولین فراخوانی خود را به **WriteLine** انجام می‌دهد، تابع **JITCompiler** فراخوانی می‌شود. تابع **JITCompiler** وظیفه کامپایل کد IL متد به دستورات اصلی پردازنده را به عهده دارد. به این دلیل که کد IL، "فقط-در-لحظه" (just-in-time) کامپایل می‌شود، این کامپوننت CLR، اغلب به عنوان یک JITer یا یک JIT Compiler اطلاق می‌شود.

نکته اگر برنامه در نسخه‌ی x86 از ویندوز و یا در WoW64 در حال اجراست، کامپایلر JIT دستورات x86 تولید می‌کند. اگر برنامه شما به عنوان یک برنامه ۶۴ بیتی بر روی نسخه‌ی x64 یا Itanium ویندوز در حال اجراست، کامپایلر JIT به ترتیب دستورات x64 یا IA64 تولید می‌کند.

وقتی **JITCompiler** فراخوانی می‌شود، این تابع می‌داند که چه متدی فراخوانی شده است و چه نوعی این متد را تعریف می‌کند. سپس تابع **JITCompiler** متادیتای اسمبلی را برای یافتن کد IL متد فراخوانی شده، جستجو می‌کند. **JITCompiler** سپس کد را بازبینی کرده و این کد IL را به دستورات اصلی پردازنده کامپایل می‌کند. این دستورات اصلی پردازنده در بلوکی از حافظه که به صورت پویا تخصیص یافته است، قرار می‌گیرند. سپس **JITCompiler** به ورودی برای متد فراخوانی شده در ساختمان داده‌ی داخلی نوع که توسط CLR ایجاد شده است باز می‌گردد و ارجاعی که اولین بار با آن فراخوانی شده بود را با آدرس بلوکی از حافظه حاوی کد اصلی پردازنده که خودش کامپایل کرده، جایگزین می‌کند. سرانجام، **JITCompiler** به کد موجود در بلوک حافظه پرش می‌کند. این کد، پیاده‌سازی متد **WriteLine** (نسخه‌ای که یک پارامتر ورودی **String** دریافت می‌کند) است. وقتی این کد تمام می‌شود و برمی‌گردد، به کد متد **Main** بازگشته و اجرای برنامه ادامه می‌یابد.

اکنون **Main** برای دومین بار **WriteLine** را فراخوانی می‌کند. این بار کد **WriteLine** قبلاً بازبینی و کامپایل شده است. پس فراخوانی، مستقیماً به بلوک حافظه می‌رود و از روی تابع **JITCompiler** کاملاً پرش می‌کند. پس از آنکه متد **WriteLine** کامپایل شد، کنترل اجرای برنامه به **Main** بر می‌گردد. شکل ۱-۵ وضعیت پردازنده وقتی که دومین فراخوانی به **WriteLine** صورت می‌گیرد را نشان می‌دهد.



شکل ۵-۱ فراخوانی یک متد برای دومین بار

فقط وقتی برای اولین بار متد فراخوانی می‌شود کمی زمان هدر می‌رود. تمام دیگر فراخوانی‌ها به متد در بالاترین سرعت کد اصلی اجرا می‌شود چرا که بازیابی و کامپایل به کد اصلی دیگر نیاز نیست.

کامپایلر JIT دستورات اصلی پردازنده را در حافظه پویا نگهداری می‌کند. این یعنی کد کامپایل شده وقتی که برنامه تمام می‌شود از بین خواهد رفت. پس اگر شما برنامه را دوباره در آینده اجرا کنید و یا دو نمونه از برنامه را به صورت همزمان (در دو پردازنده متفاوت سیستم عامل) اجرا کنید، کامپایلر JIT مجبور است کد IL را مجدد به دستورات اصلی کامپایل کند.

برای اغلب برنامه‌ها، کاهش سرعتی که کامپایلر JIT باعث آن می‌شود زیاد نیست. اغلب برنامه‌ها متدها را چندین بار در طول برنامه فراخوانی می‌کنند. این متدها وقتی برنامه اجرا می‌شود فقط یک بار با ضربه عملکردی مواجه می‌شوند. در ضمن، اغلب، زمان بیشتری در داخل متد نسبت به فراخوانی متد صرف می‌شود.

این را هم بدانید که کامپایلر JIT ای CLR کد اصلی را بهینه می‌کند، همانطور که کامپایلر مدیریت نشده ++C این کار را انجام می‌دهد. مجدداً، شاید تولید کد بهینه زمان بیشتری صرف کند اما کد با عملکرد بسیار بهتری نسبت به کد بهینه نشده اجرا می‌شود.

دو سویچ برای کامپایلر سی‌شارپ وجود دارد که بر روی بهینه‌سازی کد اثرگذارند، **/optimize** و **/debug**. جدول زیر اثر این سویچ‌ها بر کیفیت کد IL تولیدی توسط کامپایلر سی‌شارپ و کیفیت کد اصلی تولیدی توسط کامپایلر JIT را نشان می‌دهد:

تنظیمات سویچ کامپایلر	کیفیت کد IL سی‌شارپ	کیفیت کد اصلی JIT
-debug- /optimize- (پیشفرض)	بهینه نشده	بهینه شده
/optimize- /debug(+/full/pdbonly)	بهینه نشده	بهینه نشده
/optimize+ /debug(-/+/full/pdbonly)	بهینه شده	بهینه شده

با **-optimize**، کد IL بهینه نشده‌ی تولیدی توسط کامپایلر سی‌شارپ شامل تعداد زیادی دستورات NOP و پرش‌هایی به خطوط بعدی کد خواهد بود. این دستورات به این دلیل تولید شده‌اند که قابلیت ویرایش-آدامه^{۱۵} و ویژوال استودیو هنگام خطایابی فعال باشد، و همچنین دستورات اضافی کد را برای عملیات خطایابی با قرار دادن نقاط توقف^{۱۶} در دستورات کنترلی مثل **for**، **while**، **do**، **if**، **else**، **try**، **catch** و **finally** آسانتر کنند. وقتی کامپایلر سی‌شارپ کد را بهینه می‌کند، این دستورات انشعابی و NOP را حذف می‌کند که باعث می‌شود کد برای حرکت تک قدمی (اجرای یک دستور) در دیباگر^{۱۷} (خطایابی) مشکل شود ولی جریان کنترلی کد بهینه می‌گردد. همچنین برخی از ارزیابی متدها در داخل دیباگر کار نخواهد کرد. اما به هر حال، کد IL کوچکتر شده باعث می‌شود فایل EXE/DLL حاصل نیز کوچکتر شود و کد IL برای افراد دیگری (مثل من) که از بررسی کد IL برای درک آنچه کامپایلر تولید کرده، لذت می‌برند، آسانتر می‌شود.

افزون بر این، تنها اگر شما سویچ **/debug(+/full/pdbonly)** را مشخص کنید کامپایلر یک فایل پایگاه داده برنامه^{۱۸} (PDB) تولید می‌کند. فایل PDB به کامپایلر در یافتن متغیرهای محلی و نگاشت دستورات IL به کد اصلی کمک می‌کند. سویچ **debug:full** به کامپایلر JIT می‌گوید که شما قصد خطایابی اسمبلی را دارید و کامپایلر JIT کد اصلی حاصل از هر دستور IL را ردیابی می‌کند. این به شما اجازه می‌دهد که از ویژگی دیباگر فقط-در-لحظه و ویژوال استودیو استفاده کرده و یک دیباگر به یک پردازنده در حال اجرا متصل کنید و به راحتی کد را خطایابی کنید. بدون سویچ **debug:full**، به صورت پیشفرض کامپایلر JIT کد IL را با اطلاعات کد اصلی ردیابی نمی‌کند که باعث می‌شود کامپایلر JIT کمی سریعتر عمل کرده و حافظه کمتری مصرف کند. اگر شما یک پردازنده را با دیباگر و ویژوال استودیو اجرا کنید، به کامپایلر JIT دستور داده می‌شود که کد IL را با اطلاعات کد اصلی ردیابی کند (بدون توجه به سویچ **debug**) مگر آنکه شما **Suppress JIT Optimization On Module Load (Managed only)** را در ویژوال استودیو از منوی Tools گزینه Options شاخه‌ی Debugging و گزینه General غیر فعال کنید.

وقتی که یک پروژه‌ی جدید سی‌شارپ در ویژوال استودیو ایجاد می‌کنید، تنظیمات Debug (خطایابی) برای پروژه دارای سویچ‌های **-optimize** و **debug:full** بوده و تنظیمات Release (نهایی) دارای سویچ‌های **optimize+** و **debug:pdbonly** می‌باشد.

برای برنامه‌نویسانی که سابقه کار با C یا ++C مدیریت نشده را دارند، احتمالاً شما درباره عملکرد و سرعت موارد مطرح شده نگران هستید. گذشته از همه این موارد، کد مدیریت نشده برای یک پلتفرم پردازنده خاص کامپایل شده است و وقتی فراخوانی شود به راحتی اجرا خواهد شد. در این محیط مدیریت شده کامپایلر کد در دو مرحله انجام می‌شود. ابتدا، کامپایلر سورس کد را بررسی می‌کند و هر آنچه بتواند در تولید کد IL انجام می‌دهد. اما برای اجرای برنامه، IL باید به دستورات اصلی پردازنده در زمان اجرا تبدیل شود که نیاز به حافظه و زمان پردازنده بیشتری خواهد داشت.

حرفم را باور کنید، چون من با سابقه ++C/C با CLR آشنا شدم؛ خیلی دیرباور و درباره این کارهای اضافی نگران بودم. حقیقت اینست که مرحله دوم کامپایلر که در زمان اجرا رخ می‌دهد، به عملکرد برنامه ضربه زده و حافظه پویا را تخصیص می‌دهد. با این حال، مایکروسافت تلاش بسیار زیادی برای حداقل کردن این سربار اضافی انجام داده است.

اگر شما خیلی دیرباور هستید، شما می‌بایست تعدادی برنامه نوشته و خودتان عملکرد آن‌ها را تست کنید. به علاوه تعدادی برنامه مدیریت شده نسبتاً بزرگ که مایکروسافت یا دیگران تولید کرده‌اند را اجرا کرده و عملکرد آن‌ها را اندازه گیری کنید. من فکر می‌کنم که شما درباره عملکرد بسیار خوب آن‌ها تعجب

¹⁵ edit-and-continue

¹⁶ breakpoint

¹⁷ debugger

¹⁸ Program Database

خواهید کرد. شاید باور این برای شما سخت باشد، اما بسیاری (شامل من) (و شامل من، مترجم) فکر می‌کنند که برنامه‌های مدیریت شده بهتر از برنامه‌های مدیریت نشده عمل می‌کنند. دلایل بسیاری بر این باور وجود دارد. برای نمونه، وقتی که کامپایلر JIT کد IL را به دستورات اصلی پردازنده کامپایل می‌کند، پیرامون محیط اجرای برنامه، بیش از کامپایلر مدیریت نشده اطلاعات دارد، برخی راه‌هایی که کد مدیریت شده می‌تواند از کد مدیریت نشده پیشی بگیرد:

یک کامپایلر JIT می‌تواند تعیین کند که برنامه روی پردازنده Intel Pentium 4 اجرا می‌شود و کد اصلی‌ای تولید کند که از دستورات خاص ارائه شده توسط Pentium 4 بهره بگیرد. معمولاً، برنامه‌های مدیریت نشده برای پایین‌ترین پردازنده رایج کامپایل می‌شوند و دستورات خاص که می‌توانند سرعت اجرای برنامه را بالا ببرند نادیده گرفته می‌شوند.

- یک کامپایلر JIT می‌تواند تعیین کند که یک شرط در ماشینی که بر روی آن در حال اجراست؛ همواره غلط است. برای نمونه متدی را تصور کنید که شامل کد زیر است:

```
if (numberOfCPUs > 1) {
...
}
```

- اگر ماشین فقط یک پردازنده داشته باشد این کد می‌تواند باعث شود کامپایلر JIT هیچ دستور پردازنده‌ای را تولید نکند. در این مورد، کد اصلی بهینه شده و کوچکتر خواهد بود و سریعتر اجرا می‌شود.
- CLR می‌توانست اجرای برنامه را تحلیل کرده و در حال اجرای برنامه کد IL را مجدداً به کد اصلی کامپایل کند. این کامپایل مجدد می‌تواند بر طبق الگوهای اجرایی مشاهده شده میزان پیش‌بینی‌های اشتباه انشعابات را کاهش دهد. نسخه‌های کنونی CLR این کار را انجام نمی‌دهند ولی شاید نسخه‌های آتی این قابلیت را داشته باشند.

این‌ها تنها تعدادی از دلایلی بودند که شما باید انتظار داشته باشید کدهای مدیریت شده آینده بهتر از کدهای مدیریت نشده‌ی کنونی اجرا شوند. همانطور که اشاره کردم عملکرد کنونی برای اغلب برنامه‌ها بسیار خوب است و به مرور زمان بهتر نیز خواهد شد.

اگر تجربه‌های شما حاکی از آن است که کامپایلر JIT آن عملکرد مورد انتظار شما را به دست نمی‌دهد، می‌بایست از ابزار NGen.exe که همراه با SDK دات‌نت عرضه می‌شود استفاده کنید. این ابزار تمام کدهای IL یک اسمبلی را به کد اصلی پردازنده کامپایل کرده و کد اصلی نتیجه را در یک فایل ذخیره می‌کند. در زمان اجرا، وقتی یک اسمبلی بارگذاری می‌شود، CLR به صورت خودکار بررسی می‌کند که آیا یک نسخه‌ی قبلاً کامپایل شده از اسمبلی موجود است یا خیر و اگر این نسخه یافت شد CLR کد قبلاً کامپایل شده را بارگذاری می‌کند و دیگر به کامپایل در زمان اجرا نیازی نخواهد بود. توجه کنید که NGen.exe باید پیرامون فرضیاتی که درباره‌ی محیط اجرایی دارد، محافظه کارانه عمل کند و به همین دلیل، کد تولیدی توسط NGen.exe به میزانی که کد تولیدی توسط کامپایلر JIT بهینه است، بهینه نخواهد بود. NGen.exe در ادامه فصل بررسی خواهیم کرد.

IL و بازبینی

IL بر پایه‌ی پشته است، به این معنی که تمام دستورات آن، عملوندها را در پشته اجرایی پوش کرده و نتایج را از پشته پاپ می‌کنند. به این دلیل که IL، هیچ دستوری برای کار با رجیسترها ارائه نمی‌کند، ساخت زبان‌ها و کامپایلرهای جدید که با CLR کار کنند آسان است.

همچنین دستورات IL فارغ از نوع هستند. برای نمونه، IL، دستوری به نام **add** ارائه می‌کند که آخرین ۲ عملوند پوش شده در پشته را با همدیگر جمع می‌کند. هیچ نسخه ۳۲ یا ۶۴ بیتی از دستور **add** وجود ندارد. وقتی دستور **add** اجرا می‌شود، نوع عملوندهای موجود در پشته را تعیین کرده و عمل مناسب را انجام می‌دهد.

به نظر من، مهمترین مزیت IL این نیست که پردازنده را نادیده می‌گیرد. مهمترین مزیتی که IL فراهم می‌کند امنیت و قدرت آن است. وقتی IL به دستورات اصلی پردازنده کامپایل می‌شود، CLR فرآیندی را به نام بازبینی **verification** انجام می‌دهد. فرآیند بازبینی، کد سطح بالای IL را بررسی کرده و مطمئن می‌شود که هر آنچه کد انجام می‌دهد، امن است. برای نمونه؛ بازبینی بررسی می‌کند که هر متد با تعداد پارامتر صحیح فراخوانی شده باشد و هر پارامتر که به متد ارسال شده است از نوع صحیح باشد و مقدار برگشتی هر متد به درستی مورد استفاده فرار گرفته باشد و هر متد دارای عبارت **return** باشد و متادیتای ماژول مدیریت شده، اطلاعات تمام متدها و نوع‌ها را داراست که فرآیند بازبینی از این اطلاعات استفاده می‌کند.

در ویندوز، هر پردازنده فضای آدرس مجازی خود را دارد. فضاهای آدرس می‌بایست مجزا باشند؛ چرا که شما نمی‌توانید به کد یک برنامه اطمینان کنید. این موضوع کاملاً ممکن (و بسیار رایج) است که یک برنامه از فضای آدرس غیرمعتبر خوانده و یا در آن بنویسد. با قراردادن هر پردازنده‌ی ویندوز در یک فضای آدرس مجزا، پایداری و امنیت حاصل می‌شود و یک پردازنده نمی‌تواند بر پردازنده‌ی دیگر اثر نامطلوب بگذارد.

با بازبینی کد مدیریت شده، شما مطمئن می‌شوید که کد به صورت نادرست به حافظه دسترسی ندارد و نمی‌تواند بر کد برنامه‌های دیگر اثر گذارد. این بدین معنیست که شما می‌توانید چندین برنامه مدیریت شده را در یک فضای آدرس مجازی ویندوز اجرا کنید.

چون برنامه‌های ویندوز به منابع زیادی از سیستم عامل نیازمندند، وجود تعداد زیادی از آن‌ها می‌تواند به عملکرد سیستم ضربه زده و منابع در دسترس را محدود کند. کاهش تعداد برنامه‌ها با اجرای چندین برنامه در یک تک برنامه‌ی سیستم عامل می‌تواند عملکرد را بهبود بخشد، منابع کمتری مصرف کند و به همان قدرتی اجرا شود که گویا هر برنامه‌ی مجزا به خود را دارد. این مزیت دیگر کد مدیریت شده نسبت به کد مدیریت نشده است.

CLR در واقع قدرت اجرای چندین برنامه مدیریت شده در یک تک برنامه سیستم عامل را دارد. هر برنامه مدیریت شده در یک AppDomain اجرا می‌شود. به صورت پیشفرض، هر فایل EXE مدیریت شده در فضای آدرس مجازی خود که تنها یک AppDomain دارد اجرا می‌شود. هر چند، برنامه‌های که CLR را میزبانی می‌کند (مثل Internet Information Services یا Microsoft SQL Server) می‌تواند چندین AppDomain را در یک تک برنامه سیستم عامل اجرا کند. بخشی از فصل ۲۲ "میزبانی CLR و AppDomain ها" را به بحث AppDomain اختصاص خواهیم داد.

کد ناامن

به صورت پیشفرض، کامپایلر سی‌شارپ میکروسافت کد امن^{۱۹} تولید می‌کند. کد امن کدی است که قابل تایید امن بودن باشد. هر چند، کامپایلر سی‌شارپ میکروسافت اجازه می‌دهد که کد ناامن بنویسید. کد ناامن می‌تواند به صورت مستقیم با آدرس‌های حافظه کار کرده و بایت‌های موجود در این آدرس‌ها را دستکاری کند. این یک ویژگی بسیار قدرتمند است و معمولاً هنگام تقابل با کد مدیریت نشده و یا وقتی می‌خواهید عملکرد الگوریتم بحرانی خود را بهبود بخشید، مفید است. اما استفاده از کد ناامن ریسک مهمی را در پی دارد. کد ناامن می‌تواند ساختمان داده‌ها را تخریب کرده و منجر به آسیب پذیری امنیتی شود. به همین علت، کامپایلر سی‌شارپ اجبار می‌کند که تمام متدهایی که کد ناامن دارند با کلمه کلیدی **unsafe** علامت زده شوند. به علاوه کامپایلر سی‌شارپ از شما می‌خواهد که سورس کد را با سوئیچ **/unsafe** کامپایل کنید.

وقتی که کامپایلر JIT می‌خواهد یک متد ناامن را کامپایل کند، بررسی می‌کند که آیا اسمبلی حاوی این متد اجازه‌ی **System.Security.Permissions.SecurityPermission** را دارد و پرچم **SkipVerification** از **System.Security.Permissions.SecurityPermissionFlag** فعال باشد. اگر این پرچم فعال باشد، کامپایلر JIT کد ناامن را کامپایل کرده و اجازه می‌دهد که این کد اجرا شود. CLR به کد اطمینان می‌کند و امیدوار است که دسترسی مستقیم به حافظه و دستکاری بایت‌ها صدمه‌ای وارد نکند. اگر این پرچم فعال نباشد، کامپایلر JIT یا یک **System.Security.VerificationException** و یا یک **System.InvalidProgramException** را تولید کرده و از اجرای متد جلوگیری می‌کند. در واقع، کل برنامه احتمالاً در این نقطه متوقف می‌شود، اما حداقل صدمه‌ای وارد نشده است.

نکته به صورت پیشفرض، اسمبلی‌هایی که از ماشین محلی یا از اشتراک‌های شبکه بارگذاری می‌شوند به صورت کامل مورد اطمینان هستند. به این معنی که هر کاری می‌توانند انجام دهند که می‌تواند شامل اجرای کد ناامن باشد. هر چند به صورت پیشفرض اسمبلی‌هایی که از اینترنت اجرا می‌شوند اجازه‌ی اجرای کد ناامن را ندارند. اگر آن‌ها کد ناامن داشته باشند، یکی از اکسپشن‌های مذکور تولید می‌شود. یک مدیر یا کاربر نهایی می‌تواند این تنظیمات را تغییر دهد اما به هر حال مدیر مسئولیت کامل رفتار کد را بر عهده خواهد داشت.

میکروسافت یک ابزار به نام **PEVerify.exe** عرضه می‌کند که تمام متدهای یک اسمبلی را بررسی کرده و شما را از وجود هر متدی که کد ناامن دارد، آگاه می‌سازد. شما می‌توانید **PEVerify.exe** را برای اسمبلی‌هایی که به آن‌ها ارجاع داده اید اجرا کنید؛ این به شما می‌گوید که در اجرای برنامه یتان از طریق اینترنت یا اینترنت دچار مشکل خواهید شد یا خیر. باید از این نکته آگاه باشید که فرآیند بازبینی نیازمند دسترسی به متادیتای درون اسمبلی‌های ارجاعی است. پس زمانی که از **PEVerify.exe** برای بررسی یک اسمبلی استفاده می‌کنید، این ابزار باید بتواند به تمام اسمبلی‌های ارجاعی دسترسی داشته و آن‌ها را بارگذاری کند. چون **PEVerify.exe** از CLR برای یافتن موقعیت اسمبلی‌های ارجاعی استفاده می‌کند اسمبلی‌ها باید در همان موقعیتی باشند که در زمان اجرای اسمبلی در آن‌ها قرار دارند. این اتصال اسمبلی‌ها و قوانین آن‌ها را در فصل ۲ و ۳ "اسمبلی‌های اشتراکی و اسمبلی‌های قوی‌نام" بحث خواهیم کرد.

¹⁹ Safe Code

IL و محافظت از مالکیت معنوی شما

بعضی افراد نگرانند که کد IL، محافظت کافی از مالکیت معنوی برای الگوریتم‌های آن‌ها فراهم نمی‌کند. به بیان دیگر، آن‌ها معتقدند که شما می‌توانید یک ماژول مدیریت‌شده بسازید و فرد دیگری به کمک ابزاری مثل IL Disassembler براحتی کار کد شما را مهندسی معکوس کند.

بله، این درست است که IL کدی سطح بالاتر از دیگر زبان‌های اسمبلی است و در کل مهندسی معکوس کد IL نسبتاً آسان است. اما در پیاده‌سازی کدهای سمت سرور (مثل سرویس‌های وب، فرم وب یا رویه‌های ذخیره شده) اسمبلی شما در سرور قرار دارد. چون هیچ کس خارج از شرکت شما نمی‌تواند به اسمبلی دسترسی داشته باشد پس هیچ کس خارج از شرکت شما نمی‌تواند از ابزاری برای دیدن کد IL استفاده کند و مالکیت معنوی شما کاملاً تامین می‌شود.

اگر نگران هر یک از اسمبلی‌هایی که توزیع می‌کنید، هستید، می‌توانید از یک ابزار مبهم‌کننده (Obfuscator) استفاده کنید. این ابزارها نام تمام متغیرهای خصوصی در متادیتای اسمبلی شما را درهم می‌ریزند. این کار، عملیات بازگردانی متغیرها به نام اصلی و کشف هدف هر متد را برای اشخاص دیگر مشکل می‌کند. توجه کنید که این ابزارها تا حدی امنیت را فراهم می‌کنند چرا که IL باید بالاخره توسط کامپایلر JIT کامپایل شود.

اگر فکر می‌کنید این ابزارها امنیت مورد نظر شما را تامین نمی‌کنند می‌توانید الگوریتم‌های حساس‌تر خود را در چند ماژول مدیریت نشده که به جای کد IL و متادیتا، دستورات اصلی پردازنده را دارند، بنویسید. سپس از ویژگی CLR برای کار با کد مدیریت نشده به فرض داشتن اجازه‌ی کافی؛ برای ارتباط بین بخش‌های مدیریت‌شده و نشده‌ی برنامه‌ی خود استفاده کنید. البته، اگر نگران آن نباشید که برخی افراد، دستورات اصلی پردازنده را نیز مهندسی معکوس کنند.

ابزار تولید کد اصلی: Native Code Generator (NGen.exe)

ابزار NGen.exe که همراه با SDK دات‌نت عرضه می‌شود می‌تواند برای کامپایل کد IL به کد اصلی زمانی که برنامه در ماشین کاربر نصب می‌شود، مورد استفاده قرار گیرد. به این دلیل که کد در زمان نصب کامپایل می‌شود؛ کامپایلر JIT نیاز ندارد که کد IL را در زمان اجرا کامپایل کند و این می‌تواند موجب افزایش سرعت برنامه شود. ابزار NGen.exe در دو مورد کاربرد دارد:

- **بهبود زمان راه اندازی برنامه** اجرا کردن NGen.exe می‌تواند زمان راه اندازی برنامه را بهبود بخشد چرا که کد از قبل به کد اصلی کامپایل شده است، پس عملیات کامپایل در زمان اجرا صورت نمی‌گیرد.
- **کاهش حجم کاری برنامه** اگر فکر می‌کنید که یک اسمبلی به طور همزمان در چندین پردازنده بارگذاری می‌شود، اجرای NGen.exe بر روی اسمبلی می‌تواند حجم کاری برنامه‌ها را کاهش دهد. علت این است که NGen.exe کد IL را به کد اصلی کامپایل و خروجی را در یک فایل ذخیره می‌کند. این فایل می‌تواند درون فضای آدرس چندین پردازنده‌ای، نگاشت حافظه‌ای شده و کد آن به اشتراک گذاشته شود و دیگر هر پردازنده‌ای نیاز ندارد که یک کپی شخصی از کد آن داشته باشد.

وقتی که یک برنامه‌ی نصب، NGen.exe را بر روی یک برنامه یا یک تک اسمبلی اجرا می‌کند، کد IL تمام اسمبلی‌های آن برنامه یا تک اسمبلی مشخص شده به کد اصلی کامپایل می‌شود. یک فایل اسمبلی جدید که فقط شامل کد اصلی به جای کد IL است توسط NGen.exe ایجاد می‌شود. این فایل در پوشه‌ای در دایرکتوری C:\Windows\Assembly\NativeImages_v4.0.#####_64 قرار دارد. نام دایرکتوری شامل نسخه CLR بوده و دو رقم آخر تعیین می‌کند که کد اصلی برای x86 (نسخه ۳۲ بیتی ویندوز)، x64 یا Itanium (نسخه ۶۴ بیتی ویندوز) کامپایل شده است. این‌بار وقتی CLR یک اسمبلی را بارگذاری می‌کند، CLR نگاه می‌کند آیا فایل NGen شده متناظر وجود دارد یا خیر. اگر یک فایل اصلی یافت نشد، کامپایلر JIT طبق معمول کد IL را کامپایل می‌کند. اما اگر فایل اصلی متناظر پیدا شد، CLR از کد کامپایل شده‌ی درون فایل اصلی استفاده می‌کند و نیاز نیست متدهای فایل در زمان اجرا کامپایل شوند.

در نگاه اول، این عملیات بسیار عالی به نظر می‌رسد؛ شما تمام مزایای کد مدیریت‌شده (جمع آوری زباله، بازیابی کد، امنیت نوع و ...) را دارید بدون آنکه با مشکلات کد مدیریت‌شده (کامپایلر JIT) مواجه شوید. اما واقعیت آنگونه که به نظر می‌رسد نیست. چندین مشکل احتمالی در رابطه با فایل‌های NGen شده وجود دارد:

- **عدم محافظت از مالکیت معنوی** اکثر افراد فکر می‌کنند که می‌توانند فایل‌های NGen شده را بدون آنکه فایل شامل کد IL همراه آن باشد را به مشتریان خود داده و در پی آن مالکیت معنوی خود را حفظ کنند. متأسفانه این غیر ممکن است. در زمان اجراء CLR (برای کاربردهایی مثل سریالی کردن و یا رفلکشن) باید به متادیتای اسمبلی دسترسی داشته باشد. این امر نیازمند آن است که اسمبلی‌های حاوی کد IL و متادیتا نیز وجود داشته باشند. به علاوه اگر CLR به دلایلی نتواند از فایل NGen شده استفاده کند (در ادامه توضیح داده می‌شود)، به کامپایلر JIT برای کامپایل کد IL اسمبلی مراجعه می‌کند، پس کد IL باید در دسترس باشد.
- **فایل‌های NGen شده ممکن است به روز نباشند** وقتی که CLR یک فایل NGen شده را بارگذاری می‌کند، برخی المان‌ها را درباره‌ی ی کد از قبل کامپایل شده با محیط اجرایی کنونی مقایسه می‌کند. اگر هر یک از این المان‌ها هماهنگ نباشند، فایل NGen شده غیر قابل استفاده بوده و فرآیند عادی کامپایلر JIT استفاده می‌شود. برخی از موارد مورد مقایسه عبارتند از:
 - نسخه‌ی CLR: که با پچ‌ها و سرویس پک‌ها تغییر می‌کند.
 - نوع پردازنده: در صورت ارتقای سخت افزاری تغییر می‌کند.
 - نسخه‌ی سیستم عامل ویندوز: با اعمال سرویس پک جدید تغییر می‌کند.
 - شناسه نسخه مازول از هویت اسمبلی²⁰ (MVID): با کامپایل مجدد تغییر می‌کند.
 - شناسه‌های نسخه از اسمبلی‌های ارجاعی: با کامپایل مجدد یک اسمبلی ارجاعی، تغییر می‌کند.
 - امنیت: با برداشتن دسترسی‌هایی (مثل اجازه‌های وراثت اعلانی، زمان-پیوند اعلانی، SkipVerification یا UnmanagedCode) که قبلاً وجود داشتند، تغییر می‌کند.

توجه کنید ممکن است NGen.exe در حالت بروزرسانی اجرا شود. این عمل می‌گوید که NGen.exe بر روی تمام اسمبلی‌هایی که از قبل NGen شده اند، اجرا شود. هر گاه یک کاربر نهایی سرویس پک جدیدی از دات‌نت نصب کند، برنامه‌ی نصب سرویس پک به صورت خودکار ابزار NGen.exe را در حالت بروزرسانی اجرا می‌کند که باعث می‌شود فایل‌های NGen شده با نسخه‌ی CLR نصب شده هماهنگ باقی بمانند.

- **کارایی پایین در زمان اجرا** در هنگام کامپایل کد، NGen نمی‌تواند به اندازه‌ی کامپایلر JIT درباره‌ی محیط اجرایی فرضیاتی را در نظر بگیرد. این باعث می‌شود که NGen.exe کد ضعیفی تولید کند. برای نمونه NGen برای دستورات خاص پردازنده بهینه سازی نکرده و برای دسترسی به فیلدهای استاتیک آدرس غیر مستقیم را اضافه می‌کند چرا که آدرس واقعی فیلدهای استاتیک تا زمان اجرا مشخص نیستند. NGen برای فراخوانی سازنده‌های کلاس در هر جا کد آن را قرار می‌دهد چرا که ترتیب اجرای کد را نمی‌داند و اینکه سازنده‌ی کلاس قبلاً اجرا شده است یا نه. (برای اطلاعات پیرامون سازنده کلاس به فصل ۸ "متدها" مراجعه کنید). برخی برنامه‌های NGen شده تا ۵ درصد از همتای کامپایل شده توسط JIT کندتر عمل می‌کنند. پس اگر می‌خواهید برای افزایش کارایی برنامه خود از NGen.exe استفاده کنید باید نسخه‌های NGen شده و NGen نشده را با هم مقایسه کرده تا مطمئن شوید نسخه NGen شده ضعیف تر عمل نکند. برای برخی برنامه‌ها کاهش حجم کاری موجب افزایش کارایی برنامه می‌شود، پس استفاده از NGen می‌تواند مفید باشد.

به خاطر تمام مسائل مطرح شده، شما باید در هنگام استفاده از NGen.exe خیلی مراقب باشید. برای برنامه‌های سمت-سرور، NGen.exe اثر بسیار اندکی دارد چرا که تنها درخواست اولین کلاینت با سرعت کمتری اجرا می‌شود و در خواست‌های آتی با سرعت بالا اجرا می‌شوند. به علاوه در اکثر برنامه‌های سروری تنها یک نمونه از کد نیاز است پس کاهش حجم کاری حاصل نمی‌شود. همچنین توجه کنید که فایل‌های NGen شده در بین AppDomain ها نمی‌توانند به اشتراک گذاشته شوند. پس فایده‌ای برای NGen کردن یک فایل اسمبلی که در سناریوهای بین AppDomain (مثل ASP.NET) استفاده می‌شوند وجود ندارد.

برای برنامه‌های کلاینت، اگر یک اسمبلی به صورت همزمان توسط چندین برنامه استفاده می‌شود؛ جهت کاهش زمان راه‌اندازی و حجم کاری؛ NGen کردن اسمبلی مفید است. به علاوه اگر NGen.exe برای تمام اسمبلی‌های یک برنامه استفاده شود، CLR اصلاً نیازی به بازگذاری کامپایلر JIT نداشته و

²⁰ Assembly's identity Module Version ID

حجم کاری بازم کاهش می‌یابد. البته، تنها اگر یک اسمبلی NGen نشده باشد و یا فایل NGen شده غیر قابل استفاده باشد، کامپایلر JIT بارگذاری شده و حجم کاری برنامه افزایش می‌یابد.

کتابخانه کلاس فریمورک

دات‌نت فریمورک شامل کتابخانه کلاس فریمورک (FCL) Framework Class Library است. FCL مجموعه‌ای از اسمبلی‌های DLL؛ شامل چندین هزار تعریف نوع می‌باشد. میکروسافت کتابخانه‌های اضافی شامل ^{۲۱} Windows SideShow Managed API SDK و DirectX SDK نیز تولید کرده است. این کتابخانه‌های اضافی نوع‌های بیشتر و به دنبال آن کاربردهای بیشتری را برای استفاده شما فراهم می‌کنند. در واقع، میکروسافت کتابخانه‌های بسیاری تولید کرده است که کار با تکنولوژی‌های میکروسافت را برای برنامه‌نویسان آسان کرده‌اند. برخی از برنامه‌هایی که برنامه‌نویسان با استفاده از این اسمبلی‌ها می‌توانند بسازند:

- **سرویس‌های وب** متدها می‌توانند پیام‌های ارسال شده در اینترنت را به آسانی با استفاده از تکنولوژی‌های ASP.NET XML Web Service از میکروسافت و یا تکنولوژی (WCF) Windows Communication Foundation از میکروسافت؛ پردازش کنند.
- **برنامه‌های فرم‌های وب مبتنی بر HTML (وب سایت‌ها)** برنامه‌های ASP.NET Web Forms، می‌توانند پرس‌وجوی پایگاه داده را انجام داده و سرویس‌های وب را فراخوانی کنند، اطلاعات برگشتی را ترکیب یا فیلتر کرده و سپس آن اطلاعات را در مرورگر به کمک رابط کاربری غنی به کاربر نشان دهند.
- **برنامه‌های غنی GUI ویندوز** به جای استفاده از صفحات و فرم‌های وب برای طراحی رابط کاربری برنامه خود، می‌توانید از کاربرد قدرتمند تر و با کارایی بالای ویندوز از طریق تکنولوژی Windows Forms یا Windows Presentation Foundation (WPF) از میکروسافت، استفاده کنید. برنامه‌های GUI از کنترل‌ها، منوها، رویدادهای موس و کیبورد استفاده کرده و می‌توانند به طور مستقیم با سیستم عامل تبادل اطلاعات کنند. برنامه‌های Windows Forms، همچنین می‌توانند پرس‌وجو انجام داده و با سرویس‌های وب کار کنند.
- **برنامه‌های غنی اینترنتی^{۲۲} (RIA)** به کمک تکنولوژی Silverlight میکروسافت، می‌توانید برنامه‌هایی با رابط کاربری غنی طراحی کنید که از طریق اینترنت اجرا شوند. این برنامه‌ها می‌توانند درون یا بیرون یک مرورگر اجرا شوند. همچنین آن‌ها در سیستم عامل‌های غیر ویندوز و دستگاه‌های موبایل نیز کار می‌کنند.
- **برنامه‌های کنسولی ویندوز** برای برنامه‌هایی با رابط کاربری بسیار ساده، یک برنامه‌ی کنسولی روشی سریع و آسان برای ایجاد یک برنامه فراهم می‌کند. کامپایلرها، ابزارها و برنامه‌های کاربردی اغلب به صورت برنامه‌های کنسولی طراحی می‌شوند.
- **سرویس‌های ویندوز** بله، به کمک دات‌نت فریمورک می‌توانید برنامه‌های سرویسی بسازید که از طریق Windows Service Control Manager (SCM) قابل کنترل باشند.
- **رویه‌های ذخیره شده پایگاه داده** Microsoft SQL Server، IBM DB2 و پایگاه داده Oracle اجازه می‌دهند که برنامه‌نویسان رویه‌های ذخیره شده خود را به کمک دات‌نت فریمورک بنویسند.
- **کامپوننت** دات‌نت فریمورک به شما اجازه می‌دهد که اسمبلی‌های مستقل (کامپوننت) شامل نوع‌هایی که در انواع برنامه‌های ذکر شده قابل استفاده است بسازید.

چون FCL شامل هزاران نوع است، مجموعه‌ای از نوع‌های مرتبط در قالب یک فضای نام namespace ارائه می‌شوند. برای نمونه فضای نام System (که شما باید با آن آشنا شوید) شامل نوع پایه Object است که تمام نوع‌های دیگر، نهایتاً از آن مشتق می‌شوند. به علاوه، فضای نام System شامل نوع‌هایی برای اعداد صحیح، کاراکترها، رشته‌ها، مدیریت اکسپشن و ورودی خروجی‌های کنسول به همراه نوع‌های کاربردی برای تبدیل امن بین نوع‌های داده ای، فرمت کردن نوع‌های داده‌ای، تولید اعداد تصادفی، و انجام عملیات‌های ریاضی مختلف است. همه‌ی برنامه‌ها از نوع‌های موجود در فضای نام System استفاده می‌کنند.

برای دسترسی به ویژگی‌های فریمورک، باید بدانید کدام فضای نام شامل نوع‌هایی است که امکانات مورد نظر شما را فراهم می‌کند. نوع‌های زیادی اجازه می‌دهند که رفتارشان را سفارشی کنید، شما این کار را با مشتق کردن نوع خود از نوع مورد نظر در FCL انجام می‌دهید. ماهیت شی‌گرای پلتفرم به گونه‌ای

^{۲۱} تصادفاً؛ من شخصاً برای طراحی این SDK با میکروسافت قرارداد بسته بودم.

^{۲۲} Rich Internet Applications

است که دات نت مدل برنامه نویسی استواری برای برنامه نویسان فراهم می کند. همچنین، برنامه نویسان می توانند به راحتی فضای نام هایی از خود که شامل نوع های خودشان است بسازند. این فضای نام ها و نوع ها به راحتی با الگوی برنامه نویسی ادغام می شوند. در مقایسه با الگوهای برنامه نویسی Win32، این دیدگاه جدید، توسعه نرم افزاری را بسیار ساده می کند.

اغلب فضای نام های FCL، نوع هایی را ارائه می کنند که با هر گونه برنامه قابل استفاده است. جدول ۳-۱ برخی از فضای نام های عمومی تر را لیست کرده و توضیح خلاصه ای پیرامون کاری که نوع های موجود در آن فضای نام انجام می دهند ارائه کرده است. این نمونه ی کوچکی از فضای نام های موجود است. برای آشنایی با مجموعه ی فضای نام هایی که مایکروسافت ارائه می کند به مستندات موجود در SDK های مایکروسافت مراجعه کنید.

جدول ۳-۱ برخی از فضای نام های کلی در FCL

توضیح محتویات	Namespace
تمام نوع های پایه که توسط هر برنامه ای استفاده می شود	System
نوع هایی برای برقراری ارتباط با پایگاه داده و پردازش داده	System.Data
نوع هایی برای ورودی خروجی و کار با دایرکتوری ها و فایل ها	System.IO
نوع هایی برای ارتباطات سطح پایین شبکه و کار با پروتکل های اینترنتی	System.Net
نوع هایی که اجازه می دهند کد مدیریت شده به امکانات کد مدیریت نشده ی سیستم عامل مثل کامپوننت های COM و توابع Win32 یا DLL های سفارشی، دسترسی داشته باشد	System.Runtime.InteropServices
نوع هایی برای محافظت از داده ها و منابع	System.Security
نوع هایی برای کار با متن در اینکدینگ های مختلف مثل ASCII و Unicode	System.Text
نوع هایی برای عملیات غیر همزمان و همزمان کردن دسترسی به منابع	System.Threading
نوع هایی برای پردازش طرح ها و داده های Extensible Markup Language (XML)	System.Xml

این کتاب در مورد CLR و پیرامون نوع های کلی که با CLR ارتباط نزدیک دارند، می باشد. بنابراین، محتویات این کتاب برای همه ی برنامه نویسانی است که برنامه یا کامپوننت هایی می نویسند که با CLR کار می کند. کتاب های خوب دیگری هستند که نوع های خاصی از برنامه مثل Web Services، Web Forms، Windows Forms و ... را پوشش می دهند. این کتاب ها در شروع برای نوشتن برنامه به شما کمک می کنند. من به این کتاب ها که مختص یک نوع برنامه هستند به عنوان کتاب هایی که از بالا به پایین در یادگیری به شما کمک می کنند، نگاه می کنم چرا که بر روی یک نوع برنامه تمرکز می کنند و نه بر پلتفرم ایجاد و توسعه برنامه. در این کتاب، اطلاعاتی را ارائه می کنم که به شما در یادگیری از پایین به بالا کمک می کند. بعد از مطالعه این کتاب و یک کتاب مختص به یک نوع برنامه، شما قادر خواهید بود به راحتی و با مهارت هر نوع برنامه ای که دوست دارید بسازید.

سیستم مشترک نوع

اکنون برای شما واضح است که CLR تماما پیرامون نوع هاست. نوع ها، کاربردها را در اختیار برنامه شما و دیگر نوع ها قرار می دهند. نوع ها، مکانیزی هستند که کد نوشته شده در یک زبان می تواند با کد نوشته شده در زبان دیگری صحبت کند. چون نوع ها در ریشه ی CLR هستند، مایکروسافت یک مشخصات رسمی به نام سیستم مشترک نوع (CTS) Common Type System ایجاد کرده است که توضیح می دهد چگونه نوع ها تعریف می شوند و چگونه رفتار می کنند.

نکته در واقع، میکروسافت، CTS و دیگر بخش‌های دات‌نت شامل فرمت فایل‌ها، متادیتا، IL و دسترسی به پلتفرم زیرین (P/Invoke) را به ECMA برای استاندارد سازی، ارسال کرده است. این استاندارد، زیر ساخت مشترک زبان Common Language Infrastructure (CLI) نامیده شده و مشخصات ECMA-335 می باشد. به علاوه، میکروسافت بخش‌هایی از FCL، زبان برنامه‌نویسی سی‌شارپ (ECMA-334) و زبان برنامه‌نویسی C++/CLI را نیز ارسال کرده است. برای اطلاعات پیرامون استانداردهای صنعتی به وب سایت ECMA مربوط به Technical Committee 39 : www.ecma-international.org/ مراجعه کنید. همچنین می توانید به وب سایت میکروسافت Community Promise خود را نیز بر ECMA-334 و ECMA-335 اعمال کرده است. برای اطلاعات بیشتر مراجعه کنید به <http://msdn.microsoft.com/en-us/netframework/aa569283.aspx> . <http://www.microsoft.com/interop/cp/default.aspx>

مشخصات CTS بیان می‌کند که یک نوع می‌تواند صفر یا بیشتر عضو داشته باشد. در بخش ۲، "طراحی نوع‌ها" تمامی این اعضا را با جزئیات پوشش می‌دهد. برای الان توضیح خلاصه‌ای از آن‌ها برایتان می‌دهم:

- **فیلد (Field)** یک متغیر داده‌ای که بخشی از حالت شی است. فیلدها با نام و نوعشان شناسایی می‌شوند.
 - **متد (Method)** یک تابع که عملیاتی بر روی شی انجام داده و اغلب حالت شی را تغییر می‌دهد. متدها دارای یک نام، یک امضاء^{۲۳} و تغییر دهنده^{۲۴}ها هستند. امضاء تعداد پارامترها (و ترتیب آن‌ها)، نوع پارامترها، اینکه متد مقداری را برمی‌گرداند و در صورت برگرداندن، نوع مقدار برگشتی توسط متد را تعیین می‌کند.
 - **ویژگی (Property)** برای فراخوانی کننده، این عضو شبیه به یک فیلد است. اما برای پیاده‌سازی کننده‌ی نوع، شبیه به یک (یا دو) متد است. ویژگی‌ها اجازه می‌دهند که پیاده‌سازی کننده، پیش از آنکه دسترسی به مقدار و/یا محاسبه‌ی یک مقدار در صورت نیاز صورت بگیرد، پارامترهای ورودی و حالت شی را ارزیابی کند. آن‌ها نحو ساده‌ای برای کاربر یک نوع فراهم می‌کنند. همچنین به شما اجازه می‌دهند که فیلدهای فقط-خواندنی یا فقط-نوشتنی ایجاد کنید.
 - **رویداد (Event)** یک رویداد، یک مکانیزم خبررسانی بین یک شی و دیگر اشیاء علاقه مند، فراهم می‌کند. برای نمونه، یک دکمه می‌تواند رویدادی ارائه کند که دیگر اشیاء را هنگام کلیک شدن خود، باخبر کند.
- همچنین CTS قوانینی برای پدیداری نوع و دسترسی به اعضای یک نوع فراهم می‌کند. برای نمونه، علامت زدن یک نوع با **public** (یا **public** نامیده می‌شود) باعث می‌شود نوع، قابل رویت شده و برای هر اسمبلی در دسترس باشد. در سوی دیگر علامت زدن یک نوع با **assembly** (در سی‌شارپ **internal** نامیده می‌شود) نوع را تنها در دسترس و رویت کد درون همان اسمبلی قرار می‌دهد. بنابراین CTS قوانینی را تعیین می‌کند که اسمبلی‌ها برای یک نوع، محدوده‌ی پدیداری (قابل رویت شدن) تشکیل می‌دهند و CLR این قوانین پدیداری را اجبار می‌کند.
- یک نوع که در معرض دید یک فراخوانی کننده است، می‌تواند دسترسی به اعضای نوع را برای فراخوانی کننده محدود کند. لیست زیر برجسب‌های معتبر برای کنترل دسترسی به اعضا را نشان می‌دهد.
- **Private** عضو تنها توسط دیگر اعضای همان نوع قابل دسترسی است.
 - **Family** عضو توسط نوع‌های مشتق شده بدون توجه به آنکه در همان اسمبلی باشند، قابل دسترسی است. توجه کنید که بسیاری از زبان‌ها (مثل C++ و C#) **family** را با **protected** نشان می‌دهند.
 - **Family and assembly** عضو توسط نوع‌های مشتق شده که تنها درون همان اسمبلی تعریف شده اند، قابل دسترسی است. بسیاری از زبان‌ها (مثل C++، C# و Visual Basic) این کنترل دسترسی را ارائه نمی‌کنند. البته، زبان اسمبلی IL، این کنترل را قابل دسترسی می‌سازد.
 - **Assembly** عضو توسط هر کدی درون همان اسمبلی قبل دسترسی است. بسیاری از زبان‌ها **assembly** را با **internal** نشان می‌دهند.
 - **Family or assembly** عضو توسط نوع‌های مشتق شده در هر اسمبلی قابل دسترسی است. همچنین عضو، توسط هر نوعی در همان اسمبلی قابل دسترسی است. C#، **family or assembly** را با **protected internal** نشان می‌دهد.

²³ Signature

²⁴ Modifier

▪ **Puplic** عضو توسط هر کدی در هر اسمبلی قابل دسترسی است.

به علاوه، CTS قوانینی را تعریف می کند که بر وراثت، متدهای مجازی، دوره حیات شی و... نظارت دارد. این قوانین برای هماهنگی با مفاهیم قابل بیان در زبان‌های برنامه‌نویسی مدرن، طراحی شده اند. در واقع بخودی خود شما نیاز ندارید که قوانین CTS را فرابگیرید چرا که هر زبان نحو و قوانین نوع مربوط به خود را همانگونه که با آن آشنا هستید، عرضه می کند. و هنگام تولید اسمبلی، نحو مختص به زبان به زبان CLR که IL است، نگاشت می شود.

زمانیکه برای اولین بار کار با CLR را شروع کردم، به سرعت فهمیدم که بهتر است زبان و رفتار کد را به عنوان دو چیز مجزا و متمایز در نظر گرفت. با ++C می توانید نوع‌های خود را با اعضایشان تعریف کنید. البته که می توانید از سی شارپ یا ویژوال بیسیک برای تعریف همان نوع با همان اعضا استفاده کنید. مطمئناً، نحوی که برای تعریف نوع خود به کار می گیرید وابسته به زبان مورد انتخاب شماست؛ اما رفتار کد بدون توجه به زبان، یکسان خواهد بود چرا که CTS از CLR، رفتار نوع را تعریف می کند.

برای روشن شدن این مطلب، مثالی می زنم. CTS اجازه می دهد که یک نوع فقط از یک نوع پایه مشتق شود. پس، اگرچه زبان ++C نوع‌هایی را پشتیبانی می کند که از چندین نوع پایه مشتق شده اند، اما CTS این نوع را نپذیرفته و نمی تواند با آن کار کند. برای کمک به برنامه‌نویس، کامپایلر ++C/CLI میکروسافت در صورتی که تلاش کنید کدی مدیریت شده که شامل نوعی مشتق شده از چندین نوع پایه باشد را کامپایل کنید، خطایی را گزارش خواهد کرد. یک قانون دیگر CTS را بررسی می کنیم. تمام نوع‌ها باید (نهایتاً) از نوع از پیش تعریف شده به نام **System.Object** ارث برند. همانطور که می دانید، **Object** نام یک نوع تعریف شده در فضای نام **System** است. این **Object** ریشه‌ی تمام دیگر نوع‌ها می باشد و بنابراین تضمین می کند که هر نمونه^{۲۵} ای از هر نوعی دارای یک مجموعه رفتار حداقلی است. به ویژه، نوع **System.Object** اجازه‌ی انجام کارهای زیر را می دهد:

- مقایسه‌ی دو نمونه برای بررسی تساوی
- بدست آوردن کد هش برای نمونه
- پرس و جوی نوع واقعی از یک نمونه
- انجام کپی برداری (ییتی) از نمونه
- بدست آوردن یک نمایش رشته‌ای (متنی) از وضعیت کنونی شی نمونه.

مشخصات مشترک زبان

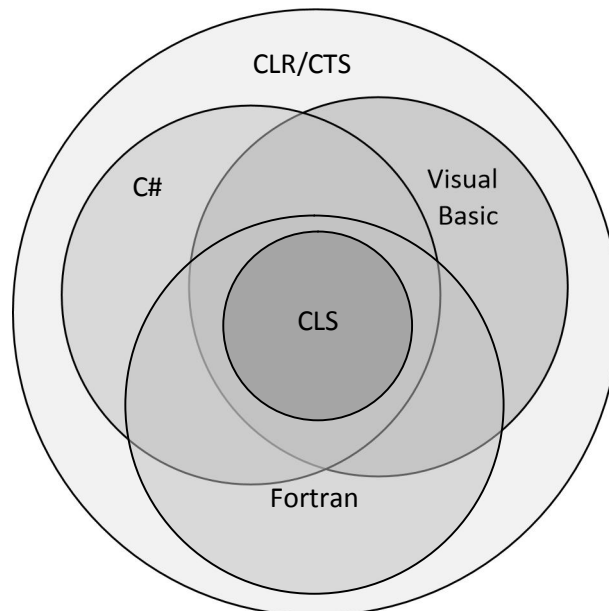
COM اجازه می دهد اشیاء ساخته شده در زبان‌های متفاوت با یکدیگر ارتباط برقرار کنند. اکنون CLR همه‌ی زبان‌ها را جمع کرده و اجازه می دهد با اشیاء ساخته شده در یک زبان همانند شهروندان یکسان از کد نوشته شده در زبانی کاملاً متفاوت، رفتار شود. این یکپارچگی به خاطر مجموعه‌های استاندارد از نوع‌ها، متادیتا (اطلاعات خود توصیف گر نوع) و محیط اجرایی مشترک CLR، ممکن است.

اگرچه این یکپارچگی زبان‌ها بسیار عالی است، اما حقیقت اینست که زبان‌های برنامه‌نویسی تفاوت بسیاری با یکدیگر دارند. برای نمونه، بعضی زبان‌ها به کوچک و بزرگ بودن حروف نمادها حساس نیستند، برخی اعداد بدون علامت، سربارگذاری عملگرها و یا متدهایی با تعداد متغیر از پارامترها را پشتیبانی نمی کنند.

اگر قصد دارید نوع‌هایی بسازید که راحتی از زبان‌های دیگر دسترس پذیر باشند، نیاز دارید تنها از ویژگی‌هایی از زبان برنامه‌نویسی خود استفاده کنید که توسط همه‌ی دیگر زبان‌ها پشتیبانی می شوند. برای کمک به شما، میکروسافت یک مشخصات مشترک زبان **Common Language Specification (CLS)** تعریف کرده است. این مشخصات، حداقل ویژگی‌هایی برای کامپایلرها بیان می کند که سازندگان کامپایلر باید آن‌ها را اعمال کنند اگر می خواهند نوع‌های تولیدی توسط کامپایلرهایشان با دیگر کامپوننت‌های تولیدی توسط زبان‌های منطبق بر CLS سازگار باشد.

CLR/CTS ویژگی‌های بسیار بیشتری از آنچه CLS تعریف کرده را پشتیبانی می کند، بنابراین اگر به قابلیت حمل بین زبانی اهمیت نمی دهید، می توانید نوع‌های بسیار غنی محدود به ویژگی‌های زبان خود تعریف کنید. به خصوص، CLS قوانینی برای نوع‌های در معرض دید خارجی و متدها تعیین کرده که برای دسترس پذیر بودن توسط دیگر زبان‌های منطبق بر CLS باید رعایت شوند. توجه کنید که قوانین CLS به کدی که فقط از درون اسمبلی تعریف کننده قابل دسترسی است اعمال نمی شود. شکل ۶-۱ مفاهیم این پاراگراف را خلاصه می کند.

²⁵ Instance



شکل ۶-۱ زبان‌ها زیر مجموعه‌ای از CLR/CTS و فرامجموعه‌ای از CLS را ارائه می‌کنند (اما نه لزوماً فرامجموعه‌های یکسان). همانطور که شکل ۶-۱ نشان می‌دهد، CLR/CTS، مجموعه‌ای از ویژگی‌ها را ارائه می‌دهد. برخی از زبان‌ها، زیر مجموعه‌ی بزرگی از CLR/CTS را پشتیبانی می‌کنند. برنامه‌نویسی که می‌خواهد در زبان اسمبلی IL برنامه بنویسد، می‌تواند از تمام ویژگی‌های CLR/CTS استفاده کند. اغلب دیگر زبان‌ها، مثل سی-شارپ، ویژوال بیسیک و فرترن زیر مجموعه‌ای از ویژگی‌های CLR/CTS را به برنامه‌نویس ارائه می‌دهند. CLS حداقل ویژگی‌هایی که همه‌ی زبان‌ها باید پشتیبانی کنند را تعریف می‌کند.

اگر نوعی را در یک زبان طراحی کنید که می‌خواهید در زبان دیگری استفاده شود، نباید از ویژگی‌های خارج از CLS، در بحث اعضای عمومی و محافظت شده استفاده کنید. انجام چنین کاری ممکن است نوع شما را از دسترس برنامه‌نویسان دیگر زبان‌ها خارج کند.

در کد زیر، یک نوع سازگار با CLS در سی‌شارپ تعریف شده است. اما نوع، تعدادی سازنده‌ی غیرسازگار با CLS دارد که باعث می‌شود کامپایلر سی‌شارپ اعلام هشدار (Warning) کند.

```
using System;
```

```
// Tell compiler to check for CLS compliance
[assembly: CLSCompliant(true)]
```

```
namespace SomeLibrary {
```

```
    // warnings appear because the class is public
    public sealed class SomeLibraryType {
```

```
        // warning: Return type of 'SomeLibrary.SomeLibraryType.Abc()'
        // is not CLS-compliant
        public UInt32 Abc() { return 0; }
```

```
        // warning: Identifier 'SomeLibrary.SomeLibraryType.abc()'
        // differing only in case is not CLS-compliant
        public void abc() { }
```

```
        // No warning: this method is private
        private UInt32 ABC() { return 0; }
```

```

}
}

```

در این کد، صفت **[assembly:CLSCompliant(true)]** بر اسمبلی اعمال شده است. این صفت به کامپایلر می‌گوید که مطمئن شود هر نوع عمومی (public) دارای ساختاری (construct) نباشد که از دسترسی به کد توسط هر زبان دیگری جلوگیری کند. وقتی این کد کامپایل می‌شود، کامپایلر سی‌شارپ دو هشدار اعلام می‌کند. هشدار اول به دلیل آنکه متد **Abc** یک عدد بدون علامت را بر می‌گرداند، گزارش شده است؛ برخی دیگر زبان‌ها نمی‌توانند به مقادیر اعداد بدون علامت دسترسی داشته باشند. هشدار دوم برای اینست که دو متد عمومی که فقط از لحاظ نوع برگشتی با هم تفاوت دارند، تعریف شده‌اند: **Abc** و **abc**. ویژگی‌های بیسیک و برخی دیگر زبان‌ها نمی‌توانند هر دو متد را فراخوانی کنند.

جالب است که اگر **public** را از ابتدای **'sealed class SomeLibraryType'** بردارید و دوباره کامپایل کنید، هر دو هشدار از بین می‌رود. علت آنست که نوع **SomeLibraryType** به صورت پیشفرض **internal** می‌شود، بنابراین دیگر از خارج اسمبلی استفاده نمی‌شود. برای لیست کاملی از قوانین CLS به بخش **"Cross Language Interoperability"** در مستندات SDK دات‌نت مراجعه کنید. (<http://msdn.microsoft.com/en-us/library/730f1wy3.aspx>)

بگذارید قوانین CLS را خلاصه کنیم. در CLR، هر عضو از یک نوع، یک فیلد (داده) یا یک متد (رفتار) است. این یعنی هر زبان برنامه‌نویسی باید به فیلدها دسترسی داشته و بتواند متدها را فراخوانی کند. فیلدهای خاص و متدهای خاص در روش‌های خاص و مشترک استفاده می‌شوند. برای سادگی برنامه‌نویسی، زبان‌ها معمولاً مفاهیم انتزاعی را برای الگوهای برنامه‌نویسی رایج ارائه می‌کنند. برای نمونه، زبان‌ها مفاهیمی چون شمارشی^{۲۶}، آرایه‌ها، ویژگی‌ها، ایندکسر^{۲۷}، نماینده^{۲۸}، رویدادها، سازنده‌ها، به اتمام رسانی کننده^{۲۹}، سربارگذاری عملگرها، عملگرهای تبدیل و ... را ارائه می‌کنند. وقتی یک کامپایلر به یکی از این‌ها در کد شما می‌رسد، باید این ساختارها را به فیلدها و متدها ترجمه کند تا CLR و هر زبان دیگری بتواند به آن‌ها دسترسی داشته باشد.

به تعریف نوع زیر توجه کنید: این تعریف شامل یک سازنده، یک به اتمام رسانی کننده، برخی سربارگذاری عملگرها، یک ویژگی، یک ایندکسر و یک رویداد است. کد زیر روش صحیح پیاده‌سازی یک نوع را نشان نمی‌دهد و فقط کامپایل می‌شود.

```

using System;
internal sealed class Test {
    // Constructor
    public Test() {}

    // Finalizer
    ~Test() {}

    // Operator overload
    public static Boolean operator == (Test t1, Test t2) {
        return true;
    }
    public static Boolean operator != (Test t1, Test t2) {
        return false;
    }

    // An operator overload
    public static Test operator + (Test t1, Test t2) { return null; }

    // A property
    public String AProperty {
        get { return null; }
    }
}

```

²⁶ enum

²⁷ indexer

²⁸ delegate

²⁹ finalizer

```

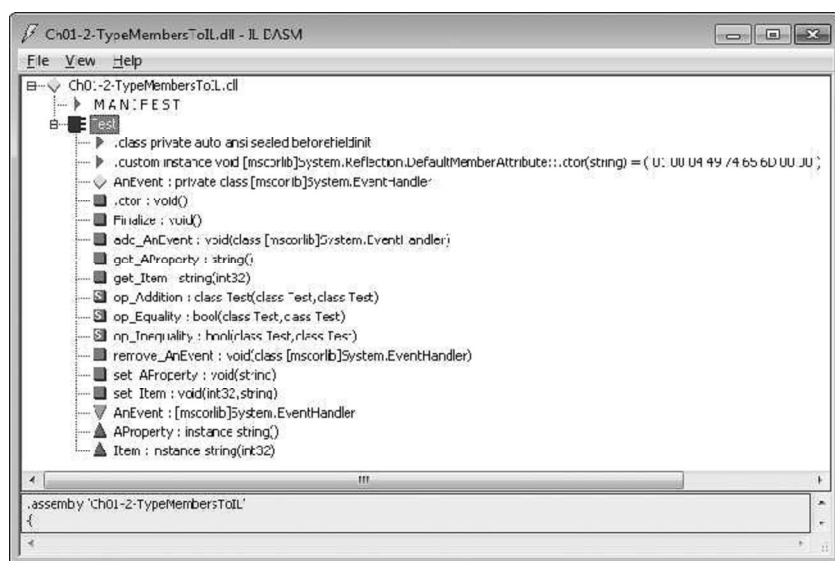
        set { }
    }

    // An indexer
    public String this[Int32 x] {
        get { return null; }
        set { }
    }

    // An event
    public event EventHandler AnEvent;
}

```

وقتی کامپایلر این کد را کامپایل می‌کند، خروجی یک نوع است که تعدادی فیلد و متد دارد. شما به راحتی می‌توانید این را با ابزار IL Disassembler (ILDasm.exe) که با SDK دات‌نت فریمورک عرضه می‌شود ببینید. بررسی این ماژول مدیریت‌شده را در شکل ۷-۱ می‌بینید.



شکل ۷-۱ ابزار ILDasm فیلدها و متدهای نوع Test را نشان می‌دهد (بدست آمده از متادیتا)

جدول ۴-۱ ساخت‌های زبان برنامه‌نویسی را با فیلدها و متدهای معادل CLR نشان می‌دهد.

جدول ۴-۱ فیلدها و متدهای نوع Test (بدست آمده از متادیتا)

عضو نوع	نوع عضو	ساخت معادل در زبان برنامه‌نویسی
AnEvent	فیلد	Event (رویداد)؛ نام فیلد AnEvent است و از نوع System.EventHandler می‌باشد.
.ctor	متد	Constructor (سازنده)
Finalize	متد	Finalizer (به اتمام رسانی کننده)
add_AnEvent	متد	Event add accessor (متد دستیابی برای اضافه کردن)
get_AProperty	متد	Property get accessor (متد دستیابی برای گرفتن ویژگی)
get_Item	متد	Indexer get accessor (متد دستیابی برای گرفتن آیتم)
op_Addition	متد	عملگر +
op_Equality	متد	عملگر ==
op_Inequality	متد	عملگر !=

remove_AnEvent	متد	Event remove accessor (متد دستیابی برای حذف)
set_AProperty	متد	Property set accessor (متد دستیابی برای تنظیم ویژگی)
set_Item	متد	Indexer set accessor (متد دستیابی برای تنظیم آیتم)

گره‌های اضافی دیگر در زیر نوع **Test** که در جدول ۴-۱ نیامده اند — **.class** ، **.custom** ، **AnEvent** ، **AProperty** و **Item** — متادیتای اضافی در مورد نوع را بیان می‌کنند. این گره‌ها بر روی فیلدها و متدها منطبق نمی‌شوند و فقط اطلاعات اضافی در مورد نوع که توسط CLR، زبان برنامه‌نویسی دیگر یا ابزاری مورد دسترسی واقع می‌شود را ارائه می‌کنند. برای نمونه، یک ابزار می‌تواند ببیند که نوع **Test** رویدادی ارائه می‌کند به نام **AnEvent** که از طریق دو متد (**remove_AnEvent** و **add_AnEvent**) در دسترس است.

تقابل با کد مدیریت نشده

دات‌نت فریمورک مزایای بسیاری نسبت به دیگر پلتفرم‌های برنامه‌نویسی دارد. اما به هر حال، شرکت‌های محدودی می‌توانند هزینه طراحی و پیاده‌سازی مجدد تمام کدهای موجود خود را تقبل کنند. مایکروسافت این را می‌داند و CLR را به گونه‌ای ساخته است تا مکانیزمی ارائه کند که یک برنامه بتواند از بخش‌های مدیریت‌شده و مدیریت نشده تشکیل شود. به خصوص، CLR سه سناریوی زیر را پشتیبانی می‌کند:

- **کد مدیریت‌شده می‌تواند یک تابع مدیریت نشده در یک DLL را فراخوانی کند** کد مدیریت‌شده به راحتی می‌تواند توابع درون DLL ها را با مکانیزمی به نام P/Invoke (برای Platform Invoke) فراخوانی کند. (توضیح مترجم: Invoke به معنای درخواست کردن است، در اینجا یعنی به جای فراخوانی مستقیم تابع، شما درخواست فراخوانی می‌کنید.) گذشته از این، بسیاری از نوع‌های تعریف شده در FCL به صورت داخلی توابعی از **Kernel32.dll** ، **User32.dll** و ... را فراخوانی می‌کنند. بسیاری از زبان‌ها مکانیزمی برای فراخوانی توابع درون DLL ها از کد مدیریت‌شده، فراهم می‌کنند. برای نمونه، یک برنامه سی‌شارپ می‌تواند تابع **CreateSemaphore** صادر شده از **Kernel32.dll** را فراخوانی کند.
- **کد مدیریت‌شده می‌تواند از یک کامپوننت COM موجود (سرور) استفاده کند** بسیاری از شرکت‌ها، قبلاً کامپوننت‌های COM مدیریت نشده تولید کرده اند. به کمک کتابخانه‌های نوع از این کامپوننت‌ها می‌توان اسمبلی مدیریت‌شده ایجاد کرد که کامپوننت COM را توصیف کند. کد مدیریت‌شده می‌تواند به نوع‌های درون اسمبلی مدیریت‌شده همانند هر نوع مدیریت‌شده دیگری دسترسی داشته باشد. برای اطلاعات بیشتر به ابزار **TlbImp.exe** (Type Library Importer) که با SDK دات‌نت فریمورک عرضه می‌شود، مراجعه کنید. گاهی شما یک کتابخانه نوع ندارید و یا می‌خواهید کنترل بیشتری بر آنچه **TlbImp.exe** تولید می‌کند داشته باشید. در این مواقع، می‌توانید نوعی را در سورس‌کد خود بسازید که CLR برای دستیابی به تقابل صحیح، از آن استفاده کند. برای نمونه، در یک برنامه سی‌شارپ می‌توانید از کامپوننت‌های **DirectX COM** استفاده کنید.
- **کد مدیریت نشده می‌تواند از نوع مدیریت‌شده استفاده کند** بسیاری از کدهای مدیریت نشده کنونی برای اجرای صحیح نیاز دارند کامپوننت COM برایشان فراهم کنید. آسانتر است که این کامپوننت‌ها را با کد مدیریت‌شده پیاده‌سازی کنید. برای نمونه می‌توانید یک کنترل **ActiveX** یا یک **shell extension** در سی‌شارپ بسازید. به ابزارهای **RegAsm.exe** و **TlbExp.exe** که با SDK دات‌نت فریمورک عرضه می‌شوند مراجعه کنید..

نکته هم اکنون مایکروسافت سورس‌کد ابزارهای **Type Library Importer** و **P/Invoke Interop Assistant** را برای کمک به برنامه‌نویسانی که با کد اصلی سروکار دارند در دسترس گذاشته است. این ابزارها و سورس‌کد آن‌ها از <http://CLRInterop.CodePlex.com/> قابل دانلود است.

فصل ۲: ساخت، بسته بندی، نصب و مدیریت برنامه ها و نوع ها

پیش از آنکه وارد فصل‌هایی از کتاب شویم که چگونگی طراحی برنامه برای دات‌نت فریمورک میکروسافت را بحث می‌کنند، ابتدا قدم‌های مورد نیاز برای ساخت، بسته‌بندی، نصب و راه‌اندازی برنامه‌ها و نوع‌های آن را خواهیم گفت. در این فصل بر ساخت اسمبلی‌هایی که تنها مورد استفاده برنامه‌ی خود شماست تمرکز می‌کنم. در فصل ۳، "اسمبلی‌های اشتراکی و اسمبلی‌های قوی‌نام" مباحث پیشرفته تری را پوشش می‌دهم که برای ساخت اسمبلی‌هایی که میان چند برنامه مشترک هستند، به آن‌ها نیاز دارید. در هر دو فصل روش‌هایی که یک مدیر می‌تواند بر اجرای برنامه‌ها و نوع‌های آن اثرگذار را نیز خواهیم گفت.

امروزه، برنامه‌ها از چندین نوع که توسط شما یا میکروسافت ساخته شده‌اند، ایجاد می‌شوند. به علاوه شرکت‌های مختلف، کامپوننت‌هایی ساخته و آن‌ها را می‌فروشند که شرکت‌های دیگر برای کاهش هزینه‌ی ساخت یک پروژه از آن‌ها استفاده می‌کنند. اگر این نوع‌ها توسط هر زبانی که با CLR کار می‌کند ساخته شده باشد، آن نوع‌ها به راحتی با یکدیگر کار می‌کنند؛ یک نوع که در یک زبان نوشته شده است بدون توجه به زبان نوعی دیگر، می‌تواند از آن نوع به عنوان نوع پایه استفاده کند.

در این فصل، چگونگی ساخت و بسته‌بندی این نوع‌ها در قالب فایل را نیز خواهیم گفت. در ادامه، بخشی از مشکلاتی که دات‌نت فریمورک آن‌ها را حل کرده است را برایتان می‌گویم.

اهداف راه‌اندازی دات‌نت فریمورک

برای سال‌ها، ویندوز میکروسافت به عدم پایداری و پیچیدگی شهرت داشته است. این سابقه، درست یا نادرست، نتیجه‌ی فاکتورهای مختلفی است. اول آنکه، همه‌ی برنامه‌های از DLL های میکروسافت یا دیگر سازندگان استفاده می‌کنند. چون، یک برنامه، کدهایی از سازندگان مختلف را اجرا می‌کند، برنامه‌نویس هر تکه کدی، ۱۰۰ درصد مطمئن نیست که چگونه این کد توسط فرد دیگری استفاده می‌شود. اگرچه این نوع تبدلات می‌تواند منجر به هرگونه مشکلی شود اما در عمل چون برنامه‌ها پیش از استفاده تست و خطایابی می‌شوند، این قبیل مشکلات کمتر بروز می‌کند.

به هر حال، کاربران معمولاً وقتی یک شرکت می‌خواهد کد خود را آپدیت کرده و فایل‌های جدید را نصب کند، با مشکل مواجه می‌شوند. به نظر می‌رسد این فایل‌ها با نسخه‌های قدیمی سازگار باشند، اما چه کسی مطمئن است؟ در واقع، وقتی یک سازنده کد خود را آپدیت می‌کند، تست مجدد و خطایابی تمام برنامه‌های تاکنون ساخته شده؛ به منظور جلوگیری از اثرات نامطلوب تغییرات، عملاً غیر ممکن است.

من مطمئنم هر کسی که این کتاب را می‌خواند، به نوعی با این مشکل مواجه شده است: هنگام نصب یک برنامه جدید، متوجه می‌شوید که بخشی از یک برنامه نصب شده را خراب کرده است. این وضعیت بد با عنوان جهنم DLL (DLL Hell) شناخته می‌شود. این‌گونه ناپایداری‌ها ترس را در ذهن و قلب کاربران عادی کامپیوتر جای می‌دهد. نتیجه نهایی این است که کاربران باید درباره‌ی نصب نرم افزار بر روی ماشین‌هایشان مراقب باشند. شخصاً، من از نصب برخی برنامه‌ها خودداری می‌کنم چرا که نگرانم بر بعضی برنامه‌های مهم که نیاز دارم، اثر بگذارد.

دلیل دوم بر شهرت ویندوز، پیچیدگی‌های نصب می‌باشد. امروزه، وقتی برنامه‌ای نصب می‌شود، همه‌ی بخش‌های سیستم را تحت تاثیر قرار می‌دهد. برای نمونه، نصب یک برنامه فایل‌هایی را کپی می‌کند، تنظیمات رجیستری را دستکاری می‌کند و میانبرهایی در دسکتاپ و منوی استارت قرار می‌دهد. علت آن است که برنامه به عنوان یک تک موجودیت ایزوله نشده است. به راحتی نمی‌توانید از برنامه خود پشتیبان تهیه کنید چرا که باید فایل‌هایی را کپی کرده و بخش‌های مرتبط در رجیستری را نیز کپی کنید. به علاوه، نمی‌توانید به راحتی برنامه را از یک ماشین به ماشین دیگر انتقال دهید؛ باید برنامه نصب را مجدداً اجرا کنید تا فایل‌ها و تنظیمات رجیستری به درستی ثبت شوند. سرانجام، به راحتی نمی‌توانید برنامه را بدون داشتن این حس بد که اجزایی از برنامه هنوز در کامپیوتر شما هستند، حذف کنید.

دلیل سوم مربوط به امنیت است. وقتی برنامه‌ها نصب می‌شوند، شامل همه نوع فایل‌هایی هستند که برخی از آن‌ها توسط شرکت‌های دیگر ساخته شده‌اند، به علاوه، برنامه‌های وب اغلب دارای کدی (مثل کنترل‌های ActiveX) هستند که بدون آنکه کاربر متوجه آن‌ها شود، دانلود می‌شوند. این کد می‌تواند هر کاری مثل حذف فایل‌ها و یا ارسال ایمیل، انجام دهد. کاربران حق دارند که از نصب برنامه‌های جدید به خاطر صدمه احتمالی آن‌ها، بترسند. برای آسایش کاربران، امنیت باید در سیستم لحاظ شود تا کاربران بتوانند به طور صریح دسترسی کدهای مختلف به منابع سیستم‌شان را کنترل کنند.

داتنت فریمورک مشکل جهنم DLL را همانطور که در این فصل و فصل ۳ می‌خوانید حل کرده است. به علاوه مشکل پراکنندگی وضعیت یک برنامه در سرتاسر دیسک یک کاربر را نیز برطرف کرده است. برای نمونه، بر خلاف COM، نوع‌ها دیگر نیاز به تنظیمات رجیستری ندارند. داتنت فریمورک دارای مدل امنیتی به نام امنیت دسترسی کد `code access security` می‌باشد. در حالیکه امنیت ویندوز مبتنی بر هویت کاربر است امنیت دسترسی کد بر پایه‌ی اجازه‌هایی است که برنامه میزبان که کامپوننت را بارگذاری می‌کند، می‌تواند داشته باشد. یک برنامه میزبان همچون `Microsoft Silverlight` اجازه‌های محدودی برای دانلود کد دارد در حالیکه برنامه‌های نصب شده محلی (خود-میزبان) با اطمینان کامل (`full trust` – همه‌ی اجازه‌ها) اجرا می‌شوند. همانطور که خواهید دید، داتنت فریمورک کاربران را قادر می‌سازد آنچه نصب می‌شود و آنچه اجرا می‌شود را کنترل کنند و در کل، ماشین‌های خود را بیش از آنچه ویندوز انجام می‌دهد، کنترل کنند.

ساخت و تبدیل نوع‌ها به یک ماژول

در این بخش، چگونگی تبدیل فایل سورس کد که شامل نوع‌های مختلفی است را به فایلی قابل اجرا توضیح خواهیم داد. با برنامه‌ی ساده زیر شروع می‌کنیم:

```
public sealed class Program {
    public static void Main() {
        System.Console.WriteLine("Hi");
    }
}
```

این برنامه یک نوع به نام `Program` تعریف می‌کند. این نوع یک متد `public` و `static` به نام `Main` دارد. درون `Main`، ارجاعی به نوع دیگر به نام `System.Console` وجود دارد. `System.Console` یک نوع پیاده‌سازی شده توسط مایکروسافت است و کد زبان میانی `Intermediate Language (IL)` که متدهای این نوع را پیاده‌سازی می‌کند درون فایل `MSCorLib.dll` قرار دارد. پس برنامه‌ی ما یک نوع را تعریف کرده است و از نوع شرکت دیگری نیز استفاده می‌کند. برای ساخت این برنامه، کد قبلی را درون یک فایل مثل `Program.cs` قرار دهید و سپس خط فرمان زیر را اجرا کنید.

```
csc.exe /out: Program.exe /t:exe /r:MSCorLib.dll Program.cs
```

این خط فرمان به کامپایلر سی‌شارپ می‌گوید یک فایل اجرایی به نام `Program.exe` (`/out: Program.exe`) تولید کند. نوع فایل تولیدی یک برنامه کنسولی `Win32` است (`/t:[target]:exe`).

وقتی کامپایلر سی‌شارپ، فایل سورس را پردازش می‌کند، می‌بیند که کد، متد `WriteLine` متعلق به نوع `System.Console` را فراخوانی کرده است و پارامتر ارسالی به این متد با آنچه متد انتظار دارد مطابق است. چون این نوع در سورس کد سی‌شارپ تعریف نشده است، برای خوشحال کردن کامپایلر سی‌شارپ، باید مجموعه اسمبلی‌هایی که برای یافتن ارجاع به نوع‌های خارجی می‌تواند به آن‌ها مراجعه کند را به آن بدهیم. در خط فرمان فوق سوییچ `MSCorLib.dll` را اضافه کرده ام که به کامپایلر می‌گوید برای یافتن نوع‌های خارجی در اسمبلی‌ای که با فایل `MSCorLib.dll` شناخته می‌شود جستجو کند.

`MSCorLib.dll` فایل ویژه‌ای است که همه‌ی نوع‌های پایه مثل `Byte`، `Char`، `String`، `Int32` و ... را داراست. این نوع‌ها به حدی استفاده می‌شوند که کامپایلر سی‌شارپ به صورت خودکار اسمبلی `MSCorLib.dll` را ارجاع می‌کند. به بیان دیگر خط فرمان زیر (بدون سوییچ `/r`) نتیجه‌ای مشابه با خط قبلی می‌دهد:

```
csc.exe /out: Program.exe /t:exe Program.cs
```

گذشته از این، چون سوییچ‌های `/out:Program.exe` و `/t:exe` با پیش فرض کامپایلر سی‌شارپ مطابقت دارند، خط فرمان زیر نتیجه‌ای مشابه تولید می‌کند:

```
csc.exe Program.cs
```

اگر به هر دلیل نمی‌خواهید که کامپایلر سی‌شارپ اسمبلی `MSCorLib.dll` را ارجاع کند از سوییچ `/nostdlib` استفاده کنید. برای نمونه خط فرمان زیر، وقتی `csc.exe` بخواهد فایل `Program.cs` را کامپایل کند، چون نوع `System.Console` در `MSCorLib.dll` تعریف شده است، خطایی را گزارش می‌کند:

```
csc.exe /out: Program.exe /t:exe /nostdlib Program.cs
```

اکنون، به فایل `Program.exe` که کامپایلر سی‌شارپ تولید کرده است نگاهی بیاندازیم. این فایل واقعا چیست؟ برای مبتدیان، این یک فایل استاندارد اجرایی (portable executable (PE) است. به این معنی که ماشینی با ویندوز ۳۲ بیتی یا ۶۴ بیتی می‌تواند آن را بارگذاری و اجرا کند. ویندوز، ۲ نوع

برنامه را پشتیبانی می‌کند، آن‌هایی که رابط کاربری کنسول (Console User Interface (CUI) و یا رابط کاربری گرافیکی (Graphical User Interface (GUI) دارند. چون من سویچ `/t:exe` را تعیین کردم، کامپایلر سی‌شارپ یک برنامه CUI تولید کرده است. برای تولید یک برنامه GUI از سویچ `/t:winexe` استفاده کنید.

فایل‌های جواب

قبل از آنکه بحث سویچ‌های کامپایلر را ترک کنیم، می‌خواهم کمی پیرامون فایل‌های جواب `response files` صحبت کنم. یک فایل جواب، یک فایل متنی شامل مجموعه‌ای از سویچ‌های کامپایلر است. وقتی `CSC.exe` را اجرا می‌کنید، کامپایلر فایل‌های جواب را باز کرده و هر سویچی که در آن‌ها تعیین شده باشد را استفاده می‌کند. شما با قرار دادن علامت `@` قبل از نام فایل جواب، به کامپایلر می‌گویید که از فایل جواب استفاده کند. برای نمونه شما می‌توانید یک فایل جواب به نام `MyProject.rsp` داشته باشید که حاوی متن زیر باشد:

```
/out:MyProject.exe
/target:winexe
```

برای آنکه `CSC.exe` از این تنظیمات استفاده کند از خط فرمان زیر استفاده کنید:

```
csc.exe @MyProject.rsp CodeFile1.cs CodeFile2.cs
```

این فایل به کامپایلر می‌گوید که نام فایل خروجی و نوع هدف چه باشد. همانطور که می‌دانید، فایل‌های جواب بسیار رایج هستند، چرا که دیگر نیاز ندارید برای هر کامپایل پروژه به صورت دستی سویچ‌ها را تعیین کنید. کامپایلر سی‌شارپ از چندین فایل جواب پشتیبانی می‌کند. علاوه بر فایل‌هایی که شما صریحاً تعیین می‌کنید، کامپایلر به صورت خودکار به دنبال فایلی به نام `CSC.rsp` می‌گردد و وقتی `CSC.exe` را اجرا می‌کند، در دایرکتوری جاری به دنبال فایل `CSC.rsp` می‌گردد. شما باید تنظیمات مربوط به یک پروژه خاص را در این فایل قرار دهید. همچنین کامپایلر در دایرکتوری حاوی `CSC.exe` به دنبال فایل سراسری `CSC.rsp` می‌گردد. تنظیماتی که می‌خواهید بر تمامی پروژه‌هایتان اعمال شود را در این فایل قرار دهید. کامپایلر تنظیمات همه‌ی این فایل‌های جواب را جمع و از آن‌ها استفاده می‌کند. اگر تنظیمات مشابه در فایل‌های جواب محلی و سراسری وجود داشته باشد، تنظیمات فایل محلی بر فایل سراسری برتری دارد. به همین طریق تنظیماتی که صریحاً از خط فرمان تعیین می‌کنید بر تنظیمات یک فایل جواب محلی برتری خواهد داشت. وقتی شما دات‌نت فریمورک را نصب می‌کنید، آن یک فایل پیش فرض سراسری `CSC.rsp` در دایرکتوری `%SystemRoot%\Microsoft.NET\Framework\vX.X.X` (که `X.X.X` شماره نسخه‌ی دات‌نت نصب شده است) قرار می‌دهد. نسخه ۴.۰ از این فایل دارای سویچ‌های زیر است:

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.
# Reference the common Framework libraries
```

```
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Deployment.dll
/r:System.Device.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
```

```

/r:System.Messaging.dll
/r:System.Numerics.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll

```

چون فایل **CSC.rsp** سراسری، تمام اسمبلی‌های لیست شده را ارجاع می‌کند، شما نیاز ندارید که این اسمبلی را صریحا با سوییچ **/reference** ارجاع دهید. این فایل جواب، کار برنامه‌نویسان را آسان می‌کند چرا که برای استفاده از این نوع‌ها و فضاهای نام مختلفی که میکروسافت عرضه کرده، هنگام کامپایل مجبور نیستند برای هر کدام، از سوییچ **/reference** استفاده کنند.

ارجاع تمام اسمبلی‌ها کمی بر سرعت کامپایلر اثر می‌گذارد، اما اگر سورس کد شما به نوع یا اعضای تعریف شده در این اسمبلی‌ها ارجاع نمی‌کند، نه بر فایل اسمبلی حاصل و نه بر عملکرد برنامه در هنگام اجرا اثری خواهد گذاشت.

نکته اگر از سوییچ **/reference** برای ارجاع به یک اسمبلی استفاده می‌کنید می‌توانید مسیر کامل یک فایل را مشخص کنید. هر چند اگر مسیر کامل را مشخص نکنید، کامپایلر مسیرهای زیر را به ترتیب جستجو می‌کند:

- دایرکتوری جاری
- دایرکتوری حاوی خود فایل **CSC.exe**. فایل **mscorlib.dll** همواره از این مسیر بدست می‌آید. این مسیر شبیه به **%SystemRoot%\Microsoft.NET\Framework\v4.0.####** است.
- هر دایرکتوری که با سوییچ **/lib** تعیین شده است.
- هر دایرکتوری که با استفاده از متغیر محلی **LIB** تعیین شده است.

البته می‌توانید سوییچ‌های خود را به فایل سراسری **CSC.rsp** اضافه کنید تا ساخت فایل‌های شما ساده تر شود، اما این کار بازسازی محیط برنامه‌نویسی در ماشین‌های متفاوت را مشکل می‌کند؛ شما مجبورید به خاطر بسیاری **CSC.rsp** را روی هر ماشین آپدیت کنید. همچنین شما می‌توانید با تعیین سوییچ **/noconfig** به کامپایلر بگویید که هر دو فایل جواب محلی و سراسری را نادیده بگیرد.

نگاهی اجمالی بر متادیتا

اکنون می‌دانیم که چه نوع فایل PE درست کرده ایم، اما واقعا در فایل **Program.exe** چه چیزی وجود دارد؟ یک فایل مدیریت شده PE، ۴ بخش اصلی دارد: هدر **PE32(+)**، هدر CLR، متادیتا و کد IL. هدر **PE32(+)** اطلاعات استاندارد است که ویندوز از آن استفاده می‌کند. هدر CLR، بلوک کوچکی از اطلاعات مختص ماژول‌هایی است که به CLR نیاز دارند (ماژول‌های مدیریت شده) است. این هدر همچنین شامل شماره کوچک و بزرگ نسخه‌ای از CLR که ماژول برای آن ساخته شده؛ تعدادی پرچم، یک نشانه **MethodDef** (بعدها توضیح داده می‌شود) که متد آغازین ماژول را برای ماژول‌های **CUI** و **GUI** اجرایی تعیین می‌کند و یک امضای دیجیتالی نام قوی انتخابی (در فصل ۳ بحث می‌شود) است. سرانجام، هدر شامل اندازه و آفست برخی جدول‌های خاص متادیتا که درون ماژول هستند می‌باشد. شما می‌توانید فرمت دقیق هدر CLR را با بررسی **IMAGE_COR20_HEADER** که در فایل هدر **CorHrd.h** تعریف شده، ببینید.

متادیتا بلوکی از داده‌های باینری است که از چندین جدول تشکیل شده است. سه دسته از این جدول‌ها وجود دارد: جدول‌های تعریف، جدول‌های ارجاع و جدول‌های مانیفست. جدول ۲-۱ رایجترین جدول‌های تعریف که در بلوک متادیتای ماژول قرار دارند را لیست می‌کند.

جدول ۱-۲ جدول‌های رایج تعریف متادیتا

نام جدول تعریف متادیتا	توضیح
ModuleDef	همیشه شامل یک ورودی است که ماژول را شناسایی می‌کند. این ورودی شامل نام ماژول و پسوند آن (بدون مسیر) و یک ID نسخه ماژول (در قالب یک GUID ایجاد شده توسط کامپایلر) می‌باشد. این اجازه می‌دهد که فایل تغییر نام داده شود در حالیکه نام اصلی آن نگهداری می‌شود. هرچند، تغییر نام یک فایل اصلاً توصیه نمی‌شود چرا که ممکن است CLR نتواند اسمبلی مورد نظر را در زمان اجرا پیدا کند.
TypeDef	شامل یک ورودی برای هر نوع تعریف شده در ماژول است. هر ورودی شامل نام نوع، نوع پایه و پرچم‌ها (public ، private و ...) و شامل اندیس متدهایش در جدول MethodDef ، فیلدهایش در جدول FieldDef ، ویژگی‌هایش در جدول PropertyDef و رویدادهایش در جدول EventDef می‌باشد.
MethodDef	شامل یک ورودی برای هر متد تعریف شده در ماژول است. هر ورودی شامل نام متد، پرچم‌ها (public ، private ، virtual ، abstract ، final و ...)، امضاء و آفستی درون ماژول که کد IL متد در آنجاست. هر ورودی همچنین می‌تواند به یک ورودی در جدول ParamDef که شامل اطلاعات پارامترهای یک متد است ارجاع کند.
FieldDef	شامل یک ورودی برای هر فیلد تعریف شده در ماژول است. هر ورودی شامل پرچم‌ها (public ، private و ...)، نوع و نام است.
ParamDef	شامل یک ورودی برای هر پارامتر تعریف شده در ماژول است. هر ورودی شامل پرچم‌ها (in ، out ، retval و ...)، نوع و نام است.
PropertyDef	شامل یک ورودی برای هر ویژگی تعریف شده در ماژول است. هر ورودی شامل پرچم‌ها، نوع و نام است.
EventDef	شامل یک ورودی برای هر رویداد تعریف شده در ماژول است. هر ورودی شامل پرچم‌ها و نام است.

وقتی کامپایلر، سورس کد شما را کامپایل می‌کند هر چیزی که کد شما تعریف می‌کند باعث می‌شود یک ورودی در یکی از جدول‌های فوق وارد شود. وقتی کامپایلر ارجاعی به نوع‌ها، فیلدها، متدها، ویژگی‌ها و رویدادها پیدا می‌کند نیز ورودی در جدول‌های متادیتا وارد می‌شود. متادیتای ساخته شده شامل جدول‌های ارجاعی است که اطلاعات موارد ارجاع داده شده را نگه می‌دارد. جدول ۲-۲ برخی از جدول‌های ارجاع را لیست می‌کند.

جدول ۲-۲ جدول‌های رایج ارجاع متادیتا

نام جدول در متادیتا	توضیح
AssemblyRef	شامل یک ورودی برای هر اسمبلی ارجاعی توسط ماژول است. هر ورودی شامل اطلاعات مورد نیاز برای اتصال اسمبلی: نام اسمبلی (بدون پسوند و مسیر)، شماره نسخه، فرهنگ و نشانه کلید عمومی (یک مقدار هش که از کلید عمومی سازنده بدست آمده و سازنده‌ی اسمبلی ارجاعی را شناسایی می‌کند) است. هر ورودی همچنین شامل پرچم‌ها و یک مقدار هش است. این مقدار هش برای کنترل بیت‌های اسمبلی ارجاعی است. CLR این مقدار هش را کاملاً نادیده گرفته و احتمالاً در آینده به این کار خود ادامه می‌دهد.
ModuleRef	شامل یک ورودی برای هر ماژول PE که نوع‌های ارجاعی توسط این ماژول را پیاده‌سازی می‌کند، است. هر ورودی همچنین شامل نام ماژول و پسوند (بدون مسیر) است. این جدول‌ها برای اتصال به نوع‌هایی که توسط ماژول‌های متفاوت از اسمبلی فراخوانی کننده، پیاده‌سازی شده اند، استفاده می‌شوند.
TypeRef	شامل یک ورودی برای هر نوع ارجاعی توسط ماژول است. هر ورودی شامل نام نوع و یک ارجاع به مکانی که نوع در آنجاست می‌باشد. اگر نوع درون نوع دیگری پیاده‌سازی شده باشد، ارجاع، یک ورودی TypeRef را نشان می‌دهد. اگر نوع در همان ماژول پیاده‌سازی شده باشد، ارجاع یک ورودی ModuleDef را نشان می‌دهد. اگر نوع، درون ماژول دیگری در همان اسمبلی پیاده‌سازی شده است، ارجاع یک ورودی ModuleRef را نشان می‌دهد و اگر نوع درون اسمبلی دیگری پیاده‌سازی شده باشد، ارجاع یک ورودی AssemblyRef را نشان می‌دهد.
MemberRef	شامل یک ورودی برای هر عضو ارجاعی توسط ماژول (فیلدها و متدها و همچنین ویژگی‌ها و متدهای رویداد) است. هر ورودی شامل نام عضو، امضاء می‌باشد و به ورودی TypeRef برای نوعی که عضو را تعریف می‌کند، اشاره دارد.

جدول‌های بسیار بیشتری از آنچه در جدول ۲-۱ و ۲-۲ لیست کردم وجود دارد، اما می‌خواستم شما را با اطلاعاتی که کامپایلر برای پردازش اطلاعات متادیتا تولید می‌کند آشنا کنم. در ابتدا گفتیم که جدول‌های مانیفست نیز وجود دارد که در پایان فصل بحث خواهیم کرد. ابزارهای متنوعی به شما اجازه می‌دهند که متادیتای درون یک فایل مدیریت‌شده PE را ببینید. علاقه شخصی من به IL Disassembler ILDasm.exe است. برای مشاهده متادیتا خط فرمان زیر را اجرا کنید:

```
ILDasm Program.exe
```

این باعث می‌شود ILDasm.exe اجرا شده و اسمبلی Program.exe را بارگذاری کند. برای مشاهده‌ی متادیتا در قالب خوب و قابل فهم، منوی View/MetaInfo/Show! را بزنید (یا CTRL + M را فشار دهید). این باعث می‌شود اطلاعات زیر ظاهر شود:

```
=====
ScopeName : test.exe
MVID      : {79CA7299-A212-4B9F-A53A-33BF7ECAFEF6}
=====
Global functions
-----

Global fields
-----

Global MemberRefs
-----

TypeDef #1 (02000002)
-----
  TypDefName: Program (02000002)
  Flags      : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
              [BeforeFieldInit] (00100101)
  Extends    : 01000001 [TypeRef] System.Object
  Method #1 (06000001) [ENTRYPOINT]
  -----
    MethodName: Main (06000001)
    Flags      : [Public] [Static] [HideBySig] [ReuseSlot] (00000096)
    RVA        : 0x00002050
    ImplFlags  : [IL] [Managed] (00000000)
    CallConvntn: [DEFAULT]
    ReturnType: Void
    No arguments.

Method #2 (06000002)
-----
  MethodName: .ctor (06000002)
  Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName]
              [RTSpecialName] [.ctor] (00001886)
  RVA        : 0x0000205e
  ImplFlags  : [IL] [Managed] (00000000)
  CallConvntn: [DEFAULT]
  hasThis
  ReturnType: Void
```

No arguments.

TypeRef #1 (01000001)

 Token: 0x01000001
 ResolutionScope: 0x23000001
 TypeRefName: System.Object
 MemberRef #1 (0a000004)

 Member: (0a000004) .ctor:
 CallCnvntn: [DEFAULT]
 hasThis
 ReturnType: Void
 No arguments.

TypeRef #2 (01000002)

 Token: 0x01000002
 ResolutionScope: 0x23000001
 TypeRefName: System.Runtime.CompilerServices.CompilationRelaxationsAttribute
 MemberRef #1 (0a000001)

 Member: (0a000001) .ctor:
 CallCnvntn: [DEFAULT]
 hasThis
 ReturnType: Void
 1 Arguments
 Argument #1: I4

TypeRef #3 (01000003)

 Token: 0x01000003
 ResolutionScope: 0x23000001
 TypeRefName: System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
 MemberRef #1 (0a000002)

 Member: (0a000002) .ctor:
 CallCnvntn: [DEFAULT]
 hasThis
 ReturnType: Void
 No arguments.

TypeRef #4 (01000004)

 Token: 0x01000004
 ResolutionScope: 0x23000001
 TypeRefName: System.Console

MemberRef #1 (0a000003)

```
-----
Member: (0a000003) writeLine:
CallConvntn: [DEFAULT]
ReturnType: Void
1 Arguments
  Argument #1: String
```

Assembly

```
-----
Token: 0x20000001
Name : test
Public Key   :
Hash Algorithm : 0x00008004
Version: 0.0.0.0
Major Version: 0x00000000
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [none] (00000000)
CustomAttribute #1 (0c000001)
```

```
-----
CustomAttribute Type: 0a000001
CustomAttributeName:
  System.Runtime.CompilerServices.CompilationRelaxationsAttribute ::
    instance void .ctor(int32)
Length: 8
Value : 01 00 08 00 00 00 00 00 > <
ctor args: (8)
```

CustomAttribute #2 (0c000002)

```
-----
CustomAttribute Type: 0a000002
CustomAttributeName:
System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::
  instance void .ctor()
Length: 30
Value : 01 00 01 00 54 02 16 57 72 61 70 4e 6f 6e 45 78 > T WrapNonEx<
       : 63 65 70 74 69 6f 6e 54 68 72 6f 77 73 01 >ceptionThrows <
ctor args: ()
```

AssemblyRef #1 (23000001)

```
-----
Token: 0x23000001
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: mscorlib
```

```

Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)

```

User Strings

```
-----
70000001 : ( 2) L"Hi"
```

```
Coff symbol name overhead: 0
=====
=====
=====
```

خوشبختانه ILDasm جدول‌های متادیتا را پردازش و ترکیب می‌کند بنابراین شما نیاز ندارید اطلاعات خام جدول‌ها را تحلیل کنید. برای نمونه، می‌بینید وقتی ILDasm یک ورودی TypeDef را نشان می‌دهد، اطلاعات تعریف عضو متناظر قبل از آنکه اولین TypeRef را نشان دهد، آمده است. شما نیاز ندارید همه‌ی این موارد را فرا بگیرید. نکته مهم این است که بدانید Program.exe شامل یک TypeDef به نام Program است. این نوع، یک کلاس عمومی مهر شده که از System.Object (یک نوع ارجاعی از یک اسمبلی دیگر) مشتق شده است را بیان می‌کند. نوع Program، دو متد Main و ctor. (سازنده) را هم تعریف می‌کند.

Main یک متد عمومی و استاتیک بوده که کد آن IL است (برخلاف کد اصلی پردازنده مثل x86). Main نوع برگشتی void دارد و آرگومانی نمی‌پذیرد. متد سازنده (که همیشه با نام ctor. نشان داده می‌شود) عمومی بوده و کد آن IL است. سازنده، نوع برگشتی void داشته و آرگومانی نمی‌پذیرد و دارای یک اشاره‌گر this است که به حافظه‌ی شی که باید هنگام فراخوانی ساخته شود، اشاره می‌کند.

قویا شما را تشویق می‌کنم که با ILDasm کار کنید. این ابزار اطلاعات با ارزشی را به شما نشان می‌دهد. و هرچه بیشتر آنچه را می‌بینید، درک کنید، بهتر CLR و قابلیت‌هایش را درک خواهید کرد. همانطور که خواهید دید، من از ILDasm بسیار در این کتاب استفاده می‌کنم.

فقط برای سرگرمی، نگاهی به آمارهایی پیرامون اسمبلی Program.exe می‌اندازیم. وقتی گزینه View/Statistics از ILDasm را انتخاب می‌کنید، اطلاعات زیر نمایش داده می‌شود:

```

File size           : 3584
PE header size     : 512 (496 used)    (14.29%)
PE additional info : 1415              (39.48%)
Num.of PE sections : 3
CLR header size    : 72                ( 2.01%)
CLR meta-data size : 612                (17.08%)
CLR additional info : 0                  ( 0.00%)
CLR method headers : 2                  ( 0.06%)
Managed code      : 18                ( 0.56%)
Data               : 2048              (57.14%)
Unaccounted        : -1095             (-30.55%)

```

Num.of PE sections : 3

.text - 1024
.rsrc - 1536
.reloc - 512

CLR meta-data size : 612

Module	-	1 (10 bytes)	
TypeDef	-	2 (28 bytes)	0 interfaces, 0 explicit layout
TypeRef	-	4 (24 bytes)	
MethodDef	-	2 (28 bytes)	0 abstract, 0 native, 2 bodies
MemberRef	-	4 (24 bytes)	
ParamDef	-	2 (12 bytes)	
CustomAttribute-		2 (12 bytes)	
Assembly	-	1 (22 bytes)	
AssemblyRef	-	1 (20 bytes)	
Strings	-	184 bytes	
Blobs	-	68 bytes	
UserStrings	-	8 bytes	
Guids	-	16 bytes	
Uncategorized	-	168 bytes	

CLR method headers : 2

Num.of method bodies - 2
Num.of fat headers - 0
Num.of tiny headers - 2

Managed code : 18

Ave method size - 9

در این جا می‌توانید اندازه‌ی فایل (به بایت) و اندازه‌ی بخش‌های مختلف تشکیل دهنده‌ی فایل (به درصد و بایت) را ببینید. برای این برنامه‌ی کوچک Program، هدر PE و متادیتا، بیشتر حجم برنامه را اشغال کرده اند. در واقع کد IL فقط ۱۸ بایت فضا مصرف کرده است. البته با بزرگ شدن برنامه و استفاده بیشتر از نوع‌های خودی و ارجاع به دیگر اسمبلی‌ها و نوع‌ها، متادیتا و هدر در مقایسه با اندازه کلی فایل، کوچک می‌شوند.

نکته ضمناً ILDasm.exe دارای باگی است که بر اطلاعات اندازه فایل اثر می‌گذارد. به خصوص، شما نمی‌توانید به این اطلاعات حساب نشده، اعتماد کنید.

ترکیب ماژول‌ها و ساخت یک اسمبلی

فایل Program.exe که در بخش قبلی مورد بحث قرار گرفت بیش از یک فایل PE با متادیتاست، آن یک اسمبلی assembly نیز هست. یک اسمبلی مجموعه‌ای از یک یا چندین فایل شامل تعریف‌های نوع و فایل‌های منبع می‌باشد. یکی از فایل‌های اسمبلی برای نگهداری مانیفست manifest است. مانیفست مجموعه‌ای دیگر از جدول‌های متادیتاست که اساساً شامل نام‌های فایل‌های تشکیل دهنده‌ی اسمبلی می‌باشند. آن‌ها همچنین نسخه‌ی اسمبلی، فرهنگ، سازنده، نوع‌های عمومی صادراتی و تمام فایل‌های تشکیل دهنده‌ی اسمبلی را شامل می‌شوند.

CLR بر روی اسمبلی‌ها عمل می‌کند، بدین گونه که CLR ابتدا فایل حاوی مانیفست را بارگذاری کرده و سپس از مانیفست برای یافتن نام دیگر فایل‌های اسمبلی استفاده می‌کند. بخشی از ویژگی‌های اسمبلی‌ها که باید به خاطر بسپارید :

- یک اسمبلی نوع‌هایی با قابلیت استفاده مجدد را تعریف می‌کند.
- یک اسمبلی همراه با یک شماره نسخه است.

▪ یک اسمبلی می‌تواند حاوی اطلاعات امنیتی همراه خود باشد.

فایل‌های یک اسمبلی این ویژگی‌ها را ندارند – به جز یک فایل که حاوی جدول‌های متادیتای مانیفست است.

برای بسته‌بندی، نسخه‌بندی، ایجاد امنیت و استفاده از نوع‌ها، باید آن‌ها را در ماژول‌هایی که بخشی از یک اسمبلی هستند، قرار دهید. در اغلب موارد، یک اسمبلی از یک تک‌فایل مثل برنامه قبلی `Program.exe` تشکیل شده است. هرچند یک اسمبلی می‌تواند از چندین فایل تشکیل شده باشد: تعدادی فایل-های PE همراه با متادیتا و تعدادی فایل‌های منبع مثل فایل‌های `.gif` یا `.jpg`. شاید نگاه به یک اسمبلی به عنوان یک `EXE` یا `DLL` منطقی به شما کمک کند.

مطمئنم بسیاری از شما خوانندگان کتاب متعجب هستید که چرا مایکروسافت این مفهوم جدید اسمبلی را معرفی کرد. دلیل این است که یک اسمبلی به شما اجازه می‌دهد تا مفاهیم منطقی و فیزیکی نوع‌های با قابلیت استفاده مجدد را از همدیگر جدا کنید. برای نمونه یک اسمبلی می‌تواند از چندین نوع تشکیل شده باشد. شما می‌توانید نوع‌های پرستفاده را در یک فایل و نوع‌هایی که کمتر استفاده می‌شوند را در فایل دیگری قرار دهید. اگر اسمبلی شما با دانلود از اینترنت نصب می‌شود، چنانچه کلاینت هرگز از نوع‌های فایل کم‌استفاده استفاده نکند آن فایل ممکن است اصلاً دانلود نشود. برای نمونه، یک تولیدکننده‌ی نرم افزار مستقل³⁰ (ISV) متخصص در کنترل‌های رابط گرافیکی، ممکن است بخواهد نوع‌های `Active Accessibility` را در یک ماژول جداگانه پیاده‌سازی کند (برای تامین شرایط `Logo` مایکروسافت). تنها کاربرانی که به ویژگی‌های دسترس‌پذیری بیشتری نیاز دارند، این ماژول را دانلود می‌کنند.

شما با تعیین عنصر `codeBase` (توضیح داده شده در فصل ۳) در فایل تنظیمات برنامه، به برنامه می‌گویید که فایل‌های اسمبلی را دانلود کند. عنصر `codeBase` یک `URL` را شناسایی می‌کند که در آن آدرس تمام فایل‌های اسمبلی قرار دارد. هنگام بارگذاری فایل‌های یک اسمبلی، `CLR`، `URL` موجود در عنصر `codeBase` را بدست آورده و کش دانلود (`download cache`) ماشین را برای یافتن فایل بررسی می‌کند. اگر فایل یافت شد، آن را بارگذاری می‌کند، اگر فایل در کش نبود، `CLR` از مکانی که `URL` به آن اشاره می‌کند، فایل را به درون کش، دانلود می‌کند. اگر فایل در آن آدرس یافت نشد، `CLR` یک اکسپشن `FileNotFoundException` در زمان اجرا تولید می‌کند.

من برای استفاده از اسمبلی‌های چندفایلی، سه دلیل را شناسایی کرده‌ام:

▪ شما می‌توانید نوع‌هایتان را میان فایل‌ها تقسیم کنید تا فایل‌ها همانند سناریوی به تدریج قبلی دانلود شوند. تقسیم بندی نوع‌ها به چندین فایل اجازه می‌دهد که بسته‌بندی و نصب برنامه به صورت جزئی و تکه تکه انجام شود.

▪ شما می‌توانید فایل‌های داده یا منبع را به اسمبلی خود اضافه کنید. برای نمونه، شما می‌توانید نوعی داشته باشید که برخی اطلاعات بیمه را محاسبه می‌کند. ممکن است این نوع برای محاسبات خود به جدول‌های آماری نیاز داشته باشد. به جای آنکه جدول‌های آماری را در سورس‌کد خود قرار دهید، می‌توانید به کمک ابزاری (مثل `Assembly Linker AL.exe` که بعداً توضیح داده می‌شود) فایل داده را به عنوان بخشی از اسمبلی قرار دهید. ضمناً این فایل می‌تواند هر فرمتی داشته باشد – یک فایل متنی، یا فایل `Excel`، یک جدول `Word` و یا هر چیز دیگر – البته تا زمانی که برنامه شما می‌داند چگونه محتویات فایل را تحلیل کند.

▪ شما می‌توانید اسمبلی‌هایی درست کنید که شامل نوع‌هایی باشند که در زبان‌های برنامه‌نویسی متفاوت پیاده‌سازی شده‌اند. برای نمونه، شما می‌توانید تعدادی نوع در سی‌شارپ و تعدادی نوع در ویژوال بیسیک بیسیک بسازید و بقیه نوع‌ها را در زبان‌های دیگر پیاده‌سازی کنید. وقتی نوعی که با سی‌شارپ نوشته‌اید را کامپایل می‌کنید، کامپایلر یک ماژول تولید می‌کند. وقتی دیگر نوع‌ها که با ویژوال بیسیک ایجاد کرده‌اید را کامپایل می‌کنید، کامپایلر ماژول مجزای دیگری را تولید می‌کند. شما می‌توانید به کمک یک ابزار همه‌ی این ماژول‌ها را به یک تک اسمبلی ترکیب کنید. برای کاربرانی که از اسمبلی استفاده می‌کنند، فقط تعدادی نوع می‌بینند، برنامه‌نویسان حتی نخواهند دانست که از زبان‌های برنامه‌نویسی متفاوتی استفاده شده است. اگر بخواهید می‌توانید `ILDasm.exe` را برای هر ماژول اجرا کنید و سورس‌کد `IL` آن را بدست آورید. سپس `ILAsm.exe` را اجرا کرده و تمام فایل‌های سورس‌کد `IL` را به آن بدهید. `ILAsm.exe` یک فایل که شامل همه‌ی نوع‌هاست را تولید می‌کند. این تکنیک نیازمند آن است که کامپایلر شما فقط کد `IL` تولید کند.

³⁰ Independent Software Vendor

مهم برای به خاطر سپاری، اسمبلی یک واحد استفاده مجدد، نسخه‌بندی و امنیت است. آن به شما اجازه می‌دهد که نوع‌ها و منابع خود را به چندین فایل تقسیم کنید تا خودتان و مشتریان در مورد بسته بندی فایل‌ها و نصب آن‌ها تصمیم بگیرید. وقتی CLR فایل حاوی مانیفست را بارگذاری می‌کند، می‌تواند تعیین کند کدام یک از دیگر فایل‌های اسمبلی، دارای نوع‌هایی هستند که برنامه به آن‌ها ارجاع داده است. هر کس که از اسمبلی استفاده می‌کند فقط باید نام فایل حاوی مانیفست را بداند، تقسیم بندی فایل‌ها از دید مشتری پنهان است و بدون ایجاد مشکل در رفتار برنامه، می‌تواند در آینده تغییر کند.

اگر چندین نوع دارید که دارای شماره نسخه و تنظیمات امنیتی مشابه هستند، توصیه می‌شود به جای تقسیم نوع‌ها میان چند فایل، تمام نوع‌ها را در یک تک فایل قرار دهید. دلیل این کار کارایی است. بارگذاری یک فایل اسمبلی زمانی از CLR و ویندوز برای یافتن اسمبلی، بارگذاری آن و مقداردهی اولیه می‌گیرد. هر چه فایل‌ها/اسمبلی‌های کمتری بارگذاری شوند بهتر است چرا که اسمبلی‌های کمتر حجم کاری را کاهش داده و تکه تکه شدن فضای آدرس پردازش را کاهش می‌دهد سرانجام اینکه NGen.exe می‌تواند بهینه سازی بهتری در هنگام پردازش فایل‌های بزرگتر انجام دهد.

برای ساخت یک اسمبلی، باید یکی از فایل‌های PE را به عنوان نگهدارنده‌ی مانیفست تعیین کنید. یا می‌توانید یک فایل PE دیگر درست کنید که فقط شامل مانیفست باشد. جدول ۲-۳ جدول‌های متادیتای مانیفست را نشان می‌دهد که یک مازول مدیریت شده را به یک اسمبلی تبدیل می‌کنند.

جدول ۲-۳ جدول‌های متادیتای مانیفست

نام جدول متادیتای مانیفست	توضیح
AssemblyDef	شامل یک ورودی است اگر مازول یک اسمبلی باشد. این ورودی شامل نام اسمبلی (بدون پسوند و مسیر)، نسخه (بزرگ، کوچک، ساخت، بازبینی)، فرهنگ، پرچم‌ها، الگوریتم هش و کلید عمومی منتشر شده (که می‌تواند null باشد) می‌باشد.
FileDef	شامل یک ورودی برای هر فایل PE و منبع که بخشی از اسمبلی است، می‌باشد (به جز فایل حاوی مانیفست چراکه به عنوان تک ورودی در جدول AssemblyDef ظاهر می‌شود). ورودی شامل نام فایل و پسوند (بدون مسیر)، مقدار هش و پرچم‌هاست. اگر اسمبلی فقط از یک فایل خودش تشکیل شده باشد، FileDef هیچ ورودی‌ای ندارد.
ManifestResourceDef	شامل یک ورودی برای هر منبع که بخشی از اسمبلی است، می‌باشد. ورودی شامل نام منبع، پرچم‌ها (public) در صورتی که از بیرون اسمبلی در معرض دید باشد و private در غیر این صورت) و یک اندیس در جدول FileDef که فایل حاوی فایل منبع یا استریم را نشان می‌دهد. اگر منبع یک فایل مستقل (مثل .jpg یا .gif) نباشد، منبع یک استریم درون یک فایل PE است. برای یک منبع تعبیه شده (embedded)، ورودی همچنین شامل آفستی است که نقطه شروع استریم درون فایل PE را نشان می‌دهد.
ExportedTypesDef	شامل یک ورودی برای هر نوع عمومی صادر شده از تمام مازول‌های PE است. ورودی شامل نام نوع، یک اندیس در جدول FileDef (که نشان می‌دهد کدام فایل اسمبلی، این نوع را پیاده‌سازی کرده است) و یک اندیس در جدول TypeDef است. نکته: برای صرفه جویی فضای فایل، نوع‌هایی که از فایل حاوی مانیفست صادر شده اند، در این جدول تکرار نشده‌اند چرا که اطلاعات نوع با استفاده از جدول TypeDef متادیتا در دسترس است.

وجود مانیفست، سطحی غیر مستقیم بین استفاده کنندگان اسمبلی و جزئیات تقسیم‌بندی اسمبلی فراهم کرده و اسمبلی را خود توصیف‌گر می‌سازد. همچنین توجه کنید فایل حاوی مانیفست شامل اطلاعات متادیتاست که تعیین می‌کند کدام فایل‌ها بخشی از اسمبلی هستند، اما فایل‌های دیگر خودشان اطلاعات متادیتا که تعیین کند بخشی از اسمبلی هستند را ندارند.

نکته فایل اسمبلی که شامل مانیفست است جدول دیگری به نام AssemblyRef نیز دارد. این جدول شامل یک ورودی برای هر اسمبلی ارجاعی توسط تمام فایل‌های اسمبلی می‌باشد. این به ابزارها اجازه می‌دهد تنها با بازکردن مانیفست اسمبلی و بدون بازکردن تمام فایل‌های اسمبلی، مجموعه‌ی اسمبلی‌های ارجاعی را ببینند. مجدداً، ورودی‌های جدول AssemblyRef برای خود توصیف‌گر کردن اسمبلی هستند.

کامپایلر سی شارپ وقتی شما یکی از سوییج‌هایی که در ادامه می‌آید را تعیین می‌کنید، یک اسمبلی تولید می‌کند : **/t[target]:exe** ،
/t[target]:winexe یا **/t[target]:library** . تمام این سوییج‌ها باعث می‌شوند کامپایلر، یک تک‌فایل PE که شامل جدول‌های متادیتای مانیفست
است تولید کند. فایل تولیدی به ترتیب یک فایل اجرایی CUI، یک فایل اجرایی GUI یا یک DLL است.

علاوه بر این سوییج‌ها، کامپایلر سی شارپ سوییج **/t[target]:module** را نیز پشتیبانی می‌کند. این سوییج به کامپایلر سی شارپ می‌گوید یک فایل PE
تولید کند که جدول‌های متادیتای مانیفست را نداشته باشد. فایل PE تولیدی همواره یک فایل DLL PE است و این فایل قبل از آنکه CLR به نوع‌های
درون آن دسترسی پیدا کند باید به یک اسمبلی اضافه شود. وقتی از سوییج **/t:module** استفاده می‌کنید، کامپایلر سی شارپ به صورت پیش فرض فایل
خروجی را با پسوند **.netmodule** نامگذاری می‌کند.

مهم متاسفانه ویژگی‌های استودیو قابلیت ساخت اسمبلی‌های چندفایلی را پشتیبانی نمی‌کند. اگر می‌خواهید یک اسمبلی چند فایلی بسازید باید به ابزارهای
خط فرمانی متوسل شوید.

راه‌های مختلفی برای افزودن یک ماژول به یک اسمبلی وجود دارد. اگر از کامپایلر سی شارپ برای ساخت یک فایل PE همراه با مانیفست، استفاده می‌کنید،
می‌توانید از سوییج **/addmodule** استفاده کنید. برای درک چگونگی ساخت اسمبلی‌های چندفایلی، فرض کنید دو فایل سورس کد داریم:

- RUT.cs شامل نوع‌هایی که کمتر استفاده می‌شوند
- FUT.cs شامل نوع‌هایی که بیشتر استفاده می‌شوند

بگذارید نوع‌های با استفاده کمتر را در یک ماژول کامپایل کنیم تا کاربران اسمبلی در صورت عدم دسترسی به این نوع‌ها، نیاز به نصب این ماژول نداشته
باشند:

```
CSC /t:module RUT.cs
```

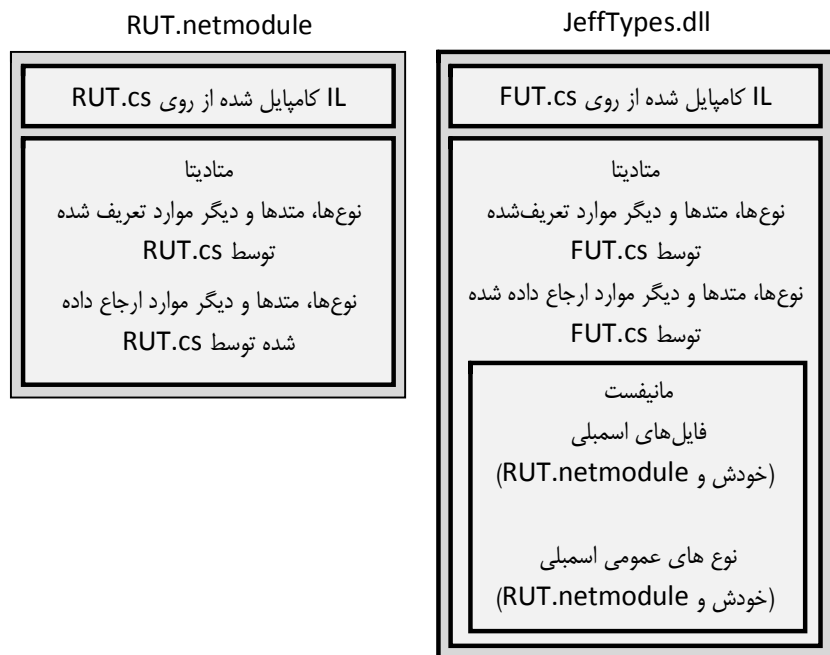
این خط باعث می‌شود کامپایلر سی شارپ یک فایل **RUT.netmodule** تولید کند. این فایل یک فایل استاندارد DLL PE است. اما به تنهایی
نمی‌تواند آن را بارگذاری کند.

حال نوع‌های با استفاده بیشتر را در ماژول دیگری کامپایل می‌کنیم. این ماژول را نگهدارنده‌ی مانیفست اسمبلی قرار می‌دهیم چرا که نوع‌های آن اغلب
مواقع استفاده می‌شوند. در واقع، چون این ماژول کل اسمبلی را معرفی می‌کند، نام فایل خروجی را به جای **FUT.dll** به **JeffTypes.dll** تغییر می‌دهیم.

```
CSC /out:JeffTypes.dll /t:library /addmodule:RUT.netmodule FUT.cs
```

این خط به کامپایلر سی شارپ می‌گوید که فایل **FUT.cs** را برای تولید فایل **JeffTypes.dll** کامپایل کند. چون سوییج **/t:library** تعیین شده است،
یک فایل DLL PE شامل جدول‌های متادیتای مانیفست در فایل **JeffTypes.dll** ریخته می‌شود. سوییج **/addmodule:RUT.netmodule** به
کامپایلر می‌گوید که **RUT.netmodule** فایلی است که باید به عنوان بخشی از اسمبلی در نظر گرفته شود. به خصوص، سوییج **/addmodule** به
کامپایلر می‌گوید که فایل را به جدول **FileDef** متادیتای مانیفست اضافه کرده و نوع‌های عمومی صادراتی از **RUT.netmodule** را به جدول
ExportedTypesDef متادیتای مانیفست بیافزاید.

وقتی کامپایلر تمام پردازش‌هایش را کامل کرد، دو فایل نمایش داده شده در شکل ۱-۲ ایجاد می‌شوند. ماژول سمت راست حاوی مانیفست است.



شکل ۱-۲ یک اسمبلی چندفایلی که از دو ماژول مدیریت شده که یکی حاوی مانیفست است، تشکیل گردیده است.

فایل RUT.netmodule حاوی کد IL می‌باشد که با کامپایل RUT.CS بدست آمده است. این فایل همچنین شامل جدول‌های متادیتا است که نوع‌ها، متدها، فیلدها، ویژگی‌ها، رویدادها و هر آنچه RUT.CS تعریف کرده را توصیف می‌کند. جدول‌های متادیتا همچنین نوع‌ها، متدها، و هر آنچه RUT.CS به آن‌ها ارجاع داده است را توصیف می‌کنند. فایل JeffTypes.dll فایل جداگانه ایست. همانند RUT.netmodule شامل کد IL است که از کامپایل FUT.CS بدست آمده و نیز شامل جدول‌های متادیتای تعریفی و ارجاعی مشابه است. اما به هر حال، JeffTypes.dll شامل جدول‌های اضافی متادیتای مانیفست است که JeffTypes.dll را به یک اسمبلی تبدیل می‌کند. این جدول‌های اضافی تمام فایل‌های تشکیل دهنده‌ی اسمبلی را توصیف می‌کنند (خود فایل JeffTypes.dll و فایل RUT.netmodule). جدول‌های متادیتای مانیفست همچنین شامل همه‌ی نوع‌های عمومی صادراتی از JeffTypes.dll و RUT.netmodule می‌باشند.

نکته در واقعیت، جدول‌های متادیتای مانیفست شامل نوع‌های صادراتی از فایل PE حاوی مانیفست، نمی‌باشند. هدف این بهینه‌سازی، کاهش بایت‌های مورد نیاز برای اطلاعات مانیفست در فایل PE است. پس عبارت "جدول‌های مانیفست همچنین شامل همه‌ی نوع‌های عمومی صادراتی از JeffTypes.dll و RUT.netmodule می‌باشند"، ۱۰۰ درصد درست نیست. اما این عبارت به درستی، آنچه مانیفست ارائه می‌دهد را بیان می‌کند.

وقتی اسمبلی JeffType.dll ساخته شد، شما می‌توانید با ILDasm.exe جدول‌های متادیتای مانیفست آن را بررسی کنید تا مطمئن شوید که فایل اسمبلی واقعا ارجاعی به نوع‌های فایل RUT.netmodule دارد. جدول‌های متادیتای FileDef و ExportedTypesDef شبیه به این هستند:

File #1 (26000001)

```
-----
Token: 0x26000001
Name : RUT.netmodule
HashValue Blob : e6 e6 df 62 2c a1 2c 59 97 65 0f 21 44 10 15 96 f2 7e db c2
Flags : [ContainsMetadata] (00000000)
```

ExportedType #1 (27000001)

```
-----
Token: 0x27000001
```

```
Name: ARarelyUsedType
Implementation token: 0x26000001
TypeDef token: 0x02000002
Flags : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
        [BeforeFieldInit](00100101)
```

از روی این می‌توانید ببینید که RUT.netmodule به عنوان یک فایل که بخشی از اسمبلی است با نشانه‌ی 0x26000001 شناخته می‌شود. از جدول ExportedTypesDef می‌توانید ببینید که یک نوع عمومی صادراتی به نام **ARarelyUsedType** وجود دارد. نشانه‌ی آن 0x26000001 است که بیان می‌کند کد IL این نوع درون فایل RUT.netmodule قرار دارد.

نکته برای کنجکاوی، نشانه‌های متادیتا مقادیر ۴ بایتی هستند. بایت بالا نوع نشانه را مشخص می‌کند (0x01=TypeRef ، 0x02=TypeDef ، 0x23=AssemblyRef ، 0x26=FileRef ، 0x27=ExportedType). برای لیست کامل، نوع شمارشی **CorTokenType** در فایل CorHdr.h که همراه SDK دات‌نت است را مشاهده کنید. سه بایت پایینی شماره‌ی ردیف در جدول متادیتای مربوطه را مشخص می‌کنند. برای نمونه، نشانه 0x26000001 به اولین ردیف جدول FileRef ارجاع می‌کند. برای بیشتر جدول‌ها، ردیف‌ها از شماره‌ی ۱ و نه ۰ آغاز می‌شوند. برای جدول TypeDef شماره‌ها از ۲ شروع می‌شوند.

هر کدی که از نوع‌های اسمبلی JeffTypes.dll استفاده می‌کند باید سوییچ `/reference:JeffTypes.dll` را هنگام کامپایل تعیین کند. این سوییچ به کامپایلر می‌گوید که اسمبلی JeffTypes.dll و تمام فایل‌های لیست شده در جدول FileDef را وقتی به دنبال یک نوع خارجی است، بارگذاری کند. کامپایلر نیاز دارد که تمام فایل‌های اسمبلی نصب شده و در دسترس باشند. اگر فایل RUT.netmodule را حذف کنید، کامپایلر سی‌شارپ خطای زیر را تولید می‌کند:

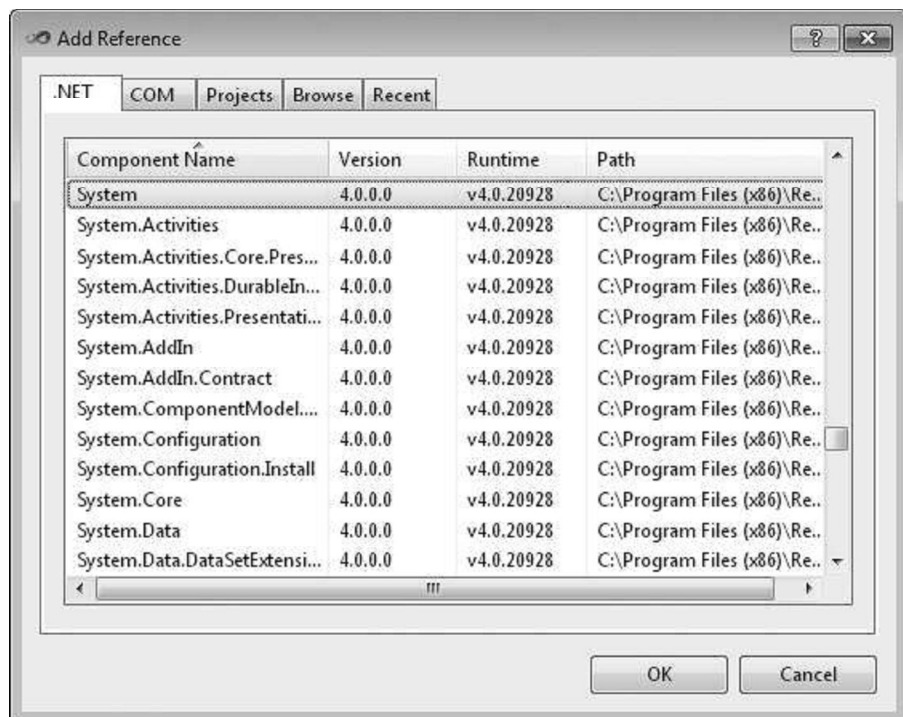
“fatal error CS0009: Metadata file 'C:\JeffTypes.dll' could not be opened— 'Error importing module 'RUT.netmodule' of assembly 'C:\JeffTypes.dll'—The system cannot find the file specified'”.

این یعنی برای ساخت یک اسمبلی جدید، تمام فایل‌های یک اسمبلی ارجاعی باید در دسترس باشند.

وقتی کد کلاینت اجرا می‌شود، متدهایی را فراخوانی می‌کند. وقتی یک متد برای اولین بار فراخوانی شد CLR نوع‌هایی که متد به آن‌ها ارجاع می‌کند مثل یک پارامتر، یک مقدار برگشتی یا یک متغیر محلی را شناسایی می‌کند. سپس CLR سعی می‌کند فایل مانیفست اسمبلی ارجاعی را پیدا کند. اگر نوع مورد استفاده در این فایل باشد، CLR سازماندهی داخلی را انجام داده و اجازه می‌دهد که نوع استفاده شود. اگر مانیفست تعیین کرد که نوع ارجاعی در فایل دیگری است، CLR سعی می‌کند فایل مورد نیاز را بارگذاری کند، سازماندهی داخلی را انجام دهد و اجازه‌ی دسترسی به نوع را فراهم کند. CLR تنها وقتی که یک متد به نوعی در یک اسمبلی بارگذاری نشده ارجاع کند، فایل‌های اسمبلی را بارگذاری می‌کند. این یعنی برای اجرای یک برنامه نیاز نیست تمام فایل‌های اسمبلی ارجاعی در دسترس باشد.

افزودن اسمبلی‌ها به یک پروژه در ویژوال استودیو

اگر از ویژوال استودیو برای ساخت پروژه خود استفاده می‌کنید، هر اسمبلی که می‌خواهید به آن ارجاع دهید را باید به پروژه‌تان اضافه کنید. برای این کار، از منوی View، Solution Explorer را باز کرده و بر روی پروژه‌ای که می‌خواهید ارجاعی به آن اضافه کنید راست کلیک کنید و گزینه Add Reference را انتخاب نمایید. این باعث می‌شود پنجره Add Reference همانند شکل ۲-۲ ظاهر شود.



شکل ۲-۲ پنجره Add Reference در ویژوال استودیو

برای ارجاع به یک اسمبلی، اسمبلی مورد نظر را از لیست انتخاب کنید. اگر اسمبلی مورد نظر شما در لیست نیست، تب Browse را بزنید تا به اسمبلی (فایل حاوی یک مانیفست) برسید. تب COM در پنجره Add Reference اجازه می‌دهد که COM مدیریت نشده از درون کد مدیریت شده از طریق یک کلاس پروکسی که ویژوال استودیو به صورت خودکار تولید می‌کند، دسترس پذیر شود. تب Projects اجازه می‌دهد که پروژه‌ی جاری، به یک اسمبلی ساخته شده توسط پروژه دیگر در همان Solution، ارجاع کند. تب Recent اجازه می‌دهد که یک اسمبلی که اخیراً به پروژه‌ی دیگری افزوده اید را اضافه کنید.

برای آنکه اسمبلی‌های خودتان در لیست تب .NET نمایش داده شود، زیرکلید زیر را به رجیستری بیافزایید:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\AssemblyFolders\MyLibName

MyLibName نام منحصر به فردی است که شما می‌سازید - ویژوال استودیو این نام را نمایش نمی‌دهد. بعد از ساخت زیرکلید، مقدار پیش فرض را به مسیری که حاوی اسمبلی‌های شماست (مثل C:\Program Files\MyLibPath) تغییر دهید. اگر از HKEY_LOCAL_MACHINE استفاده کنید، اسمبلی‌ها را برای تمام کاربران ماشین اضافه می‌کند و اگر از HKEY_CURRENT_USER استفاده کنید، اسمبلی‌ها فقط برای کاربری خاص اضافه می‌شوند.

استفاده از Assembly Linker

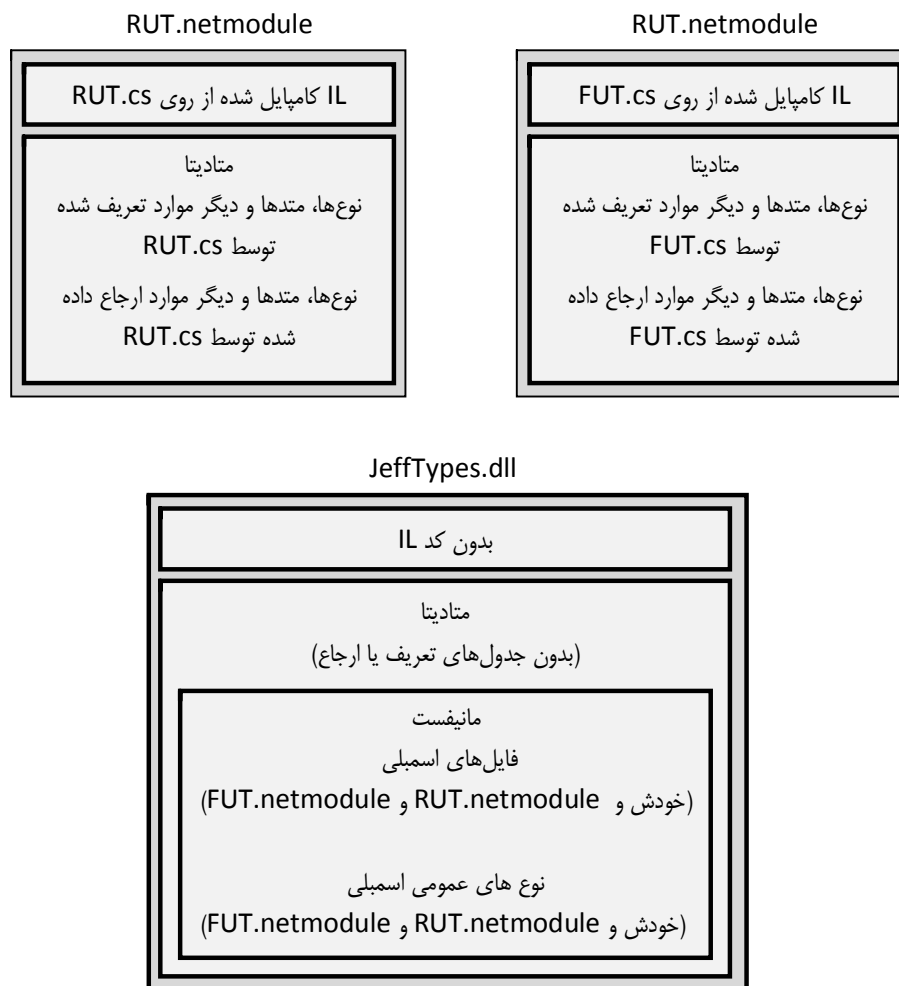
به جای استفاده از کامپایلر سی شارپ شاید بخواهید از برنامه Assembly Linker AL.exe برای ساخت اسمبلی‌هایتان استفاده کنید. Assembly Linker ابزار مفیدی برای ساخت اسمبلی‌هایی است که از ماژول‌های کامپایل شده با کامپایلرهای مختلف تشکیل می‌شوند (اگر کامپایلر شما از سویچ معادل /addmodule در سی شارپ، پشتیبانی نمی‌کند) یا شاید شما نیازهای بسته‌بندی اسمبلی خود را در زمان ساخت ندانید. همچنین می‌توانید از AL.exe برای ساخت اسمبلی‌های فقط-منبع استفاده کنید. این اسمبلی‌ها، اسمبلی‌های متحرک satellite assembly نامیده می‌شوند که اغلب برای اهداف محلی سازی (localization) استفاده می‌شوند. در پایان این فصل، پیرامون اسمبلی‌های متحرک صحبت می‌کنم.

برنامه AL.exe می‌تواند یک فایل EXE یا DLL PE که شامل یک مانیفست برای توصیف نوع‌های ماژول‌های دیگر است، تولید کند. برای درک چگونگی کار AL.exe، روش ساخت اسمبلی JeffTypes.dll را تغییر می‌دهیم:

```
CSC /t:module RUT.cs
CSC /t:module FUT.cs
```

```
al /out:JeffTypes.dll /t:library FUT.netmodule RUT.netmodule
```

شکل ۳-۲، فایل‌های حاصل از این دستورات را نشان می‌دهد.



شکل ۳-۲ یک اسمبلی چندفایلی که از سه ماژول مدیریت‌شده که یکی دارای مانیفست می‌باشد، تشکیل شده است.

در این مثال دو ماژول مجزای RUT.netmodule و FUT.netmodule ساخته شده‌اند. هیچ کدام یک اسمبلی نیستند چون جدول‌های متادیتای مانیفست را ندارند. سپس فایل سومی تولید می‌شود: JeffTypes.dll که یک فایل PE DLL (به خاطر سویچ **/t[arget]:library**) کوچک، بدون کد IL ولی حاوی جدول‌های متادیتای مانیفست است. این جدول‌ها بیان می‌کنند که RUT.netmodule و FUT.netmodule بخش‌هایی از اسمبلی هستند. اسمبلی حاصل از سه فایل تشکیل شده است: JeffTypes.dll، RUT.netmodule و FUT.netmodule. Assembly Linker راهی برای ترکیب چندین فایل به یک تک فایل ندارد.

برنامه AL.exe هم می‌تواند با تعیین سویچ‌های **/t[arget]:exe** یا **/t[arget]:winexe** فایل‌های PE از نوع CUI و GUI تولید کند. اما این کار خیلی غیرمعمول است چراکه یعنی شما یک فایل EXE PE دارید که کد IL کافی برای فراخوانی یک متد از ماژول دیگری را دارد. هنگام استفاده از AL.exe می‌توانید با تعیین سویچ **/main** تعیین کنید کدام متد در یک ماژول باید به عنوان متد آغازین لحاظ شود. مثال زیر استفاده از سویچ **/main** را در برنامه‌ی Assembly Linker نشان می‌دهد.

```
CSC /t:module /r:JeffTypes.dll Program.cs
```

```
al /out:Program.exe /t:exe /main:Program.Main Program.netmodule
```

اولین خط در اینجا، فایل Program.cs را به فایل Program.exe کامپایل می‌کند. خط دوم یک فایل PE، Program.exe، که شامل جدول‌های متادیتای مانیفست است را تولید می‌کند. به علاوه تابع سراسری کوچکی به نام **EntryPoint** وجود دارد که AL.exe به خاطر وجود سویچ خط فرمان **/main:Program.Main** تولید کرده است. این تابع **EntryPoint** حاوی کد زیر است:

```
.method privatescope static void __EntryPoint$PST06000001() cil managed
{
    .entrypoint
    // Code size 8 (0x8)
    .maxstack 8
    IL_0000: tail.
    IL_0002: call void [.module 'Program.netmodule']Program::Main()
    IL_0007: ret
} // end of method 'Global Functions':::__EntryPoint
```

همانطور که می‌بینید این متد، متد **Main** از نوع **Program** که در فایل **Program.netmodule** تعریف شده است را فراخوانی می‌کند. سوییچ **/main** آنچنان سودمند نیست چرا که بعید است شما یک اسمبلی برای یک برنامه بسازید که نقطه شروع آن در فایل PE حاوی جدول‌های متادیتای مانیفست نباشد. من فقط برای آنکه از وجود این سوییچ مطلع شوید آن را مطرح کردم. در کدهایی که با کتاب آمده‌اند یک فایل به نام **Ch02-3-4** قرار داده‌ام که همه‌ی مراحل ساخت یک اسمبلی چندفایلی را داراست. پروژه‌ی **Ch02-4** در **AppUsigMultiFileAssembly** در ویژوال استودیو به عنوان مرحله پیش‌ساخت، این فایل دسته‌ای را اجرا می‌کند. برای ساخت اسمبلی‌های چندفایلی در ویژوال استودیو، پروژه‌ی مذکور را بررسی کنید.

افزودن فایل‌های منبع به یک اسمبلی

وقتی از **AL.exe** برای ساخت یک اسمبلی استفاده می‌کنید می‌توانید فایل‌های منبع را با سوییچ **[embed[resource]]** به اسمبلی اضافه کنید. این سوییچ یک فایل (هر فایلی) دریافت می‌کند و محتویات آن را درون فایل PE خروجی جاسازی می‌کند. جدول مانیفست **ManifestResourceDef** برای انعکاس تغییرات منابع آپدیت می‌شود.

AL.exe از سوییچ **[link[resource]]** نیز پشتیبانی می‌کند که یک فایل حاوی منابع را دریافت می‌کند. سوییچ **[link[resource]]** جدول‌های **ManifestResourceDef** و **FileDef** را آپدیت می‌کند تا اعلام کند که این منابع وجود داشته و کدام یک از فایل‌های اسمبلی حاوی آن منبع است. فایل منبع درون فایل PE اسمبلی جاسازی نمی‌شود و جدا باقی می‌ماند، پس باید همراه با دیگر فایل‌های اسمبلی بسته‌بندی و نصب شود.

همانند **AL.exe**، **CSC.exe** نیز اجازه می‌دهد که منابع را به درون یک فایل اسمبلی تولیدی توسط کامپایلر سی‌شارپ، ترکیب کند. سوییچ **[resource]** به کامپایلر سی‌شارپ می‌گوید که فایل منبع را در فایل PE اسمبلی حاصل جاسازی کرده و جدول **ManifestResourceDef** را آپدیت کند. سوییچ **[linkresource]** یک ورودی به جدول‌های مانیفست **ManifestResourceDef** و **FileDef** برای ارجاع به یک فایل مستقل منبع، اضافه می‌کند.

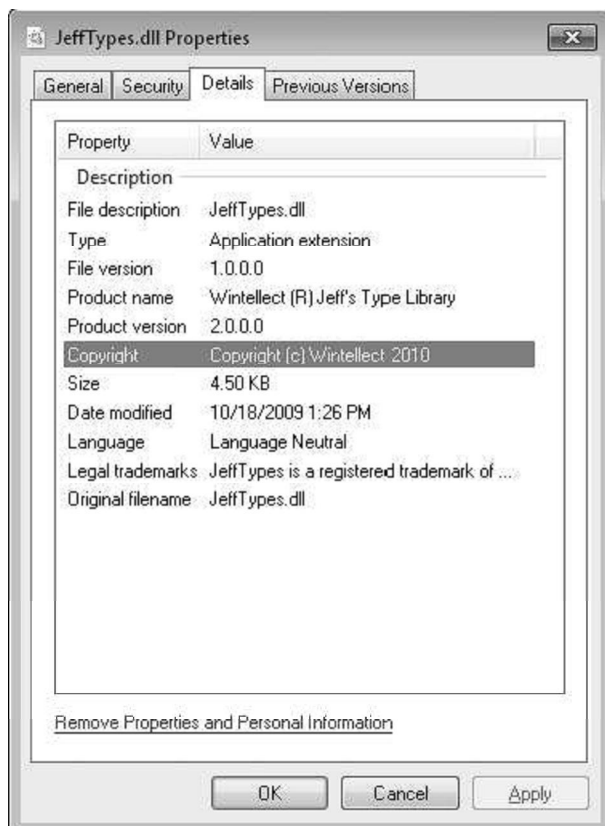
نکته آخر درباره‌ی منابع: شما می‌توانید منابع استاندارد **Win32** را در یک اسمبلی جاسازی کنید. این کار را با تعیین مسیر یک فایل **.res**. همراه با سوییچ **[win32res]** در هنگام استفاده از **AL.exe** یا **CSC.exe** انجام دهید. به علاوه به آسانی و با سرعت می‌توانید با تعیین مسیر یک فایل **.ico**. همراه با سوییچ **[win32icon]**، یک آیکون استاندارد **Win32** را در اسمبلی جاسازی کنید. هر دو **CSC.exe** و **AL.exe** از این سوییچ پشتیبانی می‌کنند. در ویژوال استودیو می‌توانید فایل‌های منبع را با نمایش **Properties** پروژه و کلیک بر تب **Application** به اسمبلی اضافه کنید. آیکون پیش فرض برای آن است که **Windows Explorer** بتواند آیکونی برای یک فایل اجرایی مدیریت‌شده نشان دهد.

نکته فایل‌های اسمبلی مدیریت شده همچنین شامل اطلاعات منابع مانیفست Win32 هستند. به صورت پیش فرض، کامپایلر سی شارپ این اطلاعات مانیفست را تولید می‌کند، اما شما می‌توانید با تعیین سویچ `/nowin32manifest` مانع این کار شوید. مانیفست پیش فرض که کامپایلر سی شارپ تولید می‌کند، شبیه این است:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker" uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

اطلاعات منبع نسخه اسمبلی

وقتی `AL.exe` یا `CSC.exe` یک فایل PE اسمبلی تولید می‌کنند، یک فایل استاندارد اطلاعات نسخه Win32 را نیز درون آن جاسازی می‌کنند. کاربران می‌توانند این اطلاعات را با نمایش Properties فایل ببینند. کد برنامه نیز می‌تواند در زمان اجرا این اطلاعات را با فراخوانی متد استاتیک `GetVersionInfo` از `System.Diagnostics.FileVersionInfo` بدست آورد. شکل ۴-۲ تب Details از پنجره Properties فایل `JeffTypes.dll` را نشان می‌دهد.



شکل ۴-۲ تب Details از پنجره Properties فایل JeffTypes.dll

هنگام ساخت یک اسمبلی باید فیلدهای اطلاعات نسخه را با اعمال صفت‌های سفارشی بر اسمبلی، در سورس کد تنظیم کنید. سورس کدی که اطلاعات نسخه مربوط به شکل ۴-۲ را تولید کرده است را در زیر می‌بینید.

```
using System.Reflection;

// FileDescription version information:
[assembly: AssemblyTitle("JeffTypes.dll")]

// Comments version information:
[assembly: AssemblyDescription("This assembly contains Jeff's types")]

// CompanyName version information:
[assembly: AssemblyCompany("wintellect")]

// ProductName version information:
[assembly: AssemblyProduct("wintellect (R) Jeff's Type Library")]

// AssemblyCopyright version information:
[assembly: AssemblyCopyright("Copyright (c) wintellect 2010")]

// LegalTrademarks version information:
[assembly: AssemblyTrademark("JeffTypes is a registered trademark of wintellect")]

// AssemblyVersion version information:
[assembly: AssemblyVersion("3.0.0.0")]

// FILEVERSION/FileVersion version information:
[assembly: AssemblyFileVersion("1.0.0.0")]

// PRODUCTVERSION/ProductVersion version information:
[assembly: AssemblyInformationalVersion("2.0.0.0")]

// Set the Language field (discussed later in the "Culture" section)
[assembly: AssemblyCulture("")]
```

مهم متأسفانه پنجره‌ی Properties از Windows Explorer اطلاعات برخی صفت‌ها را ندارد. به خصوص، خیلی عالی می‌شد اگر مقدار صفت `AssemblyVersion` را نمایش می‌داد چرا که CLR هنگام بارگذاری اسمبلی‌ها از این مقدار استفاده می‌کند (در فصل ۳ توضیح خواهیم داد).

جدول ۴-۲ فیلدهای اطلاعات نسخه و صفت‌های سفارشی متناظر با آن‌ها را نشان می‌دهد. اگر از `AL.exe` برای ساخت اسمبلی استفاده می‌کنید، می‌توانید از سویچ‌های خط فرمان به جای صفت‌های سفارشی برای تعیین این اطلاعات استفاده کنید. ستون دوم در جدول ۴-۲، سویچ خط فرمان `AL.exe` متناظر با فیلد مربوطه در اطلاعات نسخه را نشان می‌دهد. توجه کنید که کامپایلر سی‌شارپ این سویچ‌های خط فرمان را ارائه نمی‌کند و در کل استفاده از صفت‌های سفارشی برای تعیین این اطلاعات ترجیح داده می‌شود.

جدول ۴-۲: فیلدهای اطلاعات نسخه و سویچ‌های AL.exe و صفت‌های سفارشی متناظر

فیلد اطلاعات نسخه	سویچ AL.exe	صفت سفارشی
FILEVERSION	/fileversion	System.Reflection.AssemblyFileVersionAttribute
PRODUCTVERSION	/productversion	System.Reflection.AssemblyInformationalVersionAttribute
FILEFLAGS	(وجود ندارد)	همیشه به مقدار VS_FFI_FILEFLAGSMASK (تعریف شده در WinVer.h به عنوان 0x0000003F) تنظیم می‌شود.
FILEOS	(وجود ندارد)	همیشه 0 است.
FILETYPE	/target	هم اکنون همواره VOS__WINDOWS32 است.
FILESUBTYPE	(وجود ندارد)	به VFT_APP تنظیم می‌شود اگر /target:exe یا /target:winexe تعیین شده باشد. به VFT_DLL تنظیم می‌شود اگر /target:library تعیین شده باشد.
AssemblyVersion	/version	همواره به VFT2_UNKNOWN تنظیم می‌شود. (این فیلد برای VFT_APP و VFT_DLL معنی ندارد)
Comments	/description	System.Reflection.AssemblyVersionAttribute
CompanyName	/company	System.Reflection.AssemblyDescriptionAttribute
FileDescription	/title	System.Reflection.AssemblyCompanyAttribute
FileVersion	/version	System.Reflection.AssemblyTitleAttribute
InternalName	/out	System.Reflection.AssemblyFileVersionAttribute
LegalCopyright	/copyright	به نام فایل خروجی تعیین شده (بدون پسوند) تنظیم می‌شود.
LegalTrademarks	/trademark	System.Reflection.AssemblyCopyrightAttribute
OriginalFilename	/out	System.Reflection.AssemblyTrademarkAttribute
PrivateBuild	(وجود ندارد)	به نام فایل خروجی (بدون مسیر) تنظیم می‌شود.
ProductName	/product	همواره خالی است.
ProductVersion	/productversion	System.Reflection.AssemblyProductAttribute
SpecialBuild	(وجود ندارد)	همواره خالی است.

مهم هنگامیکه یک پروژه‌ی جدید سی‌شارپ در ویژوال استودیو می‌سازید، یک فایل `AssemblyInfo.cs` به صورت خودکار ایجاد می‌گردد. این فایل همه‌ی صفت‌های نسخه‌ی اسمبلی که در این بخش گفتیم به همراه تعدادی صفت دیگر که در فصل ۳ خواهیم گفت را شامل می‌شود. به سادگی می‌توانید فایل `AssemblyInfo.cs` را باز کرده و اطلاعات مربوط به پروژه‌ی خود را تغییر دهید. ویژوال استودیو پنجره‌ی محاوره‌ای برای ویرایش اطلاعات نسخه اسمبلی در این فایل، فراهم می‌کند. برای نمایش این پنجره، در `Solution Explorer` روی `Properties` پروژه‌ی خود دابل کلیک کرده و در تب `Application`، دکمه‌ی `AssemblyInformation` را کلیک کنید. پنجره‌ای شبیه شکل ۲-۵ ظاهر خواهد شد.



شکل ۲-۵ پنجره‌ی `AssemblyInformation` در ویژوال استودیو

شماره‌های نسخه (Version Numbers)

در بخش قبلی دیدید که شماره نسخه‌های مختلفی را می‌توان برای یک اسمبلی در نظر گرفت. تمام این شماره‌های نسخه، فرمت یکسانی دارند: هر کدام از چهار بخش که با نقطه از هم جدا شده، تشکیل شده‌اند؛ همانطور که در جدول ۲-۵ می‌بینید.

جدول ۲-۵ فرمت شماره‌های نسخه

	شماره بزرگ (Major)	شماره کوچک (Minor)	شماره ساخت (Build)	شماره بازبینی (Revision)
نمونه :	2	5	719	2

جدول ۲-۵ نمونه‌ای از یک شماره نسخه (2.5.719.2) را نشان می‌دهد. دو عدد اول درک عمومی از نسخه را نشان می‌دهند. عموم، این مثال را با نسخه 2.5 می‌شناسند. شماره سوم، 719، ساخت اسمبلی را نشان می‌دهد. اگر شرکت شما اسمبلی را هر روز می‌سازد، باید شماره ساخت را هر روز افزایش دهید. آخرین شماره، 2، شماره‌ی بازبینی ساخت است. اگر به هر دلیل شرکت شما مجبور شد یک اسمبلی را دوبار در روز بسازد، مثلاً برای رفع یک مشکل، شماره بازبینی باید افزایش یابد.

مایکروسافت از این فرمت شماره بندی نسخه استفاده می‌کند و قویاً توصیه می‌شود شما نیز از این قالب استفاده کنید. نسخه‌های آتی CLR پشتیبانی بهتری برای بارگذاری نسخه‌های جدید یک اسمبلی و برای بازگشت به نسخه قبلی در صورتی که نسخه جدید، برنامه کنونی را از کار بیاندازد، خواهد داشت. برای

انجام این پشتیبانی، CLR انتظار دارد که نسخه‌ای از اسمبلی که مشکلاتی را برطرف می‌کند، شماره نسخه کوچک و بزرگ (major/minor) یکسانی داشته باشد و شماره‌های ساخت و بازبینی بیانگر آپدیت‌های جدید باشند. هنگام بارگذاری یک اسمبلی، CLR به صورت خودکار آخرین نسخه نصب شده اسمبلی که دارای شماره نسخه‌ی کوچک و بزرگ یکسان با اسمبلی درخواستی باشد را پیدا می‌کند.

شما متوجه خواهید شد که یک اسمبلی دارای سه شماره نسخه است. این امر متأسفانه بسیار سردرگم کننده است. بگذارید هدف هر شماره نسخه و استفاده‌ی آن را بیان کنم:

- **AssemblyFileVersion** این شماره نسخه در اطلاعات نسخه Win32 ذخیره می‌شود. این شماره فقط برای اهداف اطلاعاتی است، CLR این شماره را اصلاً بررسی نمی‌کند. معمولاً شما بخش‌های کوچک و بزرگ شماره نسخه را برای بیان نسخه‌ای که عموم خواهند دید، تنظیم می‌کنید. سپس با هر ساخت شماره‌های ساخت و بازبینی را افزایش می‌دهید. به صورت ایده‌آل ابزارهای میکروسافت (مثل CSC.exe یا AL.exe) به صورت خودکار باید شماره‌های ساخت و بازبینی را افزایش دهند (بر طبق تاریخ و ساعت ساخت)، اما متأسفانه، این کار را نمی‌کنند. این شماره نسخه توسط Windows Explorer قابل مشاهده است و معمولاً برای شناسایی یک نسخه خاص از اسمبلی هنگام رفع مشکل مشتریان سیستم استفاده می‌شود.
- **AssemblyInformationalVersion** این شماره نسخه نیز در اطلاعات نسخه Win32 ذخیره می‌شود و مجدداً فقط برای اطلاع رسانی است؛ CLR به هیچ وجه آن را بررسی نمی‌کند. این شماره نسخه برای بیان نسخه محصولی که شامل این اسمبلی است استفاده می‌شود. برای نمونه، نسخه 2.5 از یک اسمبلی شاید حاوی چندین اسمبلی باشد، یکی از این اسمبلی‌ها نسخه 1.0 است چرا که جدید بوده و در نسخه 1.0 محصول وجود نداشته است. معمولاً بخش‌های کوچک و بزرگ این شماره نسخه را برای بیان عمومی نسخه محصول تنظیم می‌کنید. سپس با هر بسته‌بندی یک محصول کامل، بخش‌های ساخت و بازبینی را افزایش می‌دهید.
- **AssemblyVersion** این شماره نسخه در جدول متادیتای مانیفست AssemblyDef ذخیره می‌شود. CLR از این شماره نسخه برای اتصال یک اسمبلی قوی‌نام (در فصل ۳ بحث می‌شود) استفاده می‌کند. این شماره بسیار مهم بوده و برای شناسایی منحصر بفرد اسمبلی استفاده می‌شود. هنگام شروع به نوشتن یک اسمبلی، شما باید اعداد بزرگ، کوچک، ساخت و بازبینی را تنظیم کرده و آنها را مگر تا وقتی که آماده‌ی کار بر روی نسخه بعدی قابل نصب باشید، تغییر ندهید. وقتی یک اسمبلی می‌سازید، شماره نسخه‌ی اسمبلی ارجاعی در ورودی جدول AssemblyRef قرار داده می‌شود. این یعنی یک اسمبلی به یک نسخه خاص از اسمبلی ارجاعی، محکم متصل می‌شود.

فرهنگ Culture

همانند شماره‌های نسخه، اسمبلی‌ها دارای یک فرهنگ به عنوان بخشی از هویتشان هستند. برای نمونه، من می‌توانم یک اسمبلی داشته باشم که مختص آلمان باشد و اسمبلی دیگر برای آلمانی سوئیسی و اسمبلی دیگری برای انگلیسی آمریکایی و ... فرهنگ‌ها از طریق یک رشته که شامل برچسب‌های اصلی و فرعی (همانطور که در RFC1766 تعریف شده اند) است، شناسایی می‌شوند. جدول ۶-۲ تعدادی مثال را نشان می‌دهد.

جدول ۶-۲ نمونه‌هایی از برچسب‌های فرهنگ اسمبلی

فرهنگ	برچسب فرعی	برچسب اصلی
آلمانی	(ندارد)	de
آلمانی اتریشی	AT	de
آلمانی سوئیسی	CH	de
انگلیسی	(ندارد)	en
انگلیسی بریتانیایی	GB	en
انگلیسی آمریکایی	US	en

عموماً، اگر یک اسمبلی بسازید که فقط شامل کد باشد، فرهنگی به آن اختصاص نمی‌دهید. این کار به این دلیل است که کد، معمولاً فرضیات مربوط به فرهنگ را ندارد. یک اسمبلی که فرهنگی به آن اختصاص داده نشده با عنوان فرهنگ خنثی culture neutral اطلاق می‌شود.

اگر برنامه‌ای طراحی می‌کنید که منابعی با فرهنگ خاص دارد، میکروسافت قویا به شما توصیه می‌کند که یک اسمبلی بسازید که شامل کد و منابع پیش فرض برنامه شما باشد. هنگام ساخت این اسمبلی، فرهنگ خاصی را تعیین نکنید. این اسمبلی‌ای خواهد بود که دیگر اسمبلی‌ها هنگام استفاده از نوع‌های عمومی صادراتی به آن ارجاع می‌دهند.

حال می‌توانید یک یا بیشتر اسمبلی جداگانه که فقط حاوی منابع فرهنگی شما بوده و کدی در آن نیست، بسازید. اسمبلی‌هایی که با یک فرهنگ خاص علامت زده می‌شوند، اسمبلی‌های متحرک **satellite assembly** نامیده می‌شوند. برای این اسمبلی‌های متحرک یک فرهنگ که به درستی فرهنگ منابع اسمبلی را نشان می‌دهد، اختصاص دهید. شما باید برای هر فرهنگی که قصد پشتیبانی از آن را دارید، یک اسمبلی متحرک بسازید.

شما معمولاً از ابزار **AL.exe** برای ساخت اسمبلی‌های متحرک استفاده خواهید کرد. شما از یک کامپایلر استفاده نمی‌کنید، چراکه اسمبلی متحرک هیچ کدی ندارد. هنگام استفاده از **AL.exe**، فرهنگ مورد نظر را با سوییچ **/c[culture]:text** نشان می‌دهید که **text** رشته‌ای مثل **"en-US"** بیانگر انگلیسی آمریکایی (یا **"fa-IR"** برای فارسی ایرانی) است. هنگام نصب یک اسمبلی متحرک، باید آن را در زیرپوشه‌ای که نام آن با نام فرهنگ یکی باشد، قرار دهید. برای نمونه اگر دایرکتوری اصلی برنامه **C:\MyApp** باشد، اسمبلی متحرک انگلیسی آمریکایی باید در زیردایرکتوری **C:\MyApp\en-US** قرار گیرد. در زمان اجرا، به منابع اسمبلی متحرک به کمک کلاس **System.Resources.ResourceManager** دسترسی پیدا می‌کنید.

نکته امکان ساخت اسمبلی متحرک که شامل کد باشد وجود دارد، اما این کار توصیه نمی‌شود. اگر بخواهید می‌توانید فرهنگ را با صفت سفارشی **System.Reflection.AssemblyCultureAttribute** به جای سوییچ **/culture** از **AL.exe**، تعیین کنید، همانند مثال زیر:

```
// Set assembly's culture to Swiss German
[assembly:AssemblyCulture("de-CH")]
```

معمولاً، شما یک اسمبلی نمی‌سازید که یک اسمبلی متحرک را ارجاع کند. به بیان دیگر ورودی‌های **AssemblyRef** از یک اسمبلی، تماماً باید به اسمبلی‌هایی با فرهنگ خنثی ارجاع کنند. اگر می‌خواهید به نوع‌ها و اعضای یک اسمبلی متحرک دسترسی داشته باشید باید از تکنیک‌های رفلکشن که در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن" بحث می‌شود استفاده کنید.

نصب آسان برنامه (اسمبلی‌های نصب شده شخصی)

در طول این فصل چگونگی ساخت ماژول‌ها و ترکیب آن‌ها به یک اسمبلی را گفتیم. در اینجا، می‌خواهم چگونگی بسته‌بندی و نصب تمام اسمبلی‌ها برای اجرای برنامه را بگویم.

اسمبلی‌ها به هیچ روش خاصی برای بسته‌بندی نیاز ندارند. ساده‌ترین روش بسته‌بندی مجموعه‌ای از اسمبلی‌ها این است که تمام فایل‌ها را براحتی کپی کنید. برای نمونه، شما می‌توانید تمام فایل‌های اسمبلی را بر روی سی‌دی قرار داده و همراه با یک فایل نصب دسته‌ای که فایل‌ها را از سی‌دی به یک پوشه در هارددیسک کپی می‌کند، به مشتری بدهید.

چون اسمبلی‌ها همه‌ی نوع‌ها و اسمبلی‌های ارجاعی را شامل می‌شوند، کاربر فقط برنامه را اجرا می‌کند و اسمبلی‌های ارجاعی در دایرکتوری برنامه در دسترس هستند. برای اجرای برنامه هیچ تغییری در رجیستری نیاز نیست. برای حذف برنامه، فقط کافیست تمام فایل‌ها را حذف کنید. البته می‌توانید فایل‌های اسمبلی را با مکانیزم‌های دیگری مثل فایل‌های **.cab** (که برای موارد دانلود از اینترنت که فشرده سازی فایل‌ها و کاهش زمان دانلود مدنظر است) بسته‌بندی و نصب کنید. همچنین می‌توانید فایل‌های اسمبلی را در یک فایل **MSI** برای استفاده توسط **Windows Installer Service** (**MSIExec.exe**) بسته‌بندی کنید. استفاده از فایل‌های **MSI** باعث می‌شود اسمبلی‌ها بر طبق درخواست وقتی اولین بار **CLR** برای بارگذاری اسمبلی تلاش می‌کند، نصب شوند. این ویژگی **MSI** جدید نیست و برای فایل‌های **EXE** و **DLL** مدیریت نشده نیز می‌توان از این ویژگی بهره برد.

نکته استفاده از فایل دسته‌ای یا نرم افزار نصب ساده‌ی دیگر، برنامه را در ماشین کاربر نصب می‌کند اما گاهی به نرم افزار نصب پیچیده تری برای ساخت میانبر در دسکتاپ کاربر نیاز دارید. همچنین به راحتی می‌توانید از فایل‌های برنامه پشتیبان تهیه کرده یا آن را از یک ماشین به ماشین دیگر انتقال دهید، اما میانبرهای مختلف به مدیریت ویژه نیاز خواهند داشت.

البته ویژگی‌های استودیو مکانیزمی برای انتشار برنامه فراهم می‌کند. برای اینکار **Properties** پروژه را باز کرده و بر روی تب **Publish** کلیک کنید. شما می‌توانید از گزینه‌های موجود استفاده کنید تا ویژگی‌های استودیو یک فایل **MSI** تولید کرده و این فایل را در یک وب سایت، سرور **FTP** یا یک پوشه در دیسک ذخیره کند. فایل **MSI** همچنین می‌تواند هر کامپوننت پیش‌نیاز مثل **Microsoft SQL Server 2008 Express** یا **Microsoft SQL Server 2008 Express** فریمورک یا **Microsoft SQL Server 2008 Express** را در یک پوشه در دیسک ذخیره کند.

Edition را نصب کند. و در آخر، برنامه می‌تواند به صورت خودکار به کمک تکنولوژی ClickOnce وجود آپدیت را بررسی کرده و آنرا بر روی ماشین کاربر نصب کند.

اسمبلی‌هایی که در یک پوشه نصب می‌شوند، اسمبلی‌های نصب شده خصوصی **privately deployed assembly** نامیده می‌شوند چرا که فایل‌های اسمبلی با هیچ برنامه‌ی دیگری مشترک نیست (مگر آنکه برنامه‌ی دیگر در همان پوشه نصب شده باشد). اسمبلی‌های خصوصی یک برگ برنده برای برنامه‌نویسان، کاربران نهایی و مدیران هستند چرا که به راحتی در پوشه‌ی اصلی برنامه کپی می‌شوند، CLR آن‌ها را بارگذاری کرده و کد آن‌ها را اجرا می‌کند. به علاوه با حذف اسمبلی‌های این پوشه، برنامه به راحتی حذف می‌شود. این کار همچنین تهیه‌ی پشتیبان و بازگردانی آن را ساده می‌کند.

این سناریوی ساده‌ی نصب/انتقال/حذف بدین خاطر ممکن است که هر اسمبلی متادیتایی دارد که بیان می‌کند چه اسمبلی‌های ارجاعی باید بارگذاری شوند؛ هیچ تنظیم رجیستری نیاز نیست. به علاوه، اسمبلی ارجاعی قلمرو هر نوع است. این یعنی یک برنامه همیشه به همان نوعی که با آن ساخته شده است متصل می‌گردد، CLR اسمبلی متفاوتی که شاید نوعی با همان نام داشته باشد را بارگذاری نمی‌کند. این متفاوت از COM است که در آن نوع‌ها در رجیستری ثبت می‌شوند و در دسترس هر برنامه‌ای در ماشین هستند.

در فصل ۳، چگونگی نصب اسمبلی‌های اشتراکی که توسط چند برنامه قابل دسترسی هستند را بحث خواهیم کرد.

کنترل‌های مدیریتی ساده (تنظیمات)

کاربر یا مدیر بهتر می‌تواند برخی از جنبه‌های اجرایی یک برنامه را تعیین کند. برای نمونه یک مدیر شاید تصمیم بگیرد فایل‌های اسمبلی یک کاربر را جابجا کرده و اطلاعات مانیفست اسمبلی را تغییر دهد. موارد دیگری نیز چون نسخه‌بندی وجود دارد که پیرامون بعضی از این‌ها در فصل ۳ بحث خواهیم کرد.

با قرار دادن یک فایل تنظیمات در دایرکتوری برنامه، می‌توان بر برنامه کنترل‌های مدیریتی را انجام داد. منتشرکننده‌ی یک برنامه می‌تواند این فایل را ساخته و آن را بسته‌بندی کند. برنامه‌ی نصب، این فایل را در دایرکتوری برنامه نصب خواهد کرد. به علاوه، کاربر نهایی یا مدیر می‌تواند این فایل را بسازد یا محتویات آن را تغییر دهد. CLR از محتویات این فایل برای تغییر در سیاست‌هایش در یافتن و بارگذاری فایل‌های اسمبلی استفاده می‌کند.

در فصل ۳، این فایل تنظیمات را با جزئیات بیشتری بررسی می‌کنیم. اما اکنون شما را کمی با آن آشنا می‌کنم. فرض کنید سازنده‌ی یک برنامه می‌خواهد برنامه‌اش به گونه‌ای نصب شود که فایل اسمبلی **JeffTypes.dll** در پوشه‌ای متفاوت از پوشه اصلی برنامه قرار گیرد. ساختار دایرکتوری مورد نظر اینگونه است :

دایرکتوری AppDir (شامل فایل‌های اسمبلی برنامه)

Program.exe

Program.exe.config (در ادامه توضیح داده می‌شود)

زیردایرکتوری AuxFiles (شامل فایل‌های اسمبلی JeffTypes)

JeffTypes.dll

FUT.netmodule

RUT.netmodule

چون فایل‌های **JeffTypes**، دیگر در دایرکتوری اصلی برنامه قرار ندارند، CLR نمی‌تواند این فایل را یافته و بارگذاری کند، پس اجرای برنامه منجر به تولید اکسپشن **System.IO.FileNotFoundException** می‌شود. برای حل این مشکل، سازنده یک فایل XML تنظیمات درست می‌کند و آن را در دایرکتوری اصلی برنامه نصب می‌کند. نام این فایل باید نام اسمبلی اصلی برنامه با پسوند **.config** مثل **Program.exe.config** باشد. فایل تنظیمات شبیه به این است :

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="AuxFiles" />
    </assemblyBinding>
  </runtime>
</configuration>
```

وقتی CLR سعی می‌کند که یک فایل اسمبلی را پیدا کند، ابتدا همیشه دایرکتوری برنامه را می‌گردد، و اگر فایل آنجا یافت نشد، در زیردایرکتوری AuxFiles به دنبال فایل می‌گردد. شما می‌توانید چندین مسیر که با علامت سمیکالن از هم جدا شده‌اند را برای صفت privatePath از عنصر probing تعیین کنید. هر مسیر نسبت به دایرکتوری اصلی برنامه در نظر گرفته می‌شود. شما نمی‌توانید یک مسیر مطلق یا یک مسیر نسبی خارج از دایرکتوری اصلی برنامه تعیین کنید. علت آن است که یک برنامه بر دایرکتوری و زیردایرکتوری‌های خودش کنترل داشته و بر دیگر دایرکتوری‌ها کنترلی ندارد.

کاوش برای فایل‌های اسمبلی

وقتی CLR نیاز به بارگذاری یک اسمبلی دارد، چند زیردایرکتوری را جستجو می‌کند. ترتیب زیردایرکتوری‌های جستجو شده برای یافتن یک اسمبلی با فرهنگ خنثی به این گونه است (که firstPrivatePath و secondPrivatePath از طریق صفت privatePath در فایل تنظیمات تعیین می‌شوند):

```
AppDir\AsmName.dll
AppDir\AsmName\AsmName.dll
AppDir\firstPrivatePath\AsmName.dll
AppDir\firstPrivatePath\AsmName\AsmName.dll
AppDir\secondPrivatePath\AsmName.dll
AppDir\secondPrivatePath\AsmName\AsmName.dll
...
```

در این مثال، اگر فایل‌های اسمبلی JeffTypes.dll در زیرپوشه‌ای به نام JeffTypes نصب شده باشند، هیچ فایل تنظیماتی نیاز نیست، چون CLR به صورت خودکار یک زیردایرکتوری همنام با اسمبلی‌ای که به دنبال آن است را جستجو می‌کند. اگر اسمبلی در هیچ یک از زیردایرکتوری‌های مشخص شده یافت نشد، CLR از ابتدا جستجو را با پسوند .exe به جای .dll شروع می‌کند. اگر باز هم اسمبلی یافت نشد، یک اکسپشن FileNotFoundException را تولید می‌کند. برای اسمبلی‌های متحرک، قوانین مشابهی اعمال می‌شود با این تفاوت که انتظار می‌رود اسمبلی در یک زیردایرکتوری همنام با نام فرهنگ اسمبلی قرار داشته باشد. برای نمونه اگر AsmName.dll دارای فرهنگ "en-US" باشد، دایرکتوری‌های زیر جستجو می‌شوند:

```
C:\AppDir\en-US\AsmName.dll
C:\AppDir\en-US\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.dll

C:\AppDir\en-US\AsmName.exe
C:\AppDir\en-US\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.exe
```

```
C:\AppDir\en\AsmName.dll
C:\AppDir\en\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.dll
```

```
C:\AppDir\en\AsmName.exe
C:\AppDir\en\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.exe
```

همطور که می‌توانید ببینید CLR به دنبال فایل‌هایی با پسوند `.exe` یا `.dll` می‌گردد. چون این عملیات کاوش می‌تواند بسیار زمان بر باشد (به ویژه وقتی CLR در شبکه به دنبال فایل می‌گردد)، شما می‌توانید یک یا بیشتر عنصر **culture** برای محدود کردن جستجوی اسمبلی-های متحرک، در فایل XML تنظیمات تعیین کنید.

نام و مکان فایل XML تنظیمات بسته به نوع برنامه متفاوت است:

- برای برنامه‌های اجرایی (EXE)، فایل تنظیمات باید در پوشه اصلی برنامه بوده و باید دارای نام فایل EXE و پسوند `.config` در انتهای آن باشد.
- برای برنامه‌های `Microsoft ASP.NET Web Forms`، فایل باید در پوشه مجازی اصلی بوده و همواره نام `Web.config` را داشته باشد. به علاوه، زیر پوشه‌ها می‌تواند فایل `Web.config` مربوط به خود را داشته باشند و تنظیمات را به ارث ببرند. برای نمونه یک برنامه وب در آدرس `http://Wintellect.com/Training` از تنظیمات فایل‌های `Web.config` در پوشه مجازی اصلی و زیردایرکتوری `Training` استفاده می‌کند.

همانگونه که در آغاز این بخش گفته شد، تنظیمات بر یک برنامه خاص و بر ماشین اعمال می‌شوند. وقتی شما دات‌نت فریمورک را نصب می‌کنید، یک فایل `Machine.config` می‌سازد. به ازای هر نسخه نصب شده‌ی CLR در ماشین شما، یک فایل `Machine.config` وجود دارد. فایل `Machine.config` در دایرکتوری زیر قرار دارد.

`%SystemRoot%\Microsoft.NET\Framework\version\CONFIG`

`%SystemRoot%` دایرکتوری ویندوز شما را نشان می‌دهد (معمولاً `C:\Windows`) و `version` شماره نسخه از یک نسخه خاص دات‌نت فریمورک (چیزی شبیه `v4.0####`) است.

تنظیمات موجود در `Machine.config`، تنظیمات پیش فرضی را نشان می‌دهد که بر همه‌ی برنامه‌های در حال اجرا در ماشین اثرگذار است. یک مدیر می‌تواند سیاست‌هایی در سطح ماشین را با تغییر این فایل `Machine.config` اعمال کند. اما، مدیران و کاربران نهایی باید از تغییر این فایل خودداری کنند چرا که حاوی تنظیمات فراوان در مورد چیزهای مختلف است که حرکت در آن را مشکل می‌کند. به علاوه، برای آنکه بتوانید از تنظیمات برنامه پشتیبان تهیه کرده و آن را بازگردانی کنید، بهتر است تنظیمات برنامه را در فایل تنظیماتی که مخصوص آن برنامه است نگهداری کنید.

فصل ۳: اسمبلی های اشتراکی و اسمبلی های قوی نام

در فصل ۲ "ساخت، بسته‌بندی، نصب و مدیریت برنامه‌ها و نوع‌ها"، درباره‌ی مراحل ساخت، بسته‌بندی و نصب یک اسمبلی صحبت کردم. بر بیتی با عنوان نصب خصوصی تمرکز کردم که در آن اسمبلی‌ها تنها برای استفاده‌ی خود برنامه در دایرکتوری اصلی برنامه (یا زیردایرکتوری‌های آن) قرار می‌گیرند. نصب خصوصی اسمبلی‌ها کنترل کاملی بر نامگذاری، نسخه‌بندی و رفتار اسمبلی برای یک شرکت فراهم می‌کند.

در این فصل، بر ساخت اسمبلی‌هایی که توسط چندین برنامه استفاده می‌شوند، تمرکز می‌کنم. اسمبلی‌هایی که همراه با دات‌نت فریمورک میکروسافت ارائه می‌شوند یک نمونه عالی از اسمبلی‌های نصب شده سراسری هستند چرا که تمام برنامه‌های مدیریت‌شده از نوع‌های تعریف شده توسط میکروسافت در کتابخانه کلاس فریمورک (Framework Class Library (FCL استفاده می‌کنند.

همانطور که گفتم، ویندوز به عدم پایداری شهرت دارد. علت اصلی این شهرت اینست که برنامه‌ها از کدی ساخته شده‌اند که توسط فرد دیگری پیاده‌سازی شده است. گذشته از این وقتی شما یک برنامه برای ویندوز می‌نویسید، برنامه شما کدهایی که برنامه‌نویسان میکروسافت نوشته‌اند را فراخوانی می‌کند. همچنین، شرکت‌های بسیاری، کنترل‌هایی می‌سازند که برنامه‌نویسان می‌توانند در برنامه خود از آن‌ها استفاده کنند. در حقیقت دات‌نت فریمورک به این امر تشویق می‌کند و شرکت‌های بسیاری در این بخش ظاهر شده‌اند.

به مرور زمان، برنامه‌نویسان میکروسافت و برنامه‌نویسان کنترل‌ها، کدشان را تغییر می‌دهند: خطایی را برطرف می‌کنند، حفره‌های امنیتی را می‌بندند و ویژگی‌هایی به برنامه اضافه می‌کنند. در نهایت، کد جدید به ماشین کاربر می‌رسد. برنامه‌های کاربر که قبلاً نصب شده بودند و به خوبی کار می‌کردند دیگر از همان کدی که برنامه با آن تست و ساخته شده است، استفاده نمی‌کنند. در نتیجه، رفتار برنامه غیر قابل پیش‌بینی شده که منجر به عدم پایداری ویندوز می‌شود.

نسخه‌بندی فایل مسئله‌ی بسیار مشکلی است. در واقع، من ادعا می‌کنم اگر شما فایلی که توسط کد فایل‌های دیگر استفاده می‌شود را برداشته و فقط ۱ بیت آن را تغییر دهید - 0 را به 1 یا 1 را به 0 تغییر دهید - مطلقاً تضمینی وجود ندارد که فایلی که از این فایل قبل از تغییر استفاده می‌کرده است، اگر از فایل جدید استفاده کند به همان خوبی قبل کار کند. یکی از دلایل درستی این حرف این است که بسیاری از برنامه‌ها دانسته یا ندانسته دارای اشکال هستند. اگر نسخه جدید این مشکل را برطرف کند برنامه دیگر طبق انتظار کار نمی‌کند.

پس مسئله این است: شما چگونه مشکلات را برطرف کرده و ویژگی‌هایی به فایل اضافه می‌کنید در حالیکه تضمین می‌کنید هیچ برنامه‌ای از کار نخواهد افتاد؟ من به این سوال خیلی فکر کردم و به این نتیجه رسیدم: غیر ممکن است. اما، واضح است که این جواب خوبی نیست. فایل‌ها همراه با مشکل عرضه می‌شوند و شرکت‌ها همیشه می‌خواهند ویژگی‌های جدیدی را اضافه کنند. باید راهی برای عرضه فایل‌های جدید به امید آنکه برنامه به درستی کار کند، باشد. و اگر برنامه خوب کار نکند، باید راه آسانی برای بازگرداندن برنامه به آخرین وضعیت صحیح باشد. در این فصل، زیر بنایی که دات‌نت فریمورک برای حل مشکل نسخه‌بندی فراهم می‌کند را توضیح می‌دهم. بگذارید به شما هشدار دهم؛ آنچه می‌خواهم توضیح دهم پیچیده است. می‌خواهم پیرامون تعدادی الگوریتم، قوانین و سیاست‌های داخلی CLR صحبت کنم. همچنین ابزارها و برنامه‌های مختلفی که برنامه‌نویس باید استفاده کند را توضیح می‌دهم. این بحث پیچیده است چرا که حل مسئله‌ی نسخه‌بندی مشکل است.

دو نوع اسمبلی، دو نوع نصب

CLR از دو نوع اسمبلی پشتیبانی می‌کند: اسمبلی‌های ضعیف‌نام `weakly named assembly` و اسمبلی‌های قوی‌نام `strongly named assembly`.

مهم شما عبارت اسمبلی ضعیف‌نام را در هیچ یک از مستندات دات‌نت فریمورک پیدا نمی‌کنید. چرا؟ چون این عبارت را من از خودم درست کرده‌ام. در واقع، مستندات دات‌نت عبارتی برای شناسایی اسمبلی‌های ضعیف‌نام ندارد. تصمیم گرفتم برای آنکه هنگام صحبت در مورد این اسمبلی‌ها سردرگم نشوید، از این عبارت استفاده کنم.

اسمبلی‌های ضعیف‌نام و اسمبلی‌های قوی‌نام از لحاظ ساختاری یکسان هستند - بدین گونه که از فرمت فایل PE، هدر PE32(+), هدر CLR، متادیتا، جدول‌های مانیفست و IL یکسانی که در فصل ۱ "مدل اجرایی CLR" و فصل ۲ بررسی شدند، استفاده می‌کنند. و شما از ابزارهای یکسانی، مثل کامپایلر سی‌شارپ و `AL.exe` برای ساخت هر دو اسمبلی استفاده می‌کنید. تفاوت واقعی بین اسمبلی ضعیف‌نام و اسمبلی قوی‌نام آن است که یک اسمبلی قوی‌نام با

یک جفت کلید عمومی/خصوصی از سازنده، امضاء می‌شود تا سازنده‌ی اسمبلی به صورت منحصر بفرد شناسایی شود. این جفت کلید اجازه می‌دهد اسمبلی به صورت منحصر بفرد شناخته شود، ایمن شود و نسخه‌بندی روی آن صورت پذیرد و اجازه می‌دهد که اسمبلی در هر جا در ماشین کاربر یا حتی اینترنت نصب شود. این ویژگی شناسایی انحصاری یک اسمبلی به CLR اجازه می‌دهد که هنگام اتصال به یک اسمبلی قوی نام سیاست‌های امنیتی را لحاظ کند. این فصل مخصوص توضیح پیرامون اسمبلی‌های قوی نام و سیاست‌هایی که CLR بر آن‌ها اعمال می‌کند است.

یک اسمبلی به دو روش نصب می‌شود: خصوصی یا سراسری. یک اسمبلی نصب شده به صورت خصوصی، اسمبلی‌ای است که در دایرکتوری برنامه یا زیردایرکتوری‌های آن نصب شده باشد. یک اسمبلی ضعیف‌نام فقط به صورت خصوصی می‌تواند نصب شود. درباره‌ی اسمبلی‌های نصب شده به صورت خصوصی در فصل قبل توضیح دادم. یک اسمبلی نصب شده به صورت سراسری، اسمبلی‌ای است که در یک مکان شناخته شده که CLR آن مکان را هنگام جستجوی اسمبلی‌ها بررسی می‌کند، نصب می‌شود. یک اسمبلی قوی نام می‌تواند به صورت خصوصی یا سراسری نصب شود. پیرامون ساخت و نصب اسمبلی‌های قوی نام در این فصل توضیح می‌دهم.

جدول ۱-۳ انواع اسمبلی و راه‌های نصب آن‌ها را نشان می‌دهد.

جدول ۱-۳ چگونگی نصب اسمبلی‌های ضعیف‌نام و قوی نام

نوع اسمبلی	می تواند به صورت خصوصی نصب شود	می تواند به صورت سراسری نصب شود
ضعیف‌نام	بله	خیر
قوی نام	بله	بله

نکته قویا توصیه می‌شود تمام اسمبلی‌های خود را قوی نام کنید. در واقع، شاید نسخه‌های آتی CLR نیاز داشته باشند که تمام اسمبلی‌ها، قوی نام باشند و قابلیت ساخت اسمبلی‌های ضعیف‌نام برداشته شود. اسمبلی‌های ضعیف‌نام یک مشکل هستند چرا که ممکن است چندین اسمبلی با نام ضعیف یکسان داشته باشید. از جهت دیگر، اختصاص یک نام قوی برای اسمبلی، آن اسمبلی را منحصر بفرد می‌کند. اگر CLR بتواند یک اسمبلی را منحصر شناسایی کند، می‌تواند سیاست‌های بیشتری مربوط به نسخه‌بندی و سازگاری با نسخه‌های قبلی بر اسمبلی اعمال کرده و نسخه‌بندی ساده تر می‌شود. در واقع، فقط حذف قابلیت ساخت اسمبلی ضعیف نام، درک سیاست‌های نسخه‌بندی CLR را آسانتر می‌کند.

اختصاص یک نام قوی برای یک اسمبلی

اگر چندین برنامه بخواهند به یک اسمبلی دسترسی داشته باشند، اسمبلی باید در یک دایرکتوری شناخته شده قرار داشته باشد و CLR باید بداند هنگامیکه یک ارجاع به یک اسمبلی یافت می‌شود، این دایرکتوری را جستجو کند. اما یک مشکل وجود دارد: دو (یا بیشتر) شرکت می‌توانند اسمبلی‌هایی تولید کنند که دارای نام فایل یکسان باشند. سپس اگر هر دوی این اسمبلی‌ها در یک پوشه کپی شوند، آن اسمبلی که دیرتر نصب می‌شود پیروز است و تمام برنامه‌هایی که با اسمبلی قدیمی (اسمبلی‌ای که اول نصب شده است) کار می‌کردند، دیگر به درستی عمل نمی‌کنند. (این دقیقا مسئله جهنم DLL است که امروزه در ویندوز وجود دارد و هر DLL اشتراکی فقط به درون دایرکتوری System32 کپی می‌شود).

واضح است که متمایز کردن اسمبلی‌ها فقط با نام فایل چیز خوبی نیست. CLR نیاز دارد که با مکانیزمی اسمبلی‌ها را منحصر شناسایی کند. این چیزی است که عبارت اسمبلی قوی نام به آن اشاره دارد. یک اسمبلی قوی نام از چهار مشخصه که اسمبلی را یکتا می‌کنند تشکیل شده است: یک نام فایل (بدون پسوند)، یک شماره نسخه، یک هویت فرهنگی و یک کلید عمومی. چون کلیدهای عمومی اعداد بسیار بزرگی هستند، معمولا از یک عبارت هش کوچک که از کلید اصلی مشتق شده است استفاده می‌کنیم. این مقدار هش به نام نشانه‌ی کلید عمومی public hash token شناخته می‌شود. رشته‌های هویتی اسمبلی در زیر (که گاهی نام نمایشی اسمبلی assembly display name نامیده می‌شوند) چهار اسمبلی کاملا متفاوت را شناسایی می‌کنند:

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

```
"MyTypes, Version=1.0.8123.0, Culture="en-US", PublicKeyToken=b77a5c561934e089"
```

```
"MyTypes, Version=2.0.1234.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
```

اولین رشته یک فایل اسمبلی به نام **MyTypes.exe** یا **MyTypes.dll** را مشخص می‌کند (شما نمی‌توانید از روی رشته هویتی اسمبلی پسوند فایل آن را مشخص کنید). شرکت سازنده اسمبلی، نسخه 1.0.8123.0 از اسمبلی را می‌سازد و هیچ چیزی در اسمبلی به فرهنگ وابسته نیست چرا که **Culture** به **neutral** تنظیم شده است. البته هر شرکتی می‌تواند یک فایل اسمبلی با نام **MyTypes.dll** (یا **MyTypes.exe**) با شماره نسخه 1.0.8123.0 و فرهنگ خنثی تولید کند.

باید راهی برای تشخیص اسمبلی‌های این شرکت از اسمبلی‌های شرکتی دیگر که دارای مشخصه‌های یکسان است، وجود داشته باشد. به دلایل مختلف، مایکروسافت از تکنیک‌های رمزنگاری با کلیدهای عمومی/خصوصی به جای دیگر تکنیک‌های انحصاری سازی مثل **GUID**، **URL** یا **URN** ها استفاده کرده است. به خصوص، تکنیک‌های رمزنگاری روشی برای بررسی صحت بیت‌های اسمبلی هنگامی که در یک ماشین نصب می‌شوند را فراهم می‌کند و اجازه‌های امنیتی بر مبنای سازنده را ممکن می‌سازند. این تکنیک‌ها را در ادامه فصل توضیح می‌دهم. بنابراین یک شرکت که می‌خواهد اسمبلی‌هایش را انحصاری کند باید یک جفت کلید عمومی/خصوصی بسازد. سپس کلید عمومی می‌تواند با اسمبلی همراه شود. هیچ دو شرکتی نمی‌توانند کلید عمومی/خصوصی یکسانی داشته باشند و این تمایز چیز است که اجازه می‌دهد دو شرکت اسمبلی‌هایی با نام، نسخه و فرهنگ یکسان بسازند بدون آنکه نگران تداخل میان آن‌ها باشند.

نکته کلاس **System.Reflection.AssemblyName** یک کلاس کمکی برای ساخت نام یک اسمبلی و بدست آوردن بخش‌های مختلف نام یک اسمبلی است. این کلاس چندین ویژگی عمومی مثل **FullName**، **CultureInfo**، **Name**، **KeyPair** و **Version** را فراهم می‌کند. این کلاس همچنین تعدادی متد عمومی مثل **GetPublicKey**، **GetPublicKeyToken**، **SetPublicKey** و **SetPublicKeyToken** را ارائه می‌کند.

در فصل ۲ نشان دادم چگونه یک اسمبلی را نامگذاری کرده و شماره نسخه و فرهنگ آن را تنظیم کنید. یک اسمبلی ضعیف‌نام می‌تواند صفت‌های نسخه و فرهنگ اسمبلی را در متادیتای مانیفست داشته باشد، هر چند **CLR** همیشه هنگام کاوش برای اسمبلی‌های متحرک در زیردایرکتوری‌های یک اسمبلی، نسخه را نادیده گرفته و فقط اطلاعات فرهنگی را لحاظ می‌کند. چون اسمبلی‌های ضعیف‌نام همیشه به صورت خصوصی نصب می‌شوند، **CLR** از نام اسمبلی (به اضافه‌ی پسوند **.dll** یا **.exe**) هنگام جستجو برای فایل‌های اسمبلی در دایرکتوری اصلی برنامه یا هر زیردایرکتوری تعیین شده در صفت **privatePath** از عنصر **probing** در فایل **XML** تنظیمات، استفاده می‌کند.

یک اسمبلی قوی‌نام دارای یک نام فایل، یک نسخه اسمبلی و یک فرهنگ است. به علاوه یک اسمبلی قوی‌نام با کلید خصوصی سازنده‌ی آن، امضاء شده است.

اولین قدم در ساخت یک اسمبلی قوی‌نام بدست آوردن کلید به کمک برنامه **SN.exe** **Strong Name** است که همراه با **SDK** دات‌نت فریمورک و ویژوال استودیو عرضه می‌شود. این برنامه، ویژگی‌های مختلفی بسته به سویچ‌های تعیین شده برایش، ارائه می‌کند. توجه داشته باشید تمام سویچ‌های خط فرمان **SN.exe** حساس به کوچکی و بزرگی حروف هستند. برای تولید یک جفت کلید عمومی/خصوصی **SN.exe** را این گونه اجرا کنید:

```
SN -k MyCompany.snk
```

این خط به **SN.exe** می‌گوید که فایلی به نام **MyCompany.snk** تولید کند. این فایل حاوی اعداد کلید عمومی و خصوصی در فرمت باینری می‌باشد. اعداد کلید عمومی بسیار بزرگند. اگر بخواهید، می‌توانید پس از تولید فایل حاوی کلید عمومی و خصوصی، شما می‌توانید مجدداً از **SN.exe** استفاده کنید تا کلید عمومی واقعی را ببینید. برای این کار باید **SN.exe** را دو بار اجرا کنید. اولین بار، **SN.exe** را با سویچ **-p** اجرا کنید تا فایلی بسازد که فقط حاوی کلید عمومی باشد (**MyCompany.PublicKey**):

```
SN -p MyCompany.snk MyCompany.Publickey
```

سپس، **SN.exe** را با سویچ **-tp** اجرا کرده و فایلی که فقط حاوی کلید عمومی است را به آن بدهید:

```
SN -tp MyCompany.Publickey
```

وقتی من این خط را اجرا می‌کنم خروجی زیر را می‌گیرم:

```
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.20928.1
```

```
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Public key is
```

```
00240000048000009400000006020000002400005253413100040000010001003f9d621b702111
850be453b92bd6a58c020eb7b804f75d67ab302047fc786ffa3797b669215afb4d814a6f294010
b233bac0b8c8098ba809855da256d964c0d07f16463d918d651a4846a62317328cac893626a550
69f21a125bc03193261176dd629eace6c90d36858de3fcb781bfc8b817936a567cad608ae672b6
1fb80eb0
```

Public key token is 3db32f38c8b42c9a

برنامه SN.exe هیچ راهی برای نمایش کلید خصوصی ارائه نمی‌کند.

اندازه‌ی بزرگ کلیدهای عمومی کار را با آن‌ها مشکل می‌کند. جهت آسان کردن کارها برای برنامه‌نویسان (و کاربران نهایی)، نشانه کلید عمومی ساخته شده است. یک نشانه کلید عمومی یک هش ۶۴ بیتی از کلید عمومی است. سوییچ **-tp** از برنامه SN.exe در انتهای خروجی اش، نشانه کلید عمومی متناظر با کلید عمومی کامل را نشان می‌دهد.

حال که می‌دانید چگونه یک جفت کلید عمومی/خصوصی بسازید، ساخت یک اسمبلی قوی‌نام ساده است. وقتی اسمبلی خود را کامپایل می‌کنید، از سوییچ **/keyfile:<file>** همانند زیر استفاده کنید:

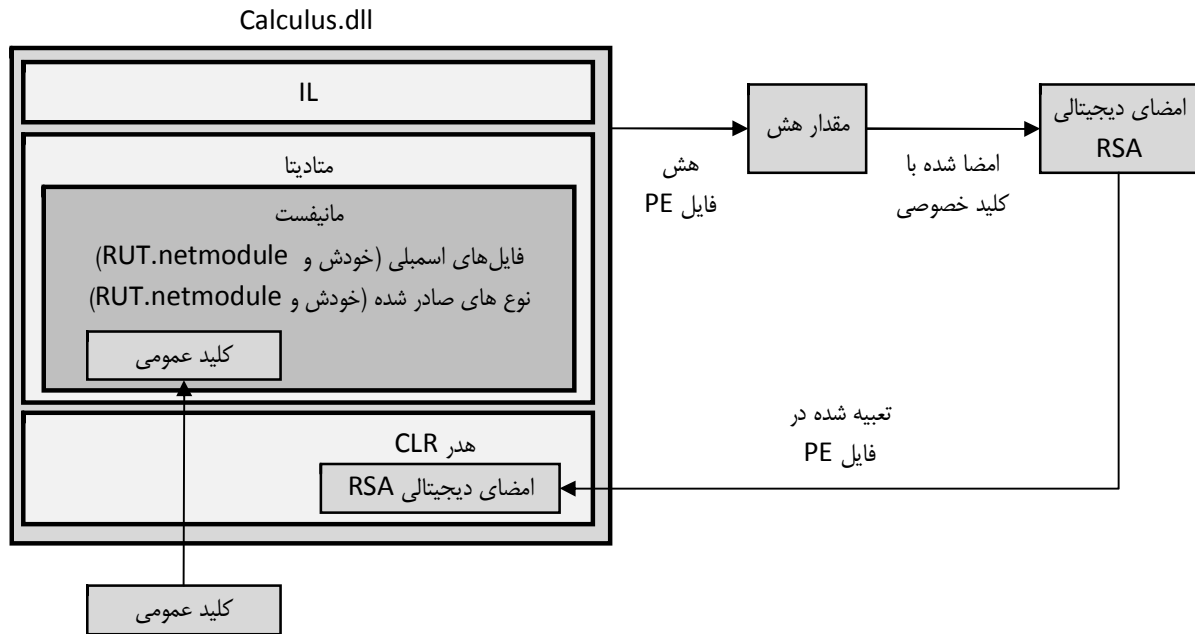
```
csc /keyfile:MyCompany.snk Program.cs
```

وقتی کامپایلر سی‌شارپ این سوییچ را می‌بیند، فایل تعیین شده (MyCompany.snk) را باز کرده، اسمبلی را با کلید خصوصی امضاء کرده و کلید عمومی را در مانیفست جاسازی می‌کند. دقت کنید که شما فقط اسمبلی حاوی مانیفست را امضاء می‌کنید، دیگر فایل‌های اسمبلی صریحاً امضاء نمی‌شوند.

اگر از ویژوال استودیو استفاده می‌کنید، می‌توانید یک فایل کلید عمومی/خصوصی بسازید. برای این کار Properties پروژه خود را باز کرده، تب Singing را کلیک کنید، گزینه‌ی Sign The Assembly را تیک زده و گزینه **<New ...>** را از لیست Choose A Strong Name Key File انتخاب کنید.

امضاء کردن یک فایل به این معنی است که: وقتی شما یک اسمبلی قوی‌نام می‌سازید، جدول متادیتای مانیفست FileDef آن شامل لیست تمام فایل‌های سازنده‌ی اسمبلی است. وقتی نام هر فایل به مانیفست اضافه می‌شود، محتویات فایل هش می‌شود و این مقدار هش همراه با نام فایل در جدول FileDef ذخیره می‌شود. شما می‌توانید الگوریتم هش پیش فرض را با تعیین سوییچ **/algid** برای AL.exe یا اعمال صفت سفارشی **System.Reflection.AssemblyAlgorithmIdAttribute** در یکی از فایل‌های سورس کد اسمبلی، تغییر دهید. به صورت پیش فرض یک الگوریتم SHA-1 استفاده می‌شود که تقریباً برای همه‌ی برنامه‌ها کافی است.

وقتی فایل PE حاوی مانیفست ایجاد شد، همانطور که در شکل ۱-۳ می‌بینید، محتویات کل فایل PE (به جز هر Authenticode Signature. اطلاعات نام قوی اسمبلی و مقدار کنترلی هدر PE) هش می‌شوند. الگوریتم هش در اینجا همیشه SHA-1 بوده و غیر قابل تغییر است. این مقدار هش با کلید خصوصی سازنده امضاء شده و امضای دیجیتالی RSA حاصل، در بخش رزرو شده‌ای درون فایل PE ذخیره می‌شود. هدر CLR از فایل PE برای انعکاس مکانی که امضای دیجیتالی درون فایل تعبیه شده است، آپدیت می‌شود.



شکل ۱-۳ امضاء کردن یک اسمبلی

همچنین کلید عمومی سازنده در جدول متادیتای مانیفست AssemblyDef در این فایل PE جاسازی می‌شود. ترکیب نام فایل، نسخه اسمبلی، فرهنگ و کلید عمومی، به یک اسمبلی یک نام قوی می‌دهد که منحصر بفرد بودن آن تضمین می‌شود. هیچ راهی وجود ندارد که دو شرکت بتوانند یک اسمبلی به نام OurLibrary تولید کنند که کلیدهای عمومی/خصوصی یکسان داشته باشند مگر آنکه دو شرکت این جفت کلید را به اشتراک بگذارند.

در این لحظه، اسمبلی و تمام فایل‌های آن آماده‌ی بسته‌بندی و توزیع هستند.

همانطور که در فصل ۵ توضیح داده‌ام، وقتی شما سورس کدتان را کامپایل می‌کنید، کامپایلر نوع‌ها و اعضایی که کد شما به آن‌ها ارجاع می‌کند را شناسایی می‌کند. شما باید اسمبلی‌های ارجاعی را برای کامپایلر تعیین کنید. برای کامپایلر سی‌شارپ از سویچ **reference** استفاده می‌کنید. بخشی از کار اسمبلی تولید جدول متادیتای AssemblyRef درون مازول مدیریت‌شده‌ی حاصل است. هر ورودی در جدول AssemblyRef، نام اسمبلی ارجاعی (بدون مسیر و پسوند)، شماره نسخه، فرهنگ و اطلاعات کلید عمومی را نشان می‌دهد.

مهم چون کلیدهای عمومی اعداد بزرگی هستند و یک تک اسمبلی شاید به چندین اسمبلی ارجاع کند، درصد زیادی از اندازه‌ی فایل خروجی با اطلاعات کلید عمومی اشغال خواهد شد. برای کاهش این فضا، مایکروسافت کلید عمومی را هش کرده و ۸ بایت پایانی آن را بر می‌دارد. این مقادیر کوچک شده‌ی کلید عمومی – که به عنوان نشانه کلید عمومی **public key token** شناخته می‌شوند – در حقیقت آن چیزی هستند که در جدول AssemblyRef ذخیره می‌شوند. معمولاً، برنامه‌نویسان و کاربران مقادیر نشانه کلید عمومی را بیش از مقادیر کلید عمومی کامل، می‌بینند.

دقت کنید که CLR هرگز از نشانه‌های کلید عمومی برای ایجاد امنیت یا هنگام گرفتن تصمیمات اطمینانی استفاده نمی‌کند چرا که ممکن است چند کلید عمومی به یک نشانه کلید عمومی هش شوند.

اطلاعات متادیتای AssemblyRef (بدست آمده از ILDasm.exe) برای نوع JeffTypes.dll که در فصل ۲ بحث شد را در اینجا می‌بینید.

AssemblyRef#1 (23000001)

Token: 0x23000001

Public Key or Token: b7 7a 5c 56 19 34 e0 89

Name: mscorlib

Version: 4.0.0.0

Major Version: 0x00000004

Minor Version: 0x00000000
 Build Number: 0x00000000
 Revision Number: 0x00000000
 Locale: <null>
 HashValue Blob:
 Flags: [none] (00000000)

از روی این اطلاعات می‌بینید که JeffTypes.dll به یک نوع اشاره می‌کند که در یک اسمبلی با مشخصات زیر جای دارد:

"MSCorLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

متأسفانه، ILDasm.exe به جای واژه‌ی Culture از Locale استفاده می‌کند. اگر شما به جدول متادیتای AssemblyDef از JeffTypes.dll نگاه کنید، این را می‌بینید:

Assembly

```
-----
Token: 0x20000001
Name : JeffTypes
Public Key :
Hash Algorithm : 0x00008004
Version: 3.0.0.0
Major Version: 0x00000003
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [none] (00000000)
```

این معادل خط زیر است:

“JeffTypes, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null”

در این خط، هیچ نشانه کلید عمومی تعیین نشده است چون در فصل ۲، اسمبلی JeffTypes.dll با یک جفت کلید عمومی/خصوصی امضاء نشده بود و یک اسمبلی ضعیف‌نام بود. اگر با SN.exe کلیدی می‌ساختم و اسمبلی را با سوئیچ **/keyfile** کامپایل می‌کردم، اسمبلی حاصل امضاء شده می‌بود. سپس اگر با ILDasm.exe متادیتای اسمبلی را بررسی می‌کردم، ورودی AssemblyDef دارای بایت‌هایی برای فیلد Public Key بوده و اسمبلی قوی‌نام می‌بود. ضمناً، ورودی AssemblyDef همواره دارای کلید عمومی کامل و نه نشانه‌ی آن است. کلید عمومی کامل برای بررسی صحت و عدم نفوذ در فایل، مورد نیاز است. ضد نفوذ بودن اسمبلی‌های قوی‌نام را در پایان فصل توضیح می‌دهم.

کش سراسری اسمبلی The Global Assembly Cache

حال که از چگونگی ساخت یک اسمبلی قوی‌نام آگاهید، زمان یادگیری چگونگی نصب آن و اینکه CLR چگونه از این اطلاعات برای یافتن اسمبلی و بارگذاری آن استفاده می‌کند، فرارسیده است.

اگر یک اسمبلی توسط چندین برنامه استفاده می‌شود، اسمبلی باید در مکانی شناخته شده جای بگیرد و CLR باید بداند این مکان را هنگامیکه یک ارجاع به اسمبلی یافت می‌شود، مورد جستجو قرار دهد. این مکان شناخته‌شده، کش سراسری اسمبلی (GAC) نامیده می‌شود که معمولاً در زیردایرکتوری زیر قرار دارد (به شرطی که ویندوز در دایرکتوری C:\Windows نصب شده باشد):

C:\Windows\Assembly

دایرکتوری GAC سازماندهی شده است: این دایرکتوری شامل چندین زیردایرکتوری می‌باشد و یک الگوریتم برای تولید نام زیردایرکتوری‌ها استفاده می‌شود. شما هرگز نباید به صورت دستی فایل‌های اسمبلی را در GAC کپی کنید و باید به جای آن از ابزارها برای دستیابی به هدف مورد نظر خود بهره ببرید. این ابزارها ساختار داخلی GAC و نحوه‌ی تولید نام صحیح برای زیردایرکتوری را می‌دانند.

رایجترین ابزار برای نصب یک اسمبلی قوی‌نام در GAC، GACUtil.exe می‌باشد. اجرا کردن این ابزار بدون هیچ آرگومانی، خروجی زیر را تولید می‌کند:

Microsoft (R) .NET Global Assembly Cache Utility. version 4.0.20928.1

Copyright (c) Microsoft Corporation. All rights reserved.

Usage: Gacutil <command> [<options>]

Commands:

`/i <assembly_path> [/r <...>] [/f]`

Installs an assembly to the global assembly cache.

`/il <assembly_path_list_file> [/r <...>] [/f]`

Installs one or more assemblies to the global assembly cache.

`/u <assembly_display_name> [/r <...>]`

Uninstalls an assembly from the global assembly cache.

`/ul <assembly_display_name_list_file> [/r <...>]`

Uninstalls one or more assemblies from the global assembly cache.

`/l [<assembly_name>]`

List the global assembly cache filtered by <assembly_name>

`/lr [<assembly_name>]`

List the global assembly cache with all traced references.

`/cdl`

Deletes the contents of the download cache

`/ldl`

Lists the contents of the download cache

`/?`

Displays a detailed help screen

Options:

`/r <reference_scheme> <reference_id> <description>`

Specifies a traced reference to install (`/i`, `/il`) or uninstall (`/u`, `/ul`).

`/f`

Forces reinstall of an assembly.

`/nologo`

Suppresses display of the logo banner

`/silent`

Suppresses display of all output

همانطور که می‌بینید در اجرای GACUtil.exe با تعیین سوییچ **/i** می‌توانید یک اسمبلی را در GAC نصب کرده و با سوییچ **/u** یک اسمبلی را از GAC حذف کنید. توجه کنید که نمی‌توانید یک اسمبلی ضعیف‌نام را در GAC قرار دهید. اگر نام فایل یک اسمبلی ضعیف‌نام را به GACUtil.exe بدهید، پیام خطای زیر را نشان می‌دهد:

“Failure adding assembly to the cache: Attempt to install an assembly without a strong name.”

نکته به صورت پیش فرض، GAC فقط توسط کاربری از گروه Windows Administrators قابل دستکاری است. GACUtil.exe نمی-واند یک اسمبلی را نصب یا حذف کند اگر کاربری که این ابزار را اجرا کرده، عضوی از این گروه نباشد.

استفاده از سویچ **/i** در ابزار GACUtil.exe برای تست‌های برنامه‌نویس بسیار رایج است. اگر از GACUtil.exe برای نصب یک اسمبلی در محیط ساخت استفاده می‌کنید توصیه می‌شود که سویچ **/r** را همراه با سویچ **/i** یا **/u** برای نصب یا حذف اسمبلی به کار ببرید. سویچ **/r** اسمبلی را با موتور نصب و حذف ویندوز همراه می‌کند. این کار به سیستم می‌گوید کدام برنامه به اسمبلی نیاز دارد و سپس برنامه و اسمبلی را به هم گره می‌زند.

نکته اگر یک اسمبلی قوی‌نام در یک فایل کابینت (.cab) بسته‌بندی یا به نحوی فشرده شود، فایل‌های اسمبلی باید قبل از آنکه GACUtil.exe آن‌ها را در GAC نصب کنید، از حالت فشرده در آمده و در فایل (های) موقتی ذخیره شده باشند. به محض آنکه فایل‌های اسمبلی نصب شدند، فایل‌های موقتی می‌توانند حذف شوند.

ابزار GACUtil.exe همراه با پک قابل توزیع دات‌نت به کاربر نهایی عرضه نمی‌شود. اگر برنامه‌ی شما شامل اسمبلی‌هایی است که باید در GAC نصب شوند می‌توانید از Windows Installer (MSI) استفاده کنید چرا که MSI تنها ابزاری است که حتما در ماشین کاربر نهایی وجود داشته و قادر به نصب اسمبلی‌ها در GAC است.

مهم نصب سراسری فایل‌های اسمبلی در GAC، یک نوع ثبت اسمبلی است و این در حالیست که رجیستری ویندوز به هیچ عنوان تحت تاثیر قرار نمی‌گیرد. نصب اسمبلی‌ها در GAC، هدف نصب، تهیه پشتیبان، بازگردانی، انتقال و حذف برنامه به صورت آسان را از بین می‌برد. پس توصیه می‌شود هر جا ممکن است از نصب سراسری خودداری کرده و از نصب خصوصی استفاده کنید.

هدف از "ثبت" یک اسمبلی در GAC چیست؟ خوب، فرض کنید دو شرکت هر کدام یک اسمبلی OurLibrary که از یک فایل OurLibrary.dll تشکیل شده است را تولید می‌کنند. واضح است، هر دوی این فایل‌ها نمی‌توانند در یک زیردایرکتوری جای گیرند چرا که اسمبلی دومی بر روی اسمبلی اولی نوشته می‌شود و مطمئناً تعدادی برنامه از کار خواهد افتاد. وقتی یک اسمبلی را در GAC نصب می‌کنید، زیردایرکتوری‌های اختصاصی در دایرکتوری C:\Windows\Assembly ایجاد شده و فایل‌های اسمبلی در یکی از این زیردایرکتوری‌ها کپی می‌شوند.

معمولاً، کسی زیردایرکتوری‌های GAC را بررسی نمی‌کند، پس ساختار GAC برای شما اهمیتی ندارد. تا زمانی که ابزارها و CLR این ساختار را می‌شناسند، همه چیز خوب است.

ساخت یک اسمبلی که به یک اسمبلی قوی نام ارجاع می‌کند

هر وقت شما یک اسمبلی می‌سازید، اسمبلی به دیگر اسمبلی‌های قوی‌نام ارجاع می‌کند. این درست است چون System.Object در MSCorLib.dll تعریف شده و یک اسمبلی قوی‌نام است. به هر حال، این احتمال هست که یک اسمبلی به نوع‌هایی در دیگر اسمبلی‌های قوی‌نام که توسط مایکروسافت یا دیگر شرکت‌ها ساخته شده است، ارجاع کند. در فصل ۲، نشان دادم چگونه از سویچ **/reference** در CSC.exe برای تعیین اسمبلی‌های ارجاعی استفاده کنید. اگر نام فایل، مسیر کاملی باشد، CSC.exe فایل مشخص شده را بارگذاری کرده و از اطلاعات متادیتای آن برای ساخت اسمبلی استفاده می‌کند. همانطور که در فصل ۲ مطرح شد، اگر نام فایل را بدون مسیر کامل تعیین کنید، CSC.exe سعی می‌کند اسمبلی را به ترتیب با جستجو در دایرکتوری‌های زیر پیدا کند:

۱. دایرکتوری کاری
۲. دایرکتوری حاوی فایل CSC.exe. این دایرکتوری حاوی DLL های CLR نیز است.
۳. هر دایرکتوری که با سویچ **/lib** تعیین کرده باشید.
۴. هر دایرکتوری که با متغیر محلی LIB تعیین شده است.

پس اگر شما یک اسمبلی می‌سازید که به System.Drawing.dll که توسط مایکروسافت ساخته شده، ارجاع می‌کند، باید سویچ **reference:System.Drawing.dll** را هنگام استفاده از CSC.exe تعیین کنید. کامپایلر دایرکتوری‌های مذکور را جستجو می‌کند و فایل

System.Drawing.dll را در پوشه‌ی حاوی CSC.exe پیدا می‌کند که این دایرکتوری حاوی DLL های نسخه‌ای از CLR که کامپایلر با آن کار می‌کند، نیز هست. اگرچه این دایرکتوری مکانی بود که اسمبلی در زمان کامپایل در آنجا یافت شد، اما اسمبلی در زمان اجرا از این دایرکتوری بارگذاری نمی‌شود.

شما می‌دانید که وقتی دانت فریمورک را نصب می‌کنید، در حقیقت دو کپی از اسمبلی‌های میکروسافت نصب می‌شوند. یک مجموعه در دایرکتوری کامپایلر/CLR نصب می‌شود و دیگری در یک زیردایرکتوری از GAC نصب می‌شود. فایل‌های موجود در دایرکتوری کامپایلر/CLR برای آن است که بتوانید به راحتی اسمبلی خود را بسازید، در حالیکه کپی‌های موجود در GAC برای بارگذاری هنگام اجرا هستند.

دلیل آنکه CSC.exe برای اسمبلی‌های ارجاعی، در GAC جستجو نمی‌کند آن است که شما باید مسیر فایل اسمبلی را بدانید ولی ساختار GAC مستند-سازی نشده است. به جای آن، CSC.exe به شما اجازه می‌دهد که یک رشته با ظاهر بهتر مثل

“system.Drawing, Version=v4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a”

را تعیین کنید. هر دوی این راه حل‌ها بدتر از دوبار کپی شدن فایل‌ها در هارد دیسک کاربر تلقی می‌شوند.

نکته هنگام ساخت یک اسمبلی، ممکن است بخواهید به یک اسمبلی که دارای هر دو نسخه‌ی x86 و x64 است، ارجاع کنید. خوشبختانه، زیردایرکتوری‌های GAC می‌توانند نسخه‌های x86 و x64 از یک اسمبلی را نگهداری کنند. اما چون این اسمبلی‌ها دارای نام یکسان هستند نمی‌توانید نسخه‌های مختلف از این اسمبلی‌ها را در دایرکتوری کامپایلر/CLR نگه دارید. هر چند این مهم نیست. وقتی دانت فریمورک را بر روی یک ماشین نصب می‌کنید، نسخه x86، x64 یا IA64 از اسمبلی‌ها در دایرکتوری کامپایلر/CLR نصب می‌شوند. هنگام ساخت یک اسمبلی، هر نسخه‌ای که بخواهید را می‌توانید ارجاع دهید چرا که تمام نسخه‌ها متادیتای یکسانی داشته و فقط کد آن‌ها متفاوت است. در زمان اجرا، نسخه صحیح اسمبلی از GAC بارگذاری می‌شود. در انتهای این فصل توضیح می‌دهم که CLR چگونه تعیین می‌کند اسمبلی را در زمان اجرا از کجا بارگذاری کند.

اسمبلی‌های قوی نام ضد نفوذ هستند

امضاء کردن یک اسمبلی با یک کلید خصوصی این اطمینان را حاصل می‌کند که دارنده‌ی کلید عمومی متناظر، اسمبلی را تولید کرده است. وقتی اسمبلی در GAC نصب می‌شود، سیستم، محتویات فایل حاوی مانیفست را هش کرده و با مقدار هش امضای دیجیتالی RSA که درون فایل PE جاسازی شده است (البته بعد از برداشتن امضاء با کلید عمومی) مقایسه می‌کند. اگر مقادیر یکسان بود، محتویات فایل تغییر نکرده است و شما می‌دانید که کلید عمومی در دست شما با کلید خصوصی سازنده مطابقت دارد. به علاوه، سیستم محتویات دیگر فایل‌های اسمبلی را هش کرده و با مقادیر ذخیره شده در جدول FileDef از فایل مانیفست مقایسه می‌کند. اگر هر یک از مقادیر یکسان نباشند، حداقل یکی از فایل‌های اسمبلی تخریب و دستکاری شده است و اسمبلی نمی‌تواند در GAC نصب شود.

مهم این مکانیزم اطمینان حاصل می‌کند که محتویات یک فایل تغییر نکرده باشد. مکانیزم به شما اجازه نمی‌دهد که سازنده را تعیین کنید مگر آنکه یقین داشته باشید که کلید عمومی در دست شما را سازنده تولید کرده است و مطمئن باشید که کلید خصوصی سازنده هویتش را به کمک تکنولوژی Authenticode میکروسافت با اسمبلی همراه کند.

وقتی یک برنامه نیاز به اتصال به یک اسمبلی دارد، CLR از مشخصات اسمبلی ارجاعی (نام، نسخه، فرهنگ و کلید عمومی) برای یافتن اسمبلی در GAC استفاده می‌کند. اگر اسمبلی یافت شد، زیردایرکتوری حاوی آن را بر می‌گرداند و فایل حاوی مانیفست بارگذاری می‌شود. یافتن اسمبلی به این روش، فراخوانی کننده‌ی اسمبلی را مطمئن می‌کند که اسمبلی بارگذاری شده در زمان اجرا و اسمبلی‌ای که کد با آن کامپایل شده است هر دو از یک سازنده می‌باشند. این اطمینان به این دلیل ممکن است که نشانه کلید عمومی در جدول AssemblyRef از اسمبلی ارجاع کننده با کلید عمومی در جدول AssemblyDef از اسمبلی ارجاع شونده، تطابق دارد. اگر اسمبلی ارجاعی در GAC وجود نداشته باشد، CLR در دایرکتوری اصلی برنامه و سپس هر دایرکتوری مشخص شده در فایل تنظیمات برنامه جستجو می‌کند و سپس اگر اسمبلی توسط MSI نصب شده بود، CLR از MSI درخواست می‌کند که اسمبلی را بیابد. اگر با هیچکدام از این روش‌ها اسمبلی یافت نشد، عملیات اتصال شکست می‌خورد و یک اکسپشن System.IO.FileNotFoundException تولید می‌شود.

وقتی فایل‌های اسمبلی قوی‌نام از مکانی جز **GAC** (مثل دایرکتوری اصلی برنامه یا عنصر **codeBase** در فایل تنظیمات) بارگذاری می‌شوند، **CLR** مقادیر هش را هنگام بارگذاری مقایسه می‌کند. به بیان دیگر هر بار که برنامه اجرا شده و اسمبلی بارگذاری شود، یک هش از فایل تولید می‌شود. این کاهش کارایی در ازای آن است که مطمئن شوید محتویات فایل تغییر نکرده است. وقتی **CLR**، عدم تساوی مقادیر هش در زمان اجرا را ببیند، یک اکسپشن **System.IO.FileLoadException** تولید می‌کند.

نکته وقتی یک اسمبلی قوی نام در **GAC** نصب می‌شود، سیستم اطمینان حاصل می‌کند محتویات فایل حاوی مانیفست تغییر نکرده باشد. این بررسی فقط یک بار در زمان نصب انجام می‌شود. به علاوه، جهت افزایش کارایی، اگر یک اسمبلی کاملاً مورد اطمینان باشد و در یک **AppDomain** کاملاً مطمئن بارگذاری شود، **CLR** دیگر بررسی نمی‌کند که این اسمبلی قوی نام دچار تغییر شده است یا خیر. به بیان دیگر، وقتی یک اسمبلی از مکانی جز **GAC** بارگذاری می‌شود، **CLR** فایل مانیفست اسمبلی را بررسی می‌کند تا از عدم تغییر آن اطمینان حاصل کند که هر بار هنگام بارگذاری منجر به کاهش سرعت نیز می‌شود.

امضای تاخیری

در آغاز فصل، بحث کردم که چگونه ابزار **SN.exe** جفت کلیدهای عمومی/خصوصی تولید می‌کند. این ابزار با فراخوانی‌هایی به **Crypto API** در ویندوز، کلیدها را تولید می‌کند. این کلیدها می‌توانند در فایل یا دستگاه‌های حافظه ذخیره شوند. برای نمونه سازمان‌های بزرگ (مثل مایکروسافت) کلید خصوصی را در سخت افزارهایی قفل شده نگهداری می‌کنند که فقط تعداد افراد کمی در شرکت به کلید خصوصی دسترسی دارند. این احتیاط مانع از لو رفتن کلید خصوصی می‌شود. کلید عمومی، خوب، عمومی است و آزادانه توزیع می‌شود.

وقتی شما آماده‌ی بسته‌بندی اسمبلی قوی‌نام خود هستید، باید از کلید خصوصی ایمن برای امضای آن استفاده کنید. اما، هنگام توسعه و آزمایش اسمبلی، دسترسی به کلید خصوصی ایمن، دردسر دارد. به همین علت، دانت فریمورک از امضای تاخیری **delayed signing** که گاهی امضای جزئی **partial signing** نیز اطلاق می‌شود، پشتیبانی می‌کند. امضای تاخیری به شما اجازه می‌دهد که یک اسمبلی را تنها با کلید عمومی شرکت خود بسازید، کلید خصوصی مورد نیاز نیست. استفاده از کلید عمومی به اسمبلی‌هایی که اسمبلی شما را ارجاع می‌کنند، اجازه می‌دهد مقدار صحیح کلید عمومی را در ورودی-های جدول **AssemblyRef** مانیفست جاسازی کنند. همچنین اجازه می‌دهد که اسمبلی به درستی در **GAC** قرار بگیرد. اگر شما اسمبلی را با کلید خصوصی شرکتان امضاء نکنید، تمام امنیت ضد نفوذ بودن فایل‌های اسمبلی را از دست می‌دهید چرا که فایل‌های اسمبلی دیگر هش نشده و امضای دیجیتالی در فایل جاسازی نمی‌شود. این از دست دادن محافظت، مسئله‌ای نیست، چون شما فقط هنگام توسعه و ساخت اسمبلی و نه هنگام بسته‌بندی و نصب، از امضای تاخیری استفاده کرده اید.

اساساً، شما مقدار کلید عمومی شرکت را در یک فایل ذخیره کرده و فایل را به هر ابزاری که با آن اسمبلی را می‌سازید، ارسال می‌کنید. (همانطور که در اوایل فصل نشان دادم، می‌توانید با سوییچ **-p** در برنامه **SN.exe**، کلید عمومی را از جفت کلید عمومی/خصوصی استخراج کنید). همچنین باید به ابزار بگویید که اسمبلی باید به صورت تاخیری امضاء شود، به این معنی که شما یک کلید خصوصی ارائه نکرده‌اید. برای کامپایلر سی‌شارپ، این کار را با سوییچ **/delaysign** انجام می‌دهید. در ویژوال استودیو، با نمایش **Properties** پروژه و کلیک بر تب **Signing** و انتخاب گزینه **Delay Sign Only** این کار را انجام می‌دهید. اگر از ابزار **AL.exe** استفاده می‌کنید، سوییچ **/delay[sign]** را تعیین کنید.

وقتی کامپایلر یا **AL.exe** می‌بیند شما قصد امضای تاخیری یک اسمبلی را دارید، ورودی **AssemblyDef** مانیفست اسمبلی که حاوی کلید عمومی اسمبلی خواهد بود را پر می‌کند. مجدداً، وجود کلید عمومی اجازه می‌دهد که اسمبلی را در **GAC** قرار بگیرد. همچنین به شما اجازه می‌دهد که دیگر اسمبلی‌هایی که این اسمبلی را ارجاع می‌کنند، نیز بسازید. اسمبلی‌های ارجاع کننده، کلید عمومی صحیح را در ورودی‌های جدول **AssemblyRef** خود، خواهند داشت. هنگام ساخت اسمبلی، فضایی در فایل **PE** برای امضای دیجیتالی **RSA** خالی باقی می‌ماند. (ابزار می‌تواند از روی اندازه‌ی کلید عمومی مقدار فضای مورد نیاز را تعیین کند). توجه کنید که در این زمان محتویات فایل‌ها هش نمی‌شوند.

در این لحظه، اسمبلی تولیدی امضای معتبری ندارد. تلاش برای نصب اسمبلی در **GAC** با شکست مواجه خواهد شد چرا که هش محتویات فایل‌ها تولید نشده است و فایل تغییر یافته به نظر می‌رسد. در هر ماشین که اسمبلی باید در **GAC** نصب شود، شما باید مانع بررسی یکپارچگی فایل‌ها توسط سیستم شوید. برای اینکار، از سوییچ **-Vr** در **SN.exe** استفاده کنید. اجرای **SN.exe** با این سوییچ به **CLR** می‌گوید که هنگام بارگذاری در زمان اجرا، مقادیر هش هر یک از فایل‌های اسمبلی را نادیده بگیرد. سوییچ **-Vr** هویت اسمبلی را در زیر کلید زیر از رجیستری اضافه می‌کند:

```
HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\StrongName\Verification
```

مهم وقتی از ابزاری برای دستکاری رجیستری استفاده می کنید، مطمئن شوید که از نسخه ۶۴بیتی آن ابزار در یک ماشین ۶۴بیتی استفاده می کنید. به صورت پیش فرض، ابزارهای ۳۲بیتی x86 در

C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools و ابزارهای ۶۴بیتی x64 در C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools\x64 قرار دارند.

وقتی کار ساخت و آزمایش برنامه تمام شد، نیاز دارید اسمبلی را به صورت رسمی امضاء کنید تا بتوانید آن را بسته بندی و نصب کنید. برای امضای اسمبلی، ابزار SN.exe را مجدداً استفاده کنید و این بار سوئیچ **-R** را با نام فایل حاوی کلید خصوصی، تعیین کنید. سوئیچ **-R** باعث می شود SN.exe محتویات فایل را هش کرده، آن را با کلید خصوصی امضاء کرده و در فضایی که قبلاً در فایل برای امضای دیجیتالی RSA رزرو شده است، جاسازی کند. بعد از این مرحله، اسمبلی کاملاً امضاء شده را نصب کنید. فراموش نکنید بر روی ماشین های آزمایش و توسعه برنامه، با سوئیچ **-Vu** یا **-Vx** از برنامه ی SN.exe، عملیات بازبینی (verification) را به حالت قبلی (فعال) بازگردانید. لیست زیر مراحل بحث شده در این بخش، جهت استفاده از تکنیک امضای تاخیری را خلاصه می کند:

۱. هنگام توسعه اسمبلی، یک فایل که فقط حاوی کلید عمومی شرکتتان است تهیه کنید و اسمبلی را با سوئیچ های **/keyfile** و **/delaysign** کامپایل کنید.

```
CSC /keyfile:MyCompany.PublicKey /delaysign MyAssembly.cs
```

۲. بعد از ساخت اسمبلی کد زیر را اجرا کنید تا CLR بدون مقایسه و هش گرفتن، به بایتهای اسمبلی اعتماد کند. این کار اجازه می دهد که اسمبلی را (اگر مایلید) در **GAC** نصب کنید. اکنون شما می توانید اسمبلی های دیگری بسازید که به این اسمبلی ارجاع می کنند و می توانید بدین گونه اسمبلی را تست کنید. دقت کنید شما مجبورید خط فرمان را فقط یکبار در هر ماشین اجرا کنید و نیاز نیست برای هر بار ساخت اسمبلی آن را اجرا کنید.

```
SN.exe -Vr MyAssembly.dll
```

۳. وقتی آماده ی بسته بندی و نصب اسمبلی شدید، کلید خصوصی شرکت را بدست آورده و خط زیر را اجرا کنید. اگر مایلید می توانید این نسخه ی جدید را در **GAC** نصب کنید اما تا قبل از انجام مرحله ۴، از نصب آن در **GAC** خودداری کنید.

```
SN.exe -R MyAssembly.dll MyCompany.PrivateKey
```

۴. برای آزمایش در شرایط واقعی، بازبینی را مثل قبل با اجرای خط زیر فعال کنید.

```
SN.exe -Vu MyAssembly.dll
```

در آغاز این بخش، اشاره کردم که سازمان ها چگونه جفت کلیدهای خود را در سخت افزارهایی مثل کارت هوشمند نگه می دارند. برای ایمن نگاه داشتن این کلیدها، باید اطمینان حاصل کنید که کلیدها هرگز در فایلی بر روی دیسک ذخیره نشوند. فراهم کننده های سرویس های رمزنگاری (CSP) نگهدارنده هایی ارائه می کنند که مکان کلیدها را مخفی می کند. برای نمونه مایکروسافت، از یک CSP استفاده می کند که هنگام دسترسی به نگهدارنده، کلید خصوصی را از یک سخت افزار بدست می آورد.

اگر کلید عمومی/خصوصی شما در یک نگهدارنده ی CSP است، باید سوئیچ های متفاوتی را برای برنامه های CSC.exe، AL.exe و SN.exe تعیین کنید: هنگام کامپایل (CSC.exe) سوئیچ **/keycontainer** را به جای **/keyfile** تعیین کنید؛ هنگام لینک کردن (AL.exe)، سوئیچ **/keyname** را به جای **/keyfile** تعیین کنید و هنگام استفاده از برنامه (SN.exe) جهت افزودن کلید خصوصی به یک اسمبلی با امضای تاخیری، سوئیچ **-Rc** را به جای **-R** تعیین کنید. SN.exe سوئیچ های دیگری هم برای کار با یک CSP فراهم می کند.

مهم امضای تاخیری هنگامیکه بخواهید عملیات دیگری بر روی اسمبلی قبل از بسته بندی انجام دهید، نیز سودمند است. برای نمونه شاید شما بخواهید یک obfuscator بر روی اسمبلی اجرا کنید. هنگامیکه اسمبلی کاملاً امضا شده باشد شما نمی توانید یک اسمبلی را (بوسیله obfuscator) درهم بریزید چون مقدار هش اشتباه خواهد شد. پس اگر می خواهید یک فایل اسمبلی را در هم بریزید و یا هر عملیات پس از- ساخت دیگری روی آن انجام دهید، مجبورید از امضای تاخیری استفاده کنید، عملیات پس از-ساخت را انجام داده و سپس SN.exe را با سوئیچ **-R** یا **-Rc** اجرا کنید تا فرآیند امضای اسمبلی و تمام هش های آن کامل شود.

نصب اسمبلی های قوی نام به صورت خصوصی

نصب اسمبلی های در GAC فواید زیادی دارد. GAC برنامه ها را قادر می سازد تا اسمبلی ها را به اشتراک گذاشته در نتیجه فضای کمتری بر روی دیسک اشغال می شود. به علاوه، نصب یک نسخه جدید از اسمبلی در GAC ساده است و می توان از طریق اعمال سیاست های سازنده (که بعداً در این فصل بحث خواهد شد) تمام برنامه ها را مجبور به استفاده از نسخه جدید کرد. GAC همچنین مدیریت نسخه های مختلف اسمبلی در کنار هم را ارائه می کند. اما به هر حال، GAC همیشه به نحوی محافظت می شود که فقط یک مدیر می تواند یک اسمبلی در آن نصب کند. در ضمن، نصب در GAC، داستان نصب ساده با کپی کردن را از بین می برد.

اگرچه اسمبلی های قوی نام می توانند در GAC نصب کرده شوند اما همیشه این کار لازم نیست. در واقع توصیه می شود تنها اگر اسمبلی میان تعداد زیادی برنامه مشترک است آن را در GAC نصب کنید. اگر اسمبلی به اشتراک گذاشته نمی شود، باید به صورت خصوصی نصب شود. نصب خصوصی اجرای نصب ساده با کپی کردن را فراهم کرده و بهتر برنامه و اسمبلی های را ایزوله می کند. همچنین قرار نیست GAC تبدیل به C:\Windows\System32 جدید شود و هر فایل مشترکی در آنجا قرار بگیرد. علت آن است که نسخه های جدید اسمبلی ها بر روی همدیگر نوشته نمی شوند، آن ها در کنار همدیگر نصب شده و فضای دیسک را اشغال می کنند. علاوه بر نصب یک اسمبلی قوی نام در GAC و یا نصب خصوصی آن، می تواند اسمبلی را در دایرکتوری دلخواهی که چند برنامه از آن آگاهند، قرار دهید. برای نمونه، شاید شما سه برنامه تولید کنید که می خواهند یک اسمبلی قوی نام را به اشتراک بگذارند. برای نصب، می توانید ۳ دایرکتوری درست کنید: یک دایرکتوری برای هر برنامه و یک دایرکتوری اضافی برای اسمبلی ای که به اشتراک می گذارید. وقتی هر برنامه را در دایرکتوری خود نصب می کنید، یک فایل XML تنظیمات نیز بسازید و عنصر **codeBase** آن را با مسیر اسمبلی اشتراکی پر کنید. حال در زمان اجرا، CLR می داند که باید در دایرکتوری اصلی اسمبلی قوی نام، به دنبال اسمبلی بگردد. البته، این تکنیک به ندرت استفاده می شود و تا حدی توصیه نمی شود چون هیچ تک برنامه ای زمان حذف فایل های اسمبلی را کنترل نمی کند.

نکته عنصر **codeBase** در فایل تنظیمات در واقع یک URL دریافت می کند. URL می توانید به هر دایرکتوری در ماشین کاربر یا آدرس وب ارجاع کند. در مورد آدرس وب، CLR به صورت خودکار فایل را دانلود و در کش دانلود کاربر (یک زیر دایرکتوری تحت C:\Users\UserName\Local Settings\Application Data\AssemblyCache\CurrentUser\CodeBase\URL که Username نام کاربر جاری ویندوز است) ذخیره می کند. وقتی اسمبلی در آینده ارجاع می شود، CLR برچسب زمانی فایل دانلود شده با فایل موجود در URL را بررسی می کند، اگر فایل موجود در URL جدیدتر بود، CLR نسخه جدید را دانلود و آن را بارگذاری می کند. اگر فایل دانلود شده جدیدتر بود، CLR این فایل را بارگذاری کرده و فایل را مجدداً دانلود نمی کند (بهبود کارایی). یک نمونه استفاده از عنصر **codeBase** در ادامه فصل آمده است.

چگونه CLR ارجاع به نوع ها را تحلیل می کند

در آغاز فصل ۲، سورس کد زیر را دیدیم:

```
public sealed class Program {
    public static void Main() {
        System.Console.WriteLine("Hi");
    }
}
```

این کد کامپایل شده و به یک اسمبلی به نام Program.exe تبدیل می شود. وقتی این برنامه را اجرا می کنید، CLR بارگذاری و مقداردهی اولیه می شود. سپس CLR، هدر CLR اسمبلی را می خواند و به دنبال MethodDefToken که نقطه شروع برنامه (Main) را نشان می دهد، می گردد. از جدول متادیتای MethodDef، آفستی درون فایل برای کد IL متد بدست می آید و کد IL توسط کامپایلر JIT به کد اصلی تبدیل می شود که کد از لحاظ امنیت نوع نیز بررسی می شود. سپس کد اصلی شروع به اجرا می کند. در زیر کد IL متد Main را می بینید. برای بدست آوردن این کد، ILDasm را اجرا کردم و از منوی View گزینه Show Bytes را انتخاب کردم و سپس متد Main را دابل کلیک کردم.

```
// SIG: 00 00 01
{
    .entrypoint
    // Method begins at RVA 0x2050
```



```
// Code size 11 (0xb)
.maxstack 8
IL_0000: /* 72 | (70)000001 */
ldstr "Hi"
IL_0005: /* 28 | (0A)000003 */
call void [mscorlib]System.Console::WriteLine(string)
IL_000a: /* 2A |          */
ret
} // end of method Program::Main
```

وقتی این کد کامپایل JIT می‌شود، CLR تمام ارجاع‌ها به نوع‌ها و اعضاها را پیدا می‌کند و اسمبلی تعریف‌کننده آن‌ها را (در صورتی که قبلاً بارگذاری نشده باشند) بارگذاری می‌کند. همانطور که می‌بینید، کد IL فوق دارای یک ارجاع به **System.Console.WriteLine** است. به خصوص، دستور **call** در IL به نشانه‌ی متادیتای **0A000003** ارجاع می‌کند. این نشانه، ورودی ۳ در جدول متادیتای **MemberRef** (جدول 0A) را نشان می‌دهد. CLR این ورودی **MemberRef** را جستجو می‌کند و می‌بیند که یکی از فیلدهایش به یک ورودی در جدول **TypeRef** (نوع **System.Console**) ارجاع می‌کند. از جدول **TypeRef**، CLR به یک ورودی **AssemblyRef**:

```
"mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

هدایت می‌شود. در این لحظه، CLR می‌داند آن به کدام اسمبلی نیاز دارد. حال CLR اسمبلی را یافته و بارگذاری می‌کند.

هنگام تحلیل یک نوع ارجاعی، CLR نوع را در یکی از سه مکان زیر می‌تواند پیدا کند:

- **همان فایل** دسترسی به یک نوع که در همان فایلی است که در زمان کامپایل تعیین شده است. (گاهی با عنوان اتصال (انقیاد) اولیه **early bound** اطلاق می‌شود). نوع، از فایل بارگذاری می‌شود و اجرا ادامه می‌یابد.
- **فایل متفاوت، همان اسمبلی** CLR مطمئن می‌شود که فایل ارجاع داده شده در واقع در جدول **FileRef** اسمبلی از مانیفست اسمبلی جاری می‌باشد. CLR سپس در دایرکتوری‌ای که فایل مانیفست اسمبلی بارگذاری شده است نگاه می‌کند. فایل بارگذاری می‌شود، مقدار هش آن بررسی می‌شود، عضو نوع یافت می‌شود و برنامه ادامه می‌یابد.
- **فایل متفاوت، اسمبلی متفاوت** وقتی یک نوع ارجاعی در فایل اسمبلی متفاوتی است، CLR، فایل حاوی مانیفست اسمبلی ارجاعی را بارگذاری می‌کند. اگر این فایل شامل نوع نبود، فایل صحیح بارگذاری می‌شود. عضو نوع یافت می‌شود و اجرا ادامه می‌یابد.

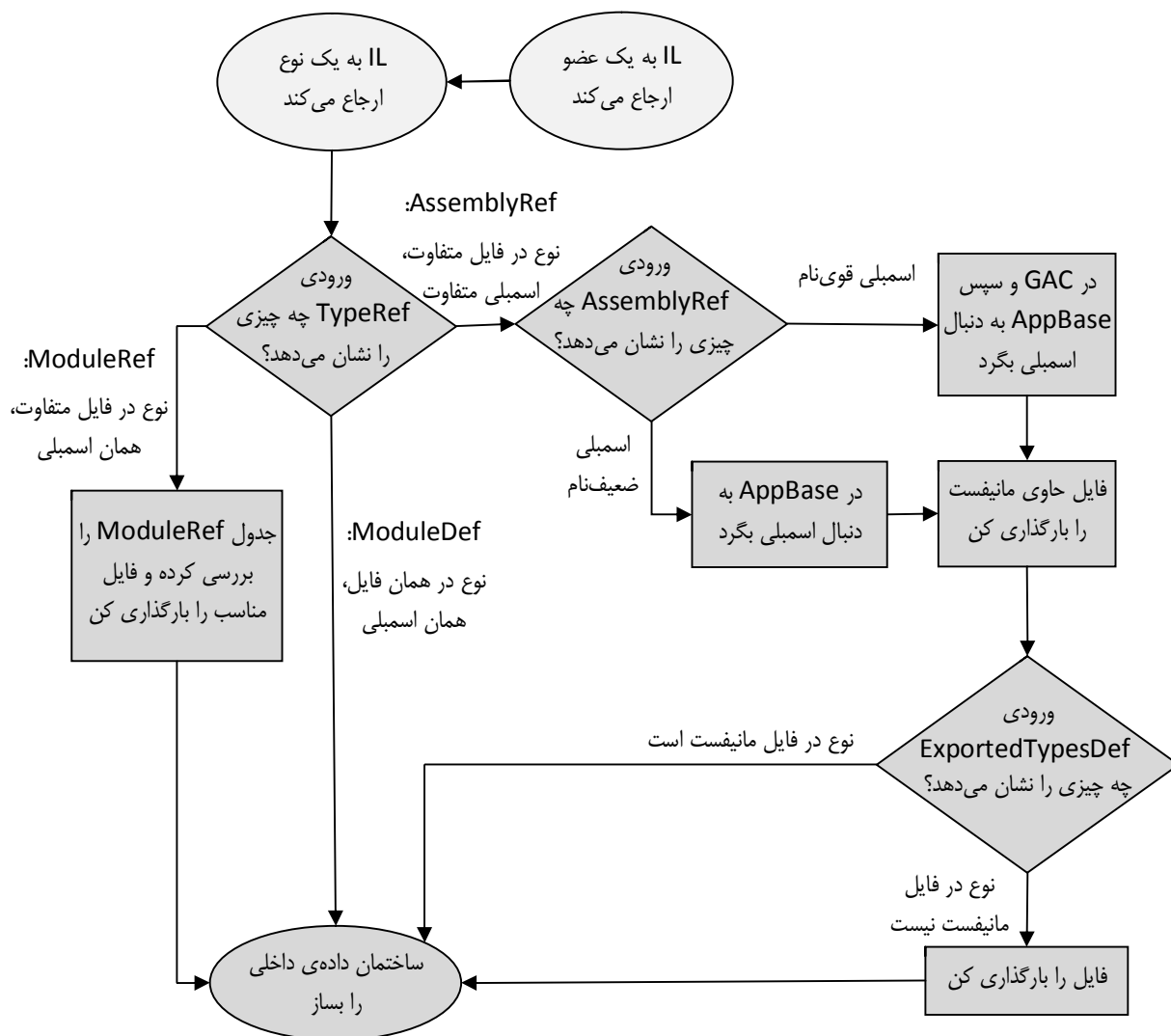
نکته جدول‌های متادیتای **ModuleDef**، **ModuleRef** و **FileDef** به فایل‌ها با نام و پسوندشان اشاره می‌کنند. اما جدول متادیتای **AssemblyRef** فقط با نام و بدون پسوند به اسمبلی‌ها اشاره می‌کند. هنگام اتصال یک اسمبلی، سیستم به صورت خودکار پسوند **.dll** و **.exe** را برای یافتن اسمبلی همانطور که در بخش "کنترل‌های مدیریتی ساده (تنظیمات)" از فصل ۲ بدان اشاره شد، به نام اسمبلی می‌افزاید.

اگر هر خطایی هنگام تحلیل ارجاع به یک نوع رخ دهد – فایل یافت نشود، فایل بارگذاری نشود، عدم تطابق مقادیر هش و... – یک اکسپشن مناسب تولید تولید خواهد شد.

نکته اگر بخواهید، کد شما می‌تواند متدهای کالبد را با رویدادهای **AssemblyResolve** و **ReflectionOnlyAssemblyResolve** و **TypeResolve** از **System.AppDomain** ثبت کند. در متدهای کالبد خود، کدی را اجرا کنید که مسئله اتصال را حل کرده و بدون تولید اکسپشن اجازه دهد اجرای برنامه ادامه یابد.

در مثال قبلی، CLR تعیین کرد که **System.Console** در فایلی غیر از فراخوانی‌کننده، قرار دارد. CLR باید به دنبال اسمبلی بگردد و فایل PE حاوی مانیفست اسمبلی را بارگذاری کند. سپس برای تعیین فایل حاوی نوع، مانیفست اسکن می‌شود. اگر فایل مانیفست شامل نوع ارجاعی بود، خوب همه چیز حل است. اگر نوع در فایل‌های دیگر اسمبلی بود، CLR فایل دیگر را بارگذاری کرده و متادیتای آن را برای یافتن نوع اسکن می‌کند. سپس CLR ساختمان داده-ی داخلی را برای بیان نوع می‌سازد و کامپایلر JIT کامپایلر **Main** را کامل می‌کند. سرانجام، متد **Main** می‌تواند شروع به اجرا شدن کند.

شکل ۲-۳ نشان می‌دهد چگونه اتصال نوع رخ می‌دهد.



نکته: اگر عملیاتی شکست بخورد، یک اکسپشن مناسب تولید می شود.

شکل ۲-۳ فلوجارت نشان می دهد که چگونه، با کد IL ای که به یک عضو یا یک نوع ارجاع می کند، CLR از متادیتا برای یافتن فایل صحیح اسمبلی و تعریف نوع استفاده می کند.

مهم صریحا بگویم که مثال توضیح داده شده ۱۰۰ درصد درست نیست. برای ارجاع به متدها و نوعهایی که با داتنت فریمورک عرضه نمی شوند، مطلب درست است. اما اسمبلی های داتنت فریمورک (شامل MSCorLib.dll) به نسخه CLR در حال اجرا وابسته اند. هر اسمبلی که به اسمبلی های داتنت فریمورک ارجاع می کند، همیشه به نسخه ای از اسمبلی که با نسخه CLR همخوانی دارد، متصل می شود. این یگانگی unification نامیده می شود و مایکروسافت چون تمام اسمبلی های داتنت فریمورک را با یک نسخه از CLR تست می کند، این کار را انجام می دهد، بنابراین این کار باعث می شود برنامه ها به درستی کار کنند.

بنابراین در مثال قبل، متد WriteLine از System.Console به هر نسخه ای از MSCorLib.dll که با نسخه CLR همخوانی دارد متصل می شود؛ بدون توجه به نسخه MSCorLib.dll که در جدول متادیتای AssemblyRef ارجاع شده است.

این قصه یک خم دیگر هم دارد: برای CLR، تمام اسمبلی ها با نام، نسخه، فرهنگ و کلید عمومی شناخته می شوند. اما GAC اسمبلی ها را با نام، نسخه، فرهنگ، کلید عمومی و معماری پردازنده می شناسد. هنگام جستجوی اسمبلی در GAC، CLR تعیین می کند برنامه در چه نوع پردازش ای در حال اجراست:

۳۲ x86 بیتی (احتمالاً توسط تکنولوژی WoW64)، ۶۴ x64 بیتی یا IA64 ۶۴ بیتی. سپس هنگام جستجوی GAC برای یک اسمبلی، CLR ابتدا به دنبال نسخه‌ای از اسمبلی مطابق با معماری پردازنده می‌گردد. اگر اسمبلی یافت نشد به دنبال نسخه‌ای مستقل از معماری پردازنده خواهد گشت. در این بخش، شما دیدید که چگونه CLR با سیاست پیش فرض خود، یک اسمبلی را پیدا می‌کند. اما، یک مدیر یا سازنده یک اسمبلی می‌تواند این سیاست-ها را تغییر دهد. در دو بخش آتی، چگونگی تغییر سیاست‌های پیش فرض اتصال را خواهیم گفت.

نکته CLR از قابلیت انتقال یک نوع (class، structure، enum، interface یا delegate) از یک اسمبلی به اسمبلی دیگر، پشتیبانی می‌کند. برای نمونه در دات‌نت ۳.۵، کلاس **System.TimeZoneInfo** در اسمبلی **System.Core.dll** تعریف شده است. اما در دات‌نت ۴.۰ مایکروسافت این کلاس را به اسمبلی **mscorlib.dll** انتقال داده است. در حالت معمولی، انتقال یک نوع از یک اسمبلی به اسمبلی دیگر برنامه‌ها را از کار می‌اندازد. هر چند CLR صفت **System.Runtime.CompilerServices.TypeForwardedToAttribute** را ارائه می‌کند که بر اسمبلی اصلی (مثل **System.Core.dll**) اعمال می‌شود. پارامتری که به سازنده این صفت می‌فرستید از نوع **System.Type** است و نوع جدید (که اکنون در **mscorlib.dll** تعریف شده است) را تعیین می‌کند. CLR از این اطلاعات استفاده می‌کند، چون سازنده **TypeForwardedToAttribute** یک **Type** دریافت می‌کند، اسمبلی حاوی این صفت وابسته به اسمبلی جدید که نوع را تعریف می‌کند خواهد بود.

اگر از این ویژگی بهره می‌برید، پس باید صفت **System.Runtime.CompilerServices.TypeForwardedFromAttribute** را بر نوع در اسمبلی جدید اعمال کرده و نام کامل اسمبلی‌ای که نوع را قبلاً تعریف می‌کرده است را به عنوان پارامتر به سازنده صفت ارسال کنید. این صفت اغلب برای ابزارها، برنامه‌های کاربردی و سریالی سازی استفاده می‌شود. چون سازنده **TypeForwardedFromAttribute** یک **String** دریافت می‌کند، اسمبلی حاوی این صفت وابسته به اسمبلی تعریف کننده نوع نیست.

کنترل‌های مدیریتی پیشرفته (تنظیمات)

در بخش "کنترل‌های مدیریتی ساده (تنظیمات)" در فصل ۲، توضیح خلاصه‌ای دادم درباره‌ی اینکه یک مدیر چگونه می‌تواند بر روش جستجو و اتصال اسمبلی توسط CLR اثر بگذارد. در آن بخش، نشان دادم چگونه فایل یک اسمبلی ارجاعی می‌تواند به زیردایرکتوری‌هایی از دایرکتوری اصلی برنامه انتقال یابد و چگونه CLR از فایل تنظیمات برنامه برای یافتن اسمبلی‌های منتقل شده استفاده می‌کند.

با بحث پیرامون صفت **privatePath** از عنصر **probing** در فصل ۲، اکنون درباره‌ی دیگر عناصر فایل XML تنظیمات در این بخش صحبت می‌کنم. در زیر یک فایل XML تنظیمات را می‌بینید:

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="AuxFiles;bin\subdir" />

      <dependentAssembly>

        <assemblyIdentity name="JeffTypes"
          publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

        <bindingRedirect
          oldVersion="1.0.0.0" newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

```

<codeBase version="2.0.0.0"
  href="http://www.wintellect.com/JeffTypes.dll" />
</dependentAssembly>

<dependentAssembly>

  <assemblyIdentity name="TypeLib"
    publicKeyToken="1f2e74e897abbcfe" culture="neutral"/>

  <bindingRedirect
    oldVersion="3.0.0.0-3.5.0.0" newVersion="4.0.0.0" />

  <publisherPolicy apply="no" />
</dependentAssembly>

</assemblyBinding>
</runtime>
</configuration>

```

این فایل XML اطلاعات با ارزشی را به CLR می‌دهد. این اطلاعات به این شرح است:

- **عنصر probing** هنگام پیدا کردن یک اسمبلی ضعیف‌نام، زیردایرکتوری‌های AuxFiles و bin\subdir از زیردایرکتوری اصلی برنامه را بگرد. برای اسمبلی‌های قوی‌نام، CLR در GAC یا URL تعیین شده در عنصر **codeBase** را نگاه می‌کند. تنها اگر هیچ عنصر **codeBase** تعریف نشده باشد، CLR در مسیرهای خصوصی برنامه به دنبال یک اسمبلی قوی‌نام می‌گردد.
 - **اولین عناصر dependentAssembly, assemblyIdentity و bindingRedirect** هنگام تلاش برای یافتن نسخه ۱.۰.۰.۰ از اسمبلی با فرهنگ خنثی JeffTypes که توسط سازمانی صاحب نشانه کلید عمومی 32ab4ba45e0a69a1 تولید شده است، به جای آن، نسخه ۲.۰.۰.۰ از همان اسمبلی را پیدا کن.
 - **عنصر codeBase** هنگام تلاش برای یافتن نسخه ۲.۰.۰.۰ از اسمبلی با فرهنگ خنثی JeffTypes که توسط سازمانی صاحب نشانه کلید عمومی 32ab4ba45e0a69a1 تولید شده است، سعی کن اسمبلی را در این URL: www.Wintellect.com/JeffTypes.dll پیدا کنی. اگرچه در فصل ۲ اشاره نکردم، یک عنصر **codeBase** می‌تواند با یک اسمبلی ضعیف‌نام نیز استفاده شود. در این حالت، شماره نسخه اسمبلی نادیده گرفته می‌شود و باید از عنصر **codeBase** فایل XML حذف شود. همچنین URL موجود در عنصر **codeBase** باید به یک دایرکتوری تحت دایرکتوری اصلی برنامه اشاره کند.
 - **دومین عناصر dependentAssembly, assemblyIdentity, bindingRedirect** هنگام تلاش برای یافتن نسخه ۳.۰.۰.۰ تا ۳.۵.۰.۰ اسمبلی با فرهنگ خنثی TypeLib که توسط سازمان صاحب نشانه کلید عمومی 1f2e74e897abbcfe تولید شده است، به جای آن، نسخه ۴.۰.۰.۰ از همان اسمبلی را پیدا کن.
 - **عنصر publisherPolicy** اگر سازمانی که اسمبلی TypeLib را تولید کرده یک فایل سیاست سازنده (که در بخش بعدی توضیح داده می‌شود) را نصب کرده است، CLR باید این فایل را نادیده بگیرد.
- هنگام کامپایل یک متد، CLR نوع‌ها و اعضای ارجاع داده شده را تعیین می‌کند. با این اطلاعات، CLR با نگاه در جدول AssemblyRef از اسمبلی ارجاع کننده، اسمبلی‌ای که هنگام ساخت اسمبلی فراخوانی کننده، ارجاع شده است را تعیین می‌کند. سپس CLR در فایل تنظیمات برنامه به دنبال اسمبلی و نسخه‌ی آن می‌گردد و هر انتقال شماره نسخه را اعمال کرده و اکنون به دنبال اسمبلی با شماره نسخه (جدید) می‌گردد.
- اگر صفت **apply** از عنصر **publisherPolicy** به **yes** تنظیم شده باشد — یا عنصر وجود نداشته باشد — CLR در GAC به دنبال نسخه جدید اسمبلی می‌گردد و هر انتقال شماره نسخه که سازنده اسمبلی تعیین کرده است را اعمال می‌کند و اکنون CLR به دنبال این نسخه از اسمبلی می‌گردد. درباره سیاست‌های سازنده در بخش بعدی بیشتر صحبت می‌کنم. سرانجام، CLR اسمبلی/نسخه جدید را در فایل Machine.config ماشین جستجو کرده و هر انتقال شماره نسخه را اعمال می‌کند.

در این لحظه، CLR نسخه‌ای از اسمبلی را که باید بارگذاری کند، می‌داند و سعی می‌کند که اسمبلی را از GAC بارگذاری کند. اگر اسمبلی در GAC نباشد و هیچ عنصر **codeBase** موجود نباشد، همانطور که در فصل ۲ گفتم CLR به دنبال اسمبلی می‌گردد. اگر فایل تنظیمات که آخرین انتقال شماره نسخه را انجام می‌دهد شامل عنصر **codeBase** باشد، CLR سعی می‌کند اسمبلی را از URL تعیین شده در عنصر **codeBase** بارگذاری کند.

به کمک این فایل‌های تنظیمات، یک مدیر واقعا می‌تواند کنترل کند که CLR چه اسمبلی‌هایی را بارگذاری می‌کند. اگر یک برنامه دچار خطا شود، مدیر می‌تواند با سازنده اسمبلی تماس بگیرد. سازنده، یک اسمبلی جدید برای مدیر ارسال می‌کند که مدیر نصب کند. به صورت پیش فرض، CLR این اسمبلی جدید را بارگذاری نمی‌کند چون اسمبلی‌های موجود به اسمبلی جدید ارجاع نمی‌کنند. به هر حال، مدیر می‌تواند فایل XML تنظیمات را تغییر دهد تا CLR اسمبلی جدید را بارگذاری کند.

اگر مدیر بخواهد تمام برنامه‌های روی ماشین از اسمبلی جدید استفاده کنند، مدیر می‌تواند فایل **Machine.config** ماشین را تغییر دهد و CLR وقتی یک برنامه به اسمبلی قدیمی ارجاع کند، اسمبلی جدید را بارگذاری می‌کند.

اگر اسمبلی جدید خطای موجود را برطرف نکرد، مدیر می‌تواند خطوط مربوط به انتقال شماره نسخه را از فایل تنظیمات حذف کند. و برنامه همانند قبل رفتار خواهد کرد. این نکته مهم است که سیستم اجازه‌ی استفاده از یک اسمبلی که دقیقا با نسخه اسمبلی در متادیتا تطابق ندارد را می‌دهد. این انعطاف پذیری خیلی سودمند است.

سیاست های کنترلی سازنده

در سناریوی بخش قبلی، سازنده‌ی یک اسمبلی به سادگی نسخه جدید از اسمبلی را برای مدیر ارسال و مدیر، اسمبلی را نصب کرد و فایل‌های XML تنظیمات برنامه یا ماشین را به صورت دستی تغییر داد. عموما وقتی یک سازنده خطایی را در یک اسمبلی برطرف می‌کند، می‌خواهد با راهی ساده اسمبلی جدید را بسته‌بندی و بین تمام کاربران توزیع کند. اما سازنده نیاز به راهی دارد تا به CLR هر کاربر بگوید که به جای اسمبلی قدیمی اسمبلی جدید را بارگذاری کند. مطمئنا هر کاربر می‌تواند فایل XML تنظیمات برنامه یا ماشین خود را تغییر دهد، اما این کار همراه با اشتباه بوده و رایج نیست. آنچه سازنده نیاز دارد راهی ساده برای ساخت اطلاعات سیاستی است که با نصب اسمبلی جدید در ماشین کاربر نصب شود. در این بخش، نشان می‌دهم که چگونه سازنده‌ی یک اسمبلی می‌تواند این اطلاعات را بسازد.

فرض کنید شما سازنده‌ی یک اسمبلی هستید و نسخه‌ی جدیدی از اسمبلی تان ساخته اید که برخی خطاها را رفع می‌کند. وقتی اسمبلی جدید خود را برای ارسال به تمام کاربران بسته‌بندی می‌کنید، باید یک فایل XML تنظیمات نیز درست کنید. این فایل تنظیمات شبیه فایل‌های تنظیماتی است که پیرامون آن‌ها صحبت کردیم. یک نمونه از این فایل (**JeffTypes.config**) را برای اسمبلی **JeffTypes.dll** می‌بینید:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>

        <assemblyIdentity name="JeffTypes"
          publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

        <bindingRedirect
          oldVersion="1.0.0.0" newVersion="2.0.0.0" />

        <codeBase version="2.0.0.0"
          href="http://www.wintellect.com/JeffTypes.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

البته، سازنده‌ها می‌توانند این اطلاعات را فقط برای اسمبلی‌هایی که خودشان می‌سازند تنظیم کنند. به علاوه عناصر نشان داده شده در این جا تنها عناصری هستند که می‌توانند در فایل تنظیمات سیاست سازنده جای بگیرند، برای نمونه شما نمی‌توانید عناصر **probing** یا **publisherPolicy** را تعیین نمایید.

این فایل تنظیمات به CLR می‌گوید که هنگام ارجاع به نسخه ۱.۰.۰.۰ اسمبلی JeffTypes، نسخه‌ی ۲.۰.۰.۰ آن را بارگذاری کند. اکنون شما به عنوان سازنده می‌توانید یک اسمبلی بسازید که حاوی فایل تنظیمات سیاست سازنده باشد. شما فایل تنظیمات سیاست سازنده را با AL.exe بدین گونه می‌سازید:

```
AL.exe /out:Policy.1.0.JeffTypes.dll
      /version:1.0.0.0
      /keyfile:MyCompany.snk
      /linkresource:JeffTypes.config
```

بگذارید معانی سویچ‌های خط فرمان AL.exe را توضیح دهم:

- **/out** این سویچ به AL.exe می‌گوید یک فایل جدید PE به نام Policy.1.0.JeffTypes.dll که فقط حاوی مانیفست است، درست کند. نام این فایل اسمبلی خیلی مهم است. اولین بخش نام، Policy، به CLR می‌گوید که این اسمبلی حاوی اطلاعات سیاست سازنده است. بخش‌های دوم و سوم نام، 1.0، به CLR می‌گوید که این اسمبلی سیاست سازنده برای هر اسمبلی JeffTypes.dll که دارای شماره بزرگ و کوچک نسخه 1.0 است می‌باشد. سیاست‌های سازنده فقط بر شماره‌های بزرگ و کوچک نسخه اسمبلی اعمال می‌شوند، شما نمی‌توانید یک سیاست سازنده بسازید که مخصوص به ساخت یا بازبینی خاصی باشد. بخش چهارم نام، JeffTypes، نام اسمبلی که این سیاست سازنده برای آن است را مشخص می‌کند. بخش پنجم و پایانی نام، dll، پسوند فایل اسمبلی تولیدی است.
- **/version** این سویچ نسخه اسمبلی سیاست سازنده را تعیین می‌کند، این شماره نسخه ربطی به خود اسمبلی JeffTypes ندارد. می‌دانید که اسمبلی‌های سیاست سازنده قابل نسخه‌بندی هستند. امروز، ممکن است سازنده یک سیاست سازنده که نسخه ۱.۰.۰.۰ از JeffTypes را به نسخه ۲.۰.۰.۰ ارجاع دهد، بسازد. در آینده، ممکن است سازنده بخواهد نسخه ۱.۰.۰.۰ از JeffTypes را به نسخه ۲.۵.۰.۰ ارجاع دهد. CLR از این شماره نسخه استفاده می‌کند تا همواره آخرین نسخه اسمبلی سیاست سازنده را بردارد.
- **/keyfile** این سویچ باعث می‌شود AL.exe اسمبلی سیاست سازنده را با جفت کلید عمومی/خصوصی سازنده امضاء کند. این جفت کلید باید با جفت کلید استفاده شده در تمام نسخه‌های اسمبلی JeffTypes مطابقت داشته باشد. ضمناً، از این طریق CLR می‌داند که همان سازنده‌ی اسمبلی JeffTypes، این فایل سیاست سازنده را تولید کرده است.
- **/linkresource** این سویچ به AL.exe می‌گوید که فایل XML تنظیمات، فایلی جداگانه از اسمبلی در نظر گرفته شود. اسمبلی تولیدی شامل دو فایل است که هر دوی آن‌ها همراه با نسخه جدید اسمبلی JeffTypes باید بسته‌بندی شده و برای نصب به کاربر تحویل داده شوند. در ضمن، شما نمی‌توانید از سویچ **/embedresource** با AL.exe برای جاسازی فایل XML تنظیمات در فایل اسمبلی استفاده کنید چرا که CLR نیاز دارد فایل XML در فایلی جداگانه باشد.

وقتی این اسمبلی سیاست سازنده ساخته شد، می‌تواند همراه با اسمبلی جدید بسته‌بندی شده و برای نصب به کاربر داده شود. اسمبلی سیاست سازنده باید در GAC نصب شود. اگرچه اسمبلی JeffTypes هم می‌تواند در GAC نصب شود، ولی اجباری در کار نیست. همچنین آن می‌تواند در دایرکتوری اصلی برنامه یا هر دایرکتوری که با URL موجود در **codeBase** تعیین شده باشد، نصب شود.

مهم یک سازنده فقط هنگام نصب یک آپدیت یا نسخه سرویس پک یک اسمبلی، باید یک اسمبلی سیاست سازنده بسازد. هنگامیکه یک نصب از نو انجام می‌شود هیچ اسمبلی سیاست سازنده نباید نصب شود.

می‌خواهم نکته آخر پیرامون سیاست سازنده را بگویم. فرض کنید یک سازنده، یک اسمبلی سیاست سازنده را توزیع کرده است و به دلایلی، اسمبلی جدید خطاهای بیشتری از آنچه برطرف کرده را بروز داده است. اگر این اتفاق بیافتد، مدیر می‌خواهد به CLR بگوید که اسمبلی سیاست سازنده را نادیده بگیرد. برای اینکار، مدیر باید فایل تنظیمات برنامه را ویرایش کند و عنصر **publisherPolicy** زیر را به آن اضافه کند.

```
<publisherPolicy apply="no"/>
```

این عنصر می‌تواند به عنوان عنصر فرزند از عنصر **<assemblyBinding>** در فایل تنظیمات برنامه قرار بگیرد تا بر تمام اسمبلی‌ها اعمال شود و یا به عنوان عنصر فرزند از عنصر **<dependentAssembly>** در فایل تنظیمات برنامه جای بگیرد تا بر یک اسمبلی خاص اعمال شود. وقتی CLR، فایل تنظیمات برنامه را پردازش می‌کند، می‌بیند که نباید GAC را برای اسمبلی سیاست سازنده جستجو کند. پس CLR با نسخه قدیمی اسمبلی به کار خود ادامه می‌دهد. توجه کنید که همچنان CLR هر سیاست تعریف شده در فایل Machine.config را بررسی و اعمال می‌کند.

مهم یک اسمبلی سیاست سازنده راهی برای یک سازنده است که درباره سازگاری نسخه‌های مختلف یک اسمبلی حرف بزند. اگر نسخه‌ی جدید یک اسمبلی با نسخه‌ی قدیمی سازگار نباشد، سازنده نباید یک اسمبلی سیاست سازنده تولید کند. در کل، وقتی نسخه‌ی جدیدی از اسمبلی می‌سازید که مشکلاتی را برطرف می‌کند، از اسمبلی سیاست سازنده استفاده کنید. شما باید نسخه جدید اسمبلی را برای سازگاری با نسخه‌های پیشین تست کنید. به بیان دیگر، اگر ویژگی‌هایی به اسمبلی خود اضافه می‌کنید، باید در نظر بگیرید که اسمبلی ارتباطی با نسخه قدیمی نداشته باشد و شما نباید یک اسمبلی سیاست سازنده بسازید. به علاوه، هیچ نیازی به تست‌های سازگاری با نسخه‌های پیشین برای اسمبلی این چنینی نیست.

فصل ۴: مبانی نوع

در این فصل، اطلاعات پایه برای کار با نوع‌ها و زبان مشترک اجرایی (CLR) را به شما می‌دهم. به خصوص، حداقل رفتارهایی که می‌توانید از هر نوع انتظار داشته باشید را خواهیم گفت. همچنین امنیت نوع، فضاهای نام (namespace)، اسمبلی‌ها و راه‌های مختلف تبدیل از یک نوع به نوع دیگر را بیان می‌کنم. در پایان، با بیان ارتباط میان نوع‌ها، پشته‌های ترد، و هیپ مدیریت‌شده در زمان اجرا، فصل را تمام می‌کنم.

همه ی نوع‌ها از System.Object مشتق می‌شوند

CLR نیاز دارد که تمام نوع‌ها سرانجام از نوع **System.Object** مشتق شوند. این بدین معنی است که دو تعریف زیر معادل یکدیگرند:

```
// Implicitly derived from object          // Explicitly derived from object
class Employee {                          class Employee : System.Object {
... ..                                     }
}
```

چون همه ی نوع‌ها سرانجام از **System.Object** مشتق می‌شوند، این تضمین شده است که هر شی از هر نوعی دارای حداقل متدهایی است. به خصوص، کلاس **System.Object** متدهای عمومی لیست شده در جدول ۴-۱ را ارائه می‌کند:

جدول ۴-۱ متدهای عمومی از System.Object

متد عمومی	توضیح
Equals	اگر دو شی دارای مقدار یکسان باشند، true برمی‌گرداند. برای اطلاعات بیشتر درباره ی این متد، بخش "هویت و برابری اشیاء" در فصل ۵ "نوع‌های اصلی، ارجاعی و مقداری" را مطالعه کنید.
GetHashCode	یک هش را برای مقدار این شی برمی‌گرداند. یک نوع باید این متد را بازنویسی کند اگر اشیاء آن، می‌خواهند در یک مجموعه جدول هش استفاده شوند. متد باید توزیع خوبی از اشیای خود ارائه کند. متاسفانه این متد در Object تعریف شده است در حالیکه بسیاری از نوع‌ها هرگز به عنوان کلید در جدول هش استفاده نمی‌شوند. به جای آن، این متد می‌بایست در یک رابط (interface) تعریف می‌شد. برای اطلاعات بیشتر پیرامون این متد، بخش "کدهای هش شی" در فصل ۵ را ببینید.
ToString	به صورت پیش فرض نام کامل نوع (this.GetType().FullName) را برمی‌گرداند. بازنویسی این متد بسیار رایج است تا یک شی String که معرف حال شی باشد را برگرداند. برای نمونه، نوع‌های اصلی، مثل Boolean و Int32 این متد را برای بیان مقدارشان در قالب یک رشته، بازنویسی کرده اند. همچنین بازنویسی متد برای خطایابی نیز رایج است، شما می‌توانید آن را فراخوانی کنید و یک رشته که مقدار فیلدهای شی را نشان می‌دهد، دریافت کنید. در واقع، دیباگر ویژوال استودیو برای نشان دادن نمایش رشته‌ای یک شی، به صورت خودکار این متد را صدا می‌زند. توجه کنید که ToString از CultureInfo ای که متعلق به ترد فراخوانی کننده است، آگاه می‌باشد. فصل ۱۴، "کاراکترها، رشته‌ها و کار با متن"، ToString را با جزئیات بیشتری بحث می‌کند.
GetType	یک نمونه از شی‌ای مشتق شده از Type را برمی‌گرداند که نوع شی استفاده شده برای فراخوانی GetType را شناسایی می‌کند. شی Type بازگردانده شده می‌تواند به کمک کلاس‌های رفلکشن برای بدست آوردن اطلاعات متادیتا درباره ی نوع شی، استفاده شود. رفلکشن در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن" صحبت می‌شود. متد GetType غیرمجازی است که مانع از بازنویسی متد و دروغ گفتن درباره نوعش و از بین بردن امنیت نوع توسط یک کلاس می‌شود.

به علاوه، نوع‌هایی که از **System.Object** مشتق می‌شوند به متدهای محافظت شده در جدول ۴-۲ دسترسی دارند.

جدول ۴-۲ متدهای محافظت شده از System.Object

متد محافظت شده	توضیح
----------------	-------

این متد غیرمجازی یک نمونه^{۳۱} جدید از نوع ساخته و فیلدهای نمونه^{۳۲} شی جدید را برابر با فیلدهای نمونه از شی **this** قرار می‌دهد. یک ارجاع به نمونه‌ی جدید برگردانده می‌شود.

Finalize

این متد مجازی، هنگامیکه جمع آوری کننده‌ی زباله تعیین کند که شی، زباله است، قبل از پس گرفتن حافظه از شی فراخوانی می‌شود. نوع‌هایی که هنگام جمع آوری نیاز به پاکسازی دارند باید این متد را بازنویسی کنند. درباره‌ی این متد مهم در فصل ۲۱ "مدیریت حافظه خودکار(جمع آوری زباله)" صحبت می‌کنم.

CLR نیاز دارد که تمام اشیاء با عملگر **new** ساخته شوند. خط زیر نشان می‌دهد که چگونه یک شی **Employee** بسازید:

```
Employee e = new Employee("Constructor Param1");
```

آنچه عملگر **new** انجام می‌دهد عبارت است از:

- تعداد بایت‌های مورد نیاز برای تمام فیلدهای نمونه‌ی تعریف شده در نوع و تمام نوع‌های پایه‌ی آن تا و شامل **System.Object** را حساب می‌کند (**System.Object** خودش فیلد نمونه‌ای ندارد). هر شی در هیپ به تعدادی اعضای اضافی - که اشاره‌گر شی نوع **type object pointer** و اندیس بلوک همزمانی **sync block index** نامیده می‌شوند - نیاز دارد. CLR از این اعضای اضافی برای مدیریت شی استفاده می‌کند. بایت‌های مورد نیاز برای این اعضای اضافی به اندازه شی افزوده می‌شود.
- حافظه را با تخصیص تعداد بایت‌های مورد نیاز برای نوع مشخص شده از هیپ مدیریت شده، به شی اختصاص می‌دهد؛ سپس تمام بایت‌ها با صفر (۰) پر می‌شوند.
- اعضای اشاره‌گر شی نوع شی و اندیس بلوک همزمانی را مقداردهی اولیه می‌کند.
- سازنده‌ی نمونه^{۳۳} از شی را صدا می‌زند و هر آرگومان تعیین شده (رشته "Constructor Param1" در مثال قبل) در فراخوانی **new** را به آن ارسال می‌کند. اکثر کامپایلرها در یک سازنده، کدی برای فراخوانی سازنده‌ی کلاس پایه قرار می‌دهند. هر سازنده مسئول مقداردهی فیلدهای نمونه‌ی تعریف شده توسط نوعی است که سازنده را صدا می‌زند. سرانجام، سازنده‌ی **System.Object** صدا زده می‌شود و این متد سازنده هیچ کاری انجام نمی‌دهد و فقط برمی‌گردد. شما می‌توانید این را با بارگذاری **mscorlib.dll** توسط **ILDasm.exe** و بررسی متد سازنده-ی **System.Object** بررسی کنید.

پس از آنکه **new** همه‌ی کارهای خود را انجام داد، یک ارجاع (یا اشاره‌گر) به شی تازه ساخته شده برمی‌گرداند. در مثال قبلی، این ارجاع در متغیر **e** که از نوع **Employee** است ذخیره می‌شود.

ضمناً، عملگر **new** دارای عملگر مکمل **delete** نیست؛ چون راهی برای آزاد کردن حافظه یک شی به صورت صریح وجود ندارد. CLR از جمع آوری زباله (که در فصل ۲۱ توضیح داده می‌شود) استفاده می‌کند که به صورت خودکار وقتی شی دیگر استفاده یا دسترسی نمی‌شود متوجه شده و حافظه شی را آزاد می‌کند.

تبدیل میان نوع ها

یکی از مهمترین ویژگی‌های CLR امنیت نوع است. در زمان اجرا، CLR همیشه بررسی می‌کند که شی از چه نوعی است. همیشه می‌توانید نوع یک شی را با فراخوانی متد **GetType** بدست آورید. چون این متد غیرمجازی است، برای یک نوع غیرممکن است که خود را به جای نوع دیگر جا بزند. برای نمونه، نوع **Employee** نمی‌تواند متد **GetType** را بازنویسی کند و نوع **SuperHero** را بازگرداند.

اغلب برنامه‌نویسان نیاز دارند که یک شی را به نوع‌های مختلف تبدیل کنند. CLR اجازه می‌دهد که یک شی را به نوع خودش یا هر یک از نوع‌های پایه‌اش تبدیل کنید. زبان برنامه‌نویسی مورد انتخاب شما چگونگی عملیات‌های تبدیل را تعیین می‌کند. برای نمونه، سی‌شارپ به هیچ نحو خاصی برای تبدیل یک شی به نوع‌های پایه‌اش نیاز ندارد چون تبدیل به نوع‌های پایه همواره به عنوان تبدیل ضمنی امن در نظر گرفته می‌شود. هرچند، سی‌شارپ نیاز دارد که به

^{۳۱} Instance: وقتی از یک نوع، شی‌ای را می‌سازید به آن شی نمونه یا Instance می‌گویند.

^{۳۲} Instance Field: فیلد نمونه به فیلدی گفته می‌شود که از طریق نمونه (Instance) شی قابل دسترسی باشد. در مقابل فیلدهای نمونه، فیلدهای استاتیک وجود دارند که از طریق خود نوع قابل دسترسی هستند.

^{۳۳} Instance Constructor: در مقابل آن سازنده‌ی نوع را داریم.

صورت صریح یک شی را به یکی از نوع‌های مشتق شده‌اش تبدیل کنید چون تبدیل این چینی می‌تواند در زمان اجرا شکست بخورد. کد زیر تبدیل به نوع-های پایه و مشتق شده را نشان می‌دهد.

```
// This type is implicitly derived from System.Object.
internal class Employee {
    ...
}
public sealed class Program {
    public static void Main() {
        // No cast needed since new returns an Employee object
        // and Object is a base type of Employee.
        Object o = new Employee();

        // Cast required since Employee is derived from Object.
        // Other languages (such as Visual Basic) might not require
        // this cast to compile.
        Employee e = (Employee) o;
    }
}
```

این کد نشان می‌دهد کامپایلر برای کامپایل کد شما به چه چیزی نیاز دارد. اکنون توضیح می‌دهم در زمان اجرا چه رخ می‌دهد. در زمان اجرا، CLR عملیات-های تبدیل را بررسی می‌کند تا مطمئن شود که تبدیل‌ها همواره به نوع واقعی شی یا هر کدام از نوع‌های پایه آن باشد. برای نمونه، کد زیر کامپایل می‌شود اما در زمان اجرا یک اکسپشن **InvalidCastException** تولید می‌کند.

```
internal class Employee {
    ...
}
internal class Manager : Employee {
    ...
}

public sealed class Program {
    public static void Main() {
        // Construct a Manager object and pass it to PromoteEmployee.
        // A Manager IS-A Object: PromoteEmployee runs OK.
        Manager m = new Manager();
        PromoteEmployee(m);

        // Construct a DateTime object and pass it to PromoteEmployee.
        // A DateTime is NOT derived from Employee. PromoteEmployee
        // throws a System.InvalidCastException exception.
        DateTime newYears = new DateTime(2010, 1, 1);
        PromoteEmployee(newYears);
    }

    public static void PromoteEmployee(Object o) {
        // At this point, the compiler doesn't know exactly what
        // type of object o refers to. So the compiler allows the
        // code to compile. However, at runtime, the CLR does know
```

```
// what type o refers to (each time the cast is performed) and
// it checks whether the object's type is Employee or any type
// that is derived from Employee.
Employee e = (Employee) o;
...
}
}
```

در متد **Main**، یک شی **Manager** ساخته می‌شود و به **PromoteEmployee** ارسال می‌گردد. این کد کامپایل و اجرا می‌شود؛ چون **Manager** سرانجام از **Object** مشتق می‌شود و **Object** چیزی است که **PromoteEmployee** انتظار دارد. درون **PromoteEmployee**، **CLR** تایید می‌کند که **o** به شی‌ای اشاره می‌کند که از نوع **Employee** یا یکی از نوع‌های مشتق شده از **Employee** است. چون **Manager** از **Employee** مشتق می‌شود، **CLR** تبدیل را انجام داده و به **PromoteEmployee** اجازه‌ی ادامه اجرا را می‌دهد. پس از آنکه **PromoteEmployee** برمی‌گردد، **Main** یک شی **DateTime** می‌سازد و آن را به **PromoteEmployee** ارسال می‌کند. مجدداً، **DateTime** از **Object** مشتق شده است و کامپایلر کدی که **PromoteEmployee** را صدا می‌زند، بدون مشکل کامپایل می‌کند. در داخل **PromoteEmployee**، **CLR** تبدیل را بررسی کرده و می‌فهمد که **o** به شی **DateTime** اشاره می‌کند و نه به **Employee** یا یکی از نوع‌های مشتق شده از **Employee**. در این لحظه، **CLR** اجازه‌ی تبدیل را نداده و یک اکسپشن **System.InvalidCastException** تولید می‌کند.

اگر **CLR** اجازه‌ی تبدیل را می‌داد، امنیت نوع به وجود نمی‌آمد و نتیجه غیر قابل پیش‌بینی می‌بود که می‌توانست به خراب شدن برنامه و ایجاد مشکلات امنیتی به خاطر جا زدن یک نوع به جای نوع دیگر منجر شود. کلاهبرداری نوع، نتیجه‌ی شکاف‌های امنیتی بسیاری است و پایداری و عملکرد برنامه را تحت تاثیر قرار می‌دهد. پس امنیت نوع یک بخش بسیار مهم از **CLR** است.

ضمناً، روش صحیح تعریف متد **PromoteEmployee**، تعیین یک نوع **Employee** به جای نوع **Object** به عنوان پارامتر آن است تا کامپایلر خطای زمان کامپایل را تولید نکند. این کار، برنامه‌نویس را از انتظار برای یافتن رخداد اکسپشن در زمان اجرا رها می‌کند. من از **Object** برای این استفاده کردم که نشان دهم کامپایلر سی‌شارپ و **CLR** چگونه با تبدیل و امنیت نوع کار می‌کنند.

تبدیل با عملگرهای **is** و **as** سی‌شارپ

راه دیگر تبدیل نوع در سی‌شارپ عملگر **is** است. عملگر **is** بررسی می‌کند که آیا یک شی با نوع داده شده سازگاری دارد و نتیجه این بررسی یک **Boolean**: **true** یا **false** است. عملگر **is** هرگز هیچ اکسپشنی تولید نمی‌کند. کد زیر را ببینیم:

```
Object o = new Object();
Boolean b1 = (o is Object); // b1 is true.
Boolean b2 = (o is Employee); // b2 is false.
```

اگر اشاره‌گر شی **null** باشد، عملگر **is** همیشه **false** برمی‌گرداند چون شی‌ای برای بررسی نوع آن وجود ندارد.

عملگر **is** معمولاً به صورت زیر استفاده می‌شود:

```
if (o is Employee) {
    Employee e = (Employee) o;
    // Use e within the remainder of the 'if' statement.
}
```

در این کد، در واقع **CLR** نوع شی را دوبار بررسی می‌کند: اول عملگر **is** بررسی می‌کند آیا **o** با نوع **Employee** سازگار است یا نه. اگر هست، درون عبارت **if**، **CLR** مجدداً هنگام تبدیل بررسی می‌کند که **o** به یک **Employee** اشاره کند. بررسی نوع که **CLR** انجام می‌دهد، امنیت را بالا می‌برد اما بی‌شک بر کارایی برنامه اثرگذار است چون **CLR** باید نوع واقعی شی‌ای که متغیر (**o**) به آن اشاره می‌کند را تعیین کند و سپس در سلسله مراتب وراثت حرکت کرده و هر نوع پایه را در مقابل نوع (**Employee**) بررسی کند. چون این روش برنامه‌نویسی بسیار رایج است، سی‌شارپ راهی برای سادگی این کد و بهبود کارایی خود با فراهم کردن عملگر **as** ارائه می‌کند:

```
Employee e = o as Employee;
if (e != null) {
    // Use e within the 'if' statement.
}
```

در این کد، CLR بررسی می‌کند آیا **o** با نوع **Employee** سازگار است و اگر هست، **as** یک اشاره‌گر غیر **null** به همان شی برمی‌گرداند. اگر **o** با نوع **Employee** سازگار نیست، عملگر **.as null** برمی‌گرداند. توجه کنید **as** باعث می‌شود CLR نوع یک شی را فقط یک بار بررسی کند. عبارت **if** فقط بررسی می‌کند آیا **e null** است که این بررسی سریعتر از بررسی نوع یک شی صورت می‌گیرد.

عملگر **as** مشابه تبدیل (casting) کار می‌کند با این تفاوت که **as** هرگز یک اکسپشن تولید نمی‌کند. به جای آن، اگر شی قابل تبدیل نبود، نتیجه **null** می‌شود. شما باید بررسی کنید آیا اشاره‌گر حاصل **null** است یا خیر. در صورت عدم بررسی، استفاده از اشاره‌گر حاصل، منجر به تولید **System.NullReferenceException** می‌شود. کد زیر این را نشان می‌دهد:

```
Object o = new Object(); // Creates a new Object object
Employee e = o as Employee; // Casts o to an Employee
// The cast above fails: no exception is thrown, but e is set to null.
```

برای آنکه مطمئن شوید هر آنچه گفته شد را خوب درک کرده اید، آزمون زیر را انجام دهید. فرض کنید دو تعریف کلاس به صورت زیر داریم:

```
internal class B { // Base class
}

internal class D : B { // Derived class
}
```

حال، خطاهای سی‌شارپ موجود در جدول ۳-۴ را بررسی کنید. برای هر خط، تعیین کنید آیا این خط کامپایل شده و با موفقیت اجرا می‌شود (OK)، منجر به خطای زمان کامپایل می‌شود (CTE) یا منجر به خطای زمان اجرا می‌شود (RTE).

جدول ۳-۴ آزمون امنیت نوع

عبارت	OK	CTE	RTE
Object o1 = new Object();	✓		
Object o2 = new B();	✓		
Object o3 = new D();	✓		
Object o4 = o3;	✓		
B b1 = new B();	✓		
B b2 = new D();	✓		
D d1 = new D();	✓		
B b3 = new Object();		✓	
D d2 = new Object();		✓	
B b4 = d1;	✓		
D d3 = b2;		✓	
D d4 = (D) d1;	✓		
D d5 = (D) b2;	✓		
D d6 = (D) b1;			✓
B b5 = (B) o1;			✓
B b6 = (D) b2;	✓		

نکته سی‌شارپ اجازه می‌دهد که متدهای عملگر تبدیل را همانگونه که در بخش "متدهای عملگر تبدیل" از فصل ۹ "پارامترها" بیان شده است، تعریف کنید. این متدها فقط هنگام تبدیل فراخوانی می‌شوند. آن‌ها هرگز هنگام استفاده از عملگر **as** یا **is** سی‌شارپ، فراخوانی نمی‌شوند.

فضاهای نام (Namespace) و اسمبلی ها

فضاهای نام اجازه می‌دهند که نوع‌های مرتبط را به صورت منطقی گروه بندی کنید و برنامه‌نویسان معمولاً از آن‌ها برای ساده سازی یافتن یک نوع خاص استفاده می‌کنند. برای نمونه، فضای نام **System.Text** تعدادی نوع برای دستکاری رشته‌ها تعریف می‌کند و فضای نام **System.IO** تعدادی نوع برای انجام عملیات‌های ورودی خروجی تعریف می‌کند. کد زیر یک شی **System.IO.FileStream** و یک شی **System.Text.StringBuilder** را تعریف می‌کند:

```
public sealed class Program {
    public static void Main() {
        System.IO.FileStream fs = new System.IO.FileStream(...);
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
    }
}
```

همانطور که می‌بینید، کد بسیار طولانی شده است و اگر راهی برای مختصر نویسی هنگام ارجاع نوع‌های **FileStream** و **StringBuilder** باشد، میزان تایپ کردن را کاهش می‌دهد. خوشبختانه، اکثر کامپایلرها مکانیزمی برای کاهش کدنویسی ارائه می‌کنند. کامپایلر سی‌شارپ این مکانیزم را از طریق دستور^{۳۴} **using** فراهم می‌کند. کد زیر معادل کد قبلی است:

```
using System.IO; // Try prepending "System.IO."
using System.Text; // Try prepending "System.Text."
```

```
public sealed class Program {
    public static void Main() {
        FileStream fs = new FileStream(...);
        StringBuilder sb = new StringBuilder();
    }
}
```

برای کامپایلر، یک فضای نام، یک راه آسان برای طولانی کردن نام نوع و انحصار بیشتر آن با افزودن نمادهایی که با نقطه از هم جدا هستند به ابتدای نام می‌باشد. پس کامپایلر در این مثال، ارجاع به **FileStream** را به عنوان **System.IO.FileStream** تفسیر می‌کند. به همین طریق، کامپایلر ارجاع به **StringBuilder** را با عنوان **System.Text.StringBuilder** تفسیر می‌کند.

استفاده از دستور **using** کاملاً انتخابی است؛ اگر ترجیح می‌دهید همیشه می‌توانید نام کامل یک نوع را تایپ کنید. دستور **using** سی‌شارپ به کامپایلر می‌گوید که پیشنهادهای مختلف را تا یافتن یک تطابق، به یک نوع اضافه کند.

مهم CLR هیچ چیزی پیرامون فضای نام نمی‌داند. وقتی شما به یک نوع دسترسی پیدا می‌کنید، CLR نیاز دارد نام کامل نوع (که ممکن است واقعا طولانی شود) و این که کدام اسمبلی شامل تعریف نوع است را بداند تا اجراکننده بتواند اسمبلی صحیح را بارگذاری کرده، نوع را یافته و به آن دسترسی پیدا کند.

در مثال قبل، کامپایلر نیاز دارد مطمئن شود که هر نوع ارجاعی، وجود داشته و کد من از نوع به درستی استفاده می‌کند: فراخوانی متدهایی که وجود دارند، ارسال تعداد صحیح آرگومان‌ها به این متدها، اطمینان از صحیح بودن نوع آرگومان‌ها، استفاده صحیح از مقدار برگشتی متد و ... اگر کامپایلر نتواند یک نوع با نام تعیین شده را در فایل‌های سورس کد یا هر اسمبلی ارجاعی بیابد، **System.IO** را به قبل از نام نوع افزوده و بررسی می‌کند آیا نوع حاصل با یک نوع موجود مطابقت دارد یا خیر.

اگر کامپایلر هنوز نتوانست تطابق پیدا کند، **System.Text** را قبل از نام نوع اضافه می‌کند. دو دستور **using** مذکور، به من اجازه می‌دهند که به راحتی در کد خود **FileStream** و **StringBuilder** را تایپ کنم و کامپایلر به صورت خودکار، ارجاع‌ها را به **System.IO.FileStream** و **System.Text.StringBuilder** بسط می‌دهد. مطمئنم براحتی می‌توانید تصور کنید که تا چه حد تایپ کمتری نیاز است و چقدر کد خواناتر می‌شود.

^{۳۴} using و namespace یک directive یا رهنمود کننده هستند. از این پس به جای directive از کلمه دستور استفاده می‌شود.

هنگام بررسی تعریف یک نوع، باید با سوییچ **reference** به کامپایلر گفته شود که کدام اسمبلی‌ها را بررسی کند. درست همانطور که در فصل ۲ "ساخت، بسته‌بندی، نصب و مدیریت برنامه‌ها و نوع‌ها" و فصل ۳ "اسمبلی‌های اشتراکی و اسمبلی‌های قوی‌نام" گفته شده است، کامپایلر تمام اسمبلی‌های ارجاعی را برای یافتن تعریف نوع اسکن می‌کند. وقتی کامپایلر اسمبلی صحیح را یافت، اطلاعات اسمبلی و اطلاعات نوع را در متادیتای ماژول مدیریت‌شده-ی تولیدی قرار می‌دهد. برای بدست آوردن اطلاعات اسمبلی، شما باید اسمبلی‌ای که هر نوع ارجاعی را تعریف می‌کند به کامپایلر ارسال کنید. کامپایلر سی-شارپ به صورت پیش‌فرض، حتی اگر صریحا اعلام نکنید، به صورت خودکار اسمبلی **mscorlib.dll** را نگاه می‌کند. اسمبلی **mscorlib.dll** حاوی تعاریف نوع‌های کتابخانه کلاس فریمورک (FCL) مثل **String**، **Object**، **Int32** و... است.

همانگونه که ممکن است تصور کنید، برخی مشکلات در روشی که کامپایلرها با فضاهای نام رفتار می‌کنند وجود دارد: ممکن است دو (یا بیشتر) نوع با نام یکسان در فضاهای نام متفاوت وجود داشته باشند. مایکروسافت قویا توصیه می‌کند که نام‌های منحصر بفرد برای نوع‌ها انتخاب کنید. اما به هر حال، گاهی اوقات این ممکن نیست. اجراکننده شما را به استفاده از کامپوننت‌ها تشویق می‌کند. برنامه شما ممکن است از یک کامپوننت ساخت مایکروسافت و یک کامپوننت دیگر ساخت **Wintellect** بهره بگیرد. هر دوی این شرکت‌ها ممکن است نوعی به نام **Widget** ارائه کنند - **Widget** مایکروسافت یک کار می‌کند و **Widget** شرکت **Wintellect** کار دیگری انجام می‌دهد. در این سناریو، شما بر نام گذاری نوع‌ها کنترلی ندارید، پس می‌توانید بین این دو نوع با نام کامل آن‌ها تفاوت قائل شوید. برای ارجاع به **Widget** شرکت **Microsoft** شما از **Microsoft.Widget** استفاده می‌کنید و برای ارجاع به **Widget** شرکت **Wintellect** شما از **Wintellect.Widget** استفاده می‌کنید. در کد زیر، ارجاع به **Widget** مبهم است، پس کامپایلر سی-شارپ خطای زیر را تولید می‌کند:

```
"error CS0104: 'widget' is an ambiguous reference"
```

```
using Microsoft; // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."
```

```
public sealed class Program {
    public static void Main() {
        widget w = new widget();// An ambiguous reference
    }
}
```

برای رفع این ابهام، باید صریحا به کامپایلر، **Widget** مورد نظر خود را اعلام کنید:

```
using Microsoft; // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."
```

```
public sealed class Program {
    public static void Main() {
        Wintellect.widget w = new Wintellect.widget(); // Not ambiguous
    }
}
```

یک شکل دیگر استفاده از دستور **using** سی‌شارپ، ساخت نام مستعار برای یک تک نوع یا فضای نام است. اگر تعداد کمی نوع دارید که از یک فضای نام استفاده می‌کنند و نمی‌خواهید فضای نام سراسری را با نوع‌های فضای نام پر کنید، این ویژگی سودمند خواهد بود. کد زیر روش دیگری برای حل مشکل ابهام که در کد قبلی دیدیم، ارائه می‌کند:

```
using Microsoft; // Try prepending "Microsoft."
using Wintellect; // Try prepending "Wintellect."
```

```
// Define WintellectWidget symbol as an alias to Wintellect.widget
using WintellectWidget = Wintellect.widget;
```

```
public sealed class Program {
    public static void Main() {
```

```
wintellectwidget w = new wintellectwidget(); // No error now
}
}
```

این گونه رفع ابهام‌ها مفید است اما در بعضی سناریوها، باید بیش از این پیش روید. تصور کنید شرکت **Australian Boomerang Company** و شرکت **Alaskan Boat Corporation (ABC)** یک نوع به نام **BuyProduct** تولید کرده‌اند که قصد دارند در اسمبلی‌های خود آن را ارائه کنند. احتمالاً هر دو شرکت یک فضای نام به نام **ABC** که دارای نوع **BuyProduct** است می‌سازند. هر کس برنامه‌ای بنویسد که نیاز به خرید هر دو کالای بومرنگ و قایق داشته باشد به نحوی به مشکل برمی‌خورد مگر آنکه زبان برنامه‌نویسی راهی برای تمایز میان اسمبلی‌ها و نه فقط فضاهای نام از طریق برنامه‌نویسی فراهم کند. خوشبختانه، کامپایلر سی‌شارپ یک ویژگی که نام‌های مستعار خارجی **extern aliases** نامیده می‌شود را ارائه می‌کند. این ویژگی راه حلی برای این مشکل نادر فراهم می‌کند. همچنین این ویژگی راهی برای دسترسی به یک نوع از دو (یا چند) نسخه مختلف از همان اسمبلی را فراهم می‌کند. برای اطلاعات بیشتر پیرامون نام‌های مستعار خارجی به مشخصات زبان سی‌شارپ مراجعه کنید.

در کتابخانه‌ی خود، وقتی در حال طراحی نوع‌هایی هستید که انتظار دارید افراد دیگری از آن‌ها استفاده کنند، شما می‌بایست این نوع‌ها را در یک فضای نام تعریف کنید تا کامپایلر به راحتی آن‌ها را ابهام زدایی کند. در واقع، جهت کاهش احتمال برخورد میان نوع‌ها، باید از اسم کامل شرکت خود (و نه مخفف آن) برای نام فضای نام استفاده کنید. با مراجعه به **SDK** دات‌نت فریمورک، می‌توانید ببینید که مایکروسافت از **“Microsoft”** برای نوع‌های مخصوص مایکروسافت استفاده کرده است (برای نمونه فضاهای نام **Microsoft.CSharp**، **Microsoft.VisualBasic** و **Microsoft.Win32** را نگاه کنید).

ساخت یک فضای نام به سادگی نوشتن یک اعلان فضای نام در کدتان است. همانند مثال زیر (در سی‌شارپ):

```
namespace CompanyName {
    public sealed class A {                // TypeDef: CompanyName.A
    }

    namespace X {
        public sealed class B { ... }     // TypeDef: CompanyName.X.B
    }
}
```

کامنت سمت راست تعاریف کلاس‌ها، نام واقعی نوع که کامپایلر در جدول متادیتای تعریف نوع قرار می‌دهد را بیان می‌کنند، این نام واقعی از دید CLR است.

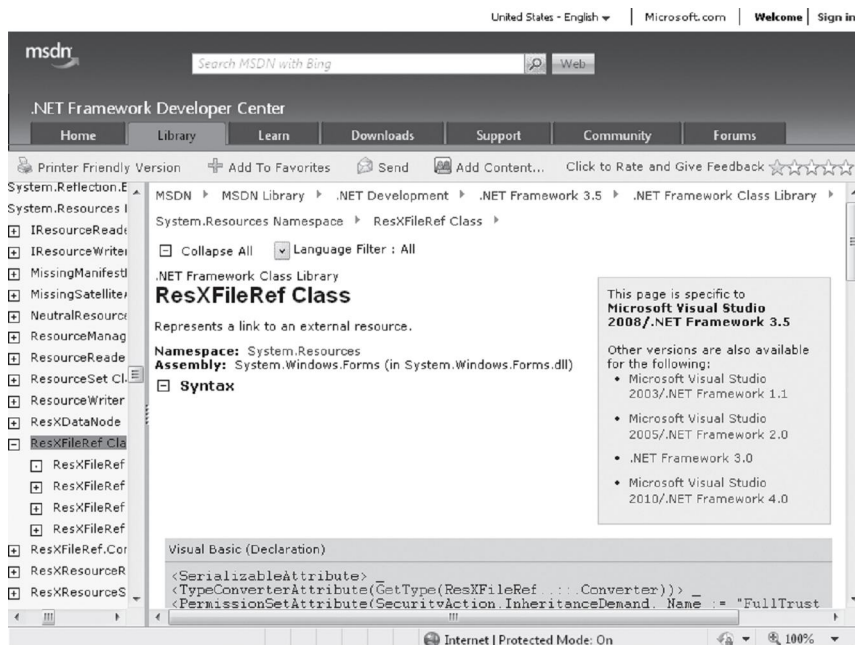
بعضی کامپایلرها اصلاً فضاهای نام را پشتیبانی نمی‌کنند و کامپایلرهای دیگر در تعریف معنی **“namespace”** برای یک زبان خاص، آزادند. در سی‌شارپ، دستور **namespace** به کامپایلر می‌گوید قبل از نام هر نوع که در کد آمده است، نام فضاهای نام را اضافه کند تا برنامه‌نویس تایپ کمتری انجام دهد.

چگونه فضاهای نام و اسمبلی‌ها به هم مرتبط می‌شوند

آگاه باشید که یک فضای نام و یک اسمبلی (فایلی که یک نوع را پیاده‌سازی می‌کند) لزوماً به هم ربط ندارند. به خصوص، نوع‌های مختلف متعلق به یک فضای نام ممکن است در چندین اسمبلی پیاده‌سازی شده باشند. برای نمونه، نوع **System.IO.FileStream** در اسمبلی **mscorlib.dll** پیاده‌سازی شده و نوع **System.IO.FileSystemWatcher** در اسمبلی **System.dll** قرار دارد. در واقع دات‌نت فریمورک اسمبلی‌ای به نام **System.IO.dll** ندارد.

یک اسمبلی می‌تواند شامل نوع‌هایی از چندین فضای نام مختلف باشد. برای نمونه، نوع‌های **System.Int32** و **System.Text.StringBuilder** هر دو در اسمبلی **mscorlib.dll** قرار دارند.

وقتی به مستندات SDK دات‌نت فریمورک نگاه کنید، راهنما به وضوح فضای نامی که نوع به آن تعلق دارد و اسمبلی‌ای که نوع در آن پیاده‌سازی شده است را نشان می‌دهد. در شکل ۱-۴، به وضوح می‌بینید (درست در بالای بخش Syntax) که نوع **ResXFileRef** بخشی از فضای نام **System.Resources** بوده و نوع در اسمبلی **System.Windows.Forms.dll** پیاده‌سازی شده است. برای کامپایل کدی که به نوع **ResXFileRef** ارجاع می‌کند، دستور `using System.Resources;` را به سورس‌کد خود اضافه کرده و از سویچ `/r:System.Windows.Forms.dll` برای کامپایلر استفاده کنید.

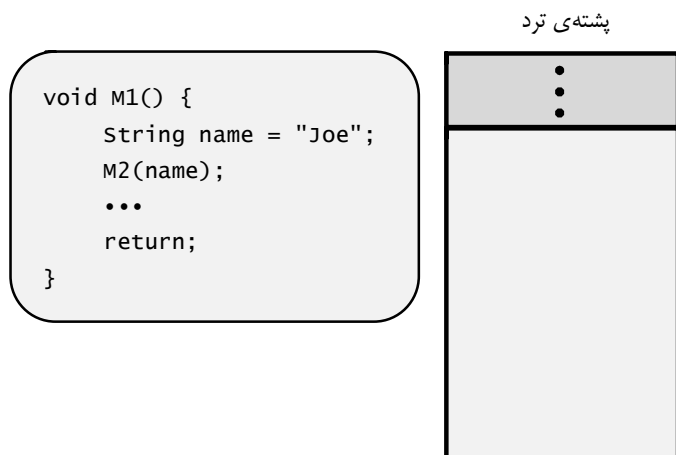


شکل ۱-۴ مستندات SDK که اطلاعات فضای نام و اسمبلی مربوط به یک نوع را نشان می‌دهد

چگونه چیزها در زمان اجرا به هم ربط پیدا می‌کنند

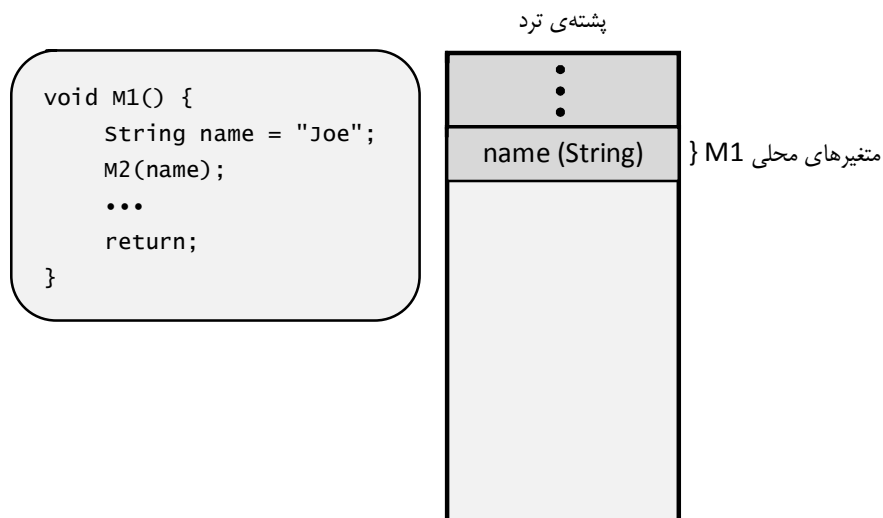
در این بخش رابطه میان نوع‌ها، شی‌ها، پشته‌ی یک ترد و هیپ مدیریت‌شده را در زمان اجرا توضیح می‌دهم. علاوه بر این، تفاوت بین فراخوانی متدهای استاتیک، متدهای نمونه (instance) و متدهای مجازی را توضیح می‌دهم. بگذارید با مبانی کامپیوتر شروع کنیم. آنچه می‌گویم مخصوص CLR نیست اما قصد دارم یک اساس کاری داشته باشیم و سپس بحث را با اطلاعات مخصوص به CLR تغییر می‌دهم.

شکل ۲-۴ یک پردازشی ویندوز که CLR روی آن بارگذاری شده است را نشان می‌دهد. در این پردازش ممکن است چندین ترد وجود داشته باشد. وقتی یک ترد ساخته می‌شود یک پشته‌ی ۱ مگابایتی به آن اختصاص داده می‌شود. این فضای پشته برای انتقال آرگومان‌ها به یک متد و متغیرهای محلی تعریف‌شده-ی درون متد استفاده می‌شود. در شکل ۲-۴ حافظه برای پشته‌ی یک ترد (در سمت راست) نشان داده شده است. پشته‌ها از آدرس‌های بالای حافظه به آدرس‌های پایین حافظه ساخته می‌شوند. در شکل، این ترد کدی را اجرا کرده است و پشته‌ی آن از قبل دارای تعدادی داده است (ناحیه سایه زده شده در بالای پشته). اکنون تصور کنید که ترد کدی را اجرا کرده است که متد **M1** را صدا می‌زند.



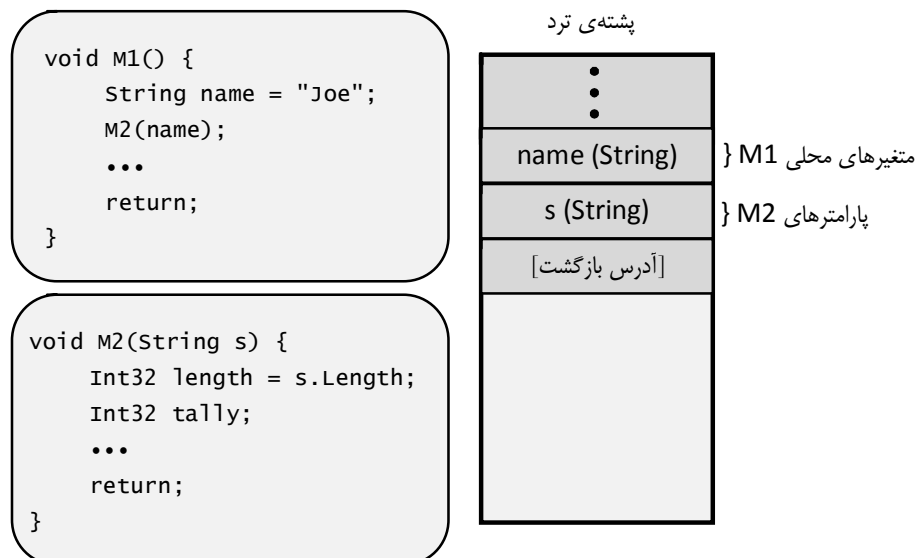
شکل ۲-۴ پشته‌ی یک ترد و متد **M1** در حال فراخوانی

ساده‌ترین متدها هم دارای کد مقدمه *prologue code* هستند که یک متد را قبل از شروع به کار، مقداردهی اولیه می‌کند. این متدها دارای کد خاتمه *epilogue code* نیز هستند که یک متد را پس از انجام کارش برای بازگشت به فراخوانی کننده، تمیز و پاک می‌کند. وقتی متد **M1** شروع به اجرا می‌کند، کد مقدمه‌ی آن حافظه را برای متغیر محلی **name** از پشته‌ی ترد تخصیص می‌دهد (شکل ۳-۴).



شکل ۳-۴ تخصیص متغیر محلی متعلق به **M1** در پشته‌ی ترد

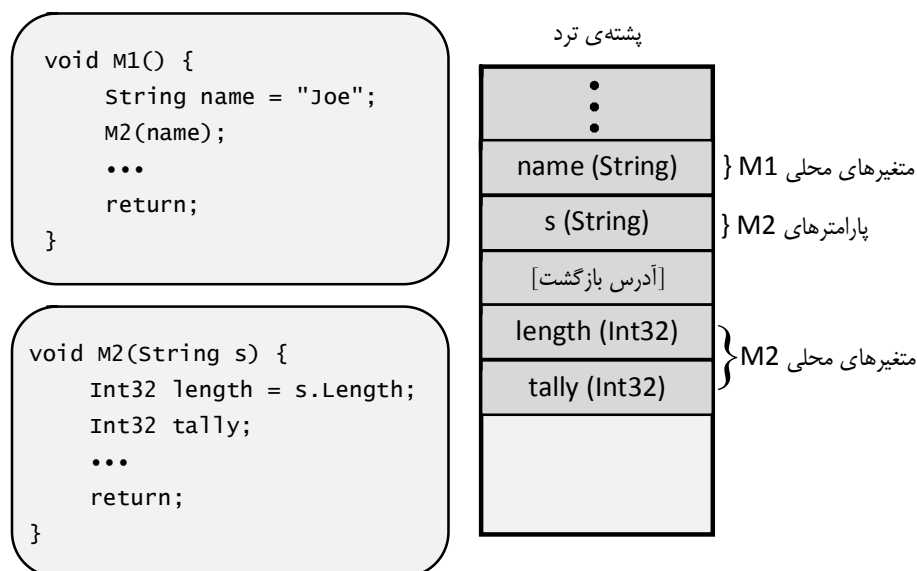
سپس، متد **M2** را صدا می‌زند و متغیر محلی **name** را به عنوان آرگومان بدان ارسال می‌کند. این باعث می‌شود که آدرس متغیر محلی **name** در پشته قرار گیرد (شکل ۴-۴). درون متد **M2**، موقعیت پشته با متغیر پارامتر **S** شناسایی می‌شود. (توجه کنید برخی معماری‌ها برای افزایش کارایی، آرگومان‌ها را از طریق رجیسترها ارسال می‌کنند، اما این تفاوت برای این بحث مهم نیست). همچنین وقتی یک متد فراخوانی می‌شود، آدرسی در متد فراخوانی کننده که متد فراخوانی شده باید به آن برگردد، در پشته قرار می‌گیرد (در شکل ۴-۴ نشان داده شده است).



شکل ۴-۴ M1 آرگومان‌ها و آدرس بازگشت را هنگام فراخوانی M2 در پشته می‌گذارد.

وقتی متد M2 شروع به اجرا می‌کند، کد مقدمه‌ی آن، حافظه را برای متغیرهای محلی **length** و **tally** از پشته‌ی ترد اختصاص می‌دهد (شکل ۴-۵). سپس کد درون M2 اجرا می‌شود. سرانجام، M2 به عبارت **return** می‌رسد که باعث می‌شود اشاره‌گر دستور پردازنده با مقدار آدرس بازگشت در پشته پر شود و قاب پشته‌ی M2 از بین می‌رود تا شبیه به آنچه در شکل ۴-۳ بود، گردد. در این لحظه، M1 به اجرای کدی که بلافاصله بعد از فراخوانی M2 آمده است، ادامه می‌دهد و قاب پشته‌ی آن دقیقاً وضعیت مورد نیاز M1 را نشان می‌دهد.

سرانجام، M1 با قراردادن مقدار آدرس برگشت (که در شکل‌ها نشان داده نشده، ولی دقیقاً در بالای آرگومان **name** در پشته قرار می‌گیرد) در اشاره‌گر دستور پردازنده، به فراخوانی کننده‌اش برمی‌گردد و قاب پشته‌ی M1 از بین می‌رود تا شبیه به آنچه در شکل ۴-۲ بود گردد. در این لحظه، متدی که M1 را صدا زده بود به اجرای کدی که بلافاصله بعد از فراخوانی M1 آمده است ادامه می‌دهد و قاب پشته‌ی آن دقیقاً وضعیت مورد نیاز آن متد را منعکس می‌کند.



شکل ۴-۵ اختصاص متغیرهای محلی M2 در پشته‌ی ترد

حال، بگذارید بحث را به سمت CLR بچرخانیم. فرض کنید این دو تعریف کلاس را داریم:

```

internal class Employee {
    public Int32 GetYearsEmployed() { ... }
    public virtual String GetProgressReport() { ... }
}

```

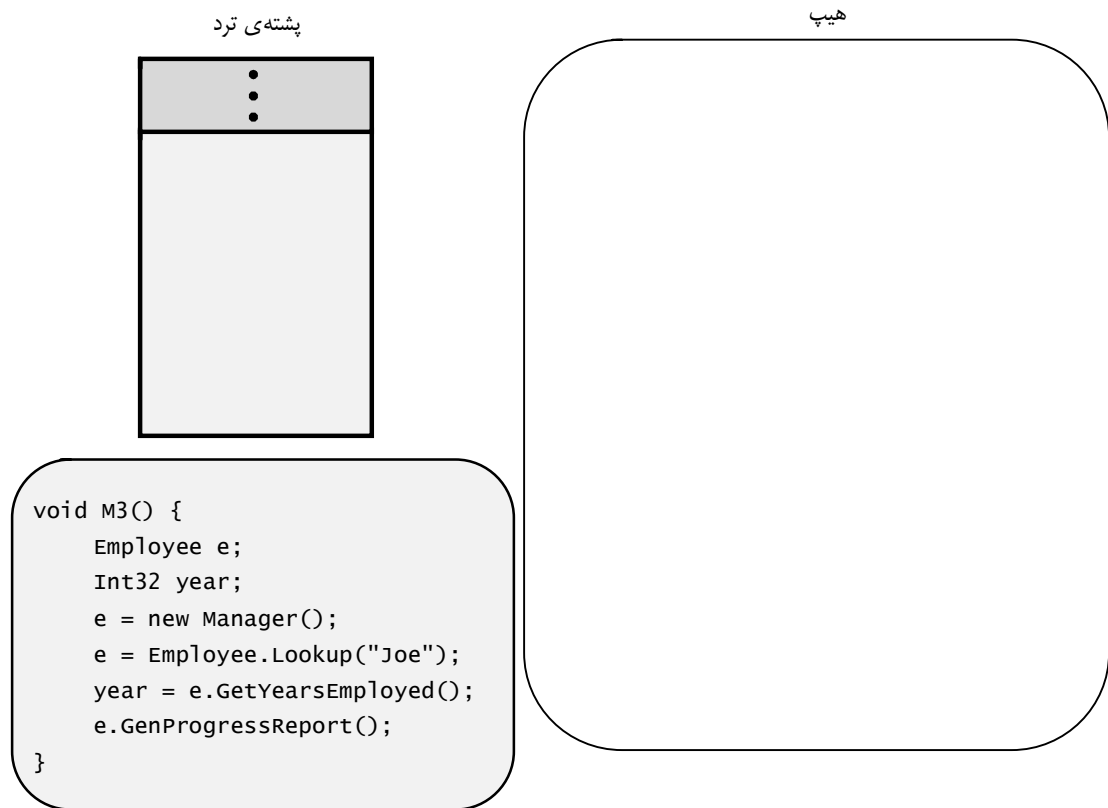
```

public static Employee Lookup(String name) { ... }
}

internal sealed class Manager : Employee {
    public override String GetProgressReport() { ... }
}

```

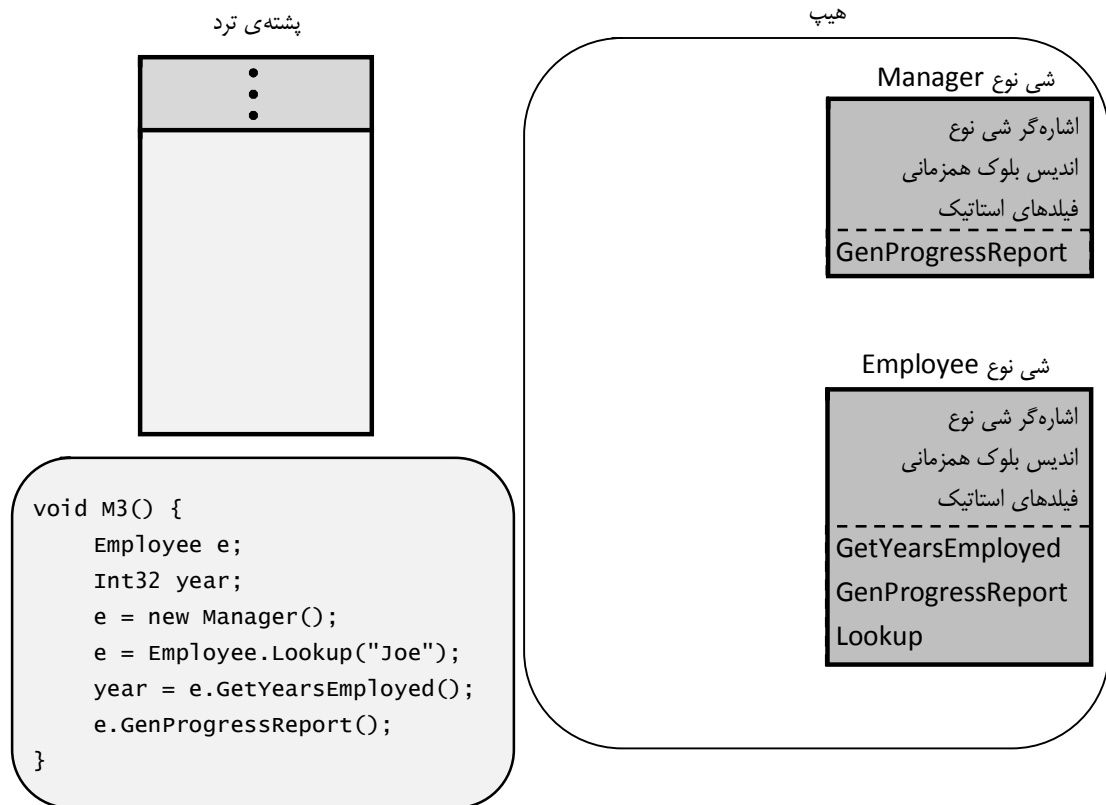
پردازشی ویندوز ما شروع به کار کرده است، CLR درون آن بارگذاری شده، هیپ مدیریت شده مقداردهی اولیه گشته و ترد (همراه با حافظه پشته ۱ مگابایتی) ساخته شده است. این ترد قبلاً کدی را اجرا کرده است و این کد تصمیم دارد که متد **M3** را صدا بزند. همه‌ی این‌ها در شکل ۶-۴ آمده است. متد **M3** شامل کدی است که نشان می‌دهد CLR چگونه کار می‌کند، این کدی نیست که معمولاً شما می‌نویسید، چون در واقع کار مفیدی انجام نمی‌دهد.



شکل ۶-۴ CLR در پردازش بارگذاری شده، هیپ مقداردهی گشته و یک پشته‌ی ترد همراه با **M3** که در حال فراخوانی است

وقتی کامپایلر فقط-در-لحظه (JIT) کد زبان میانی (IL) را به دستورات اصلی پردازنده تبدیل می‌کند، همه‌ی نوع‌هایی که درون **M3** ارجاع داده می‌شوند را می‌بیند: **Employee**، **Int32**، **Manager** و **String** (به خاطر "Joe"). در این لحظه، CLR مطمئن می‌شود اسمبلی‌هایی که این نوع‌ها را تعریف می‌کنند، بارگذاری شده باشند. سپس به کمک متادیتای اسمبلی، CLR اطلاعات این نوع را استخراج کرده و ساختمان داده‌هایی برای بیان نوع‌ها ایجاد می‌کند. ساختمان داده‌های اشیاء نوع ^{۳۵} **Employee** و **Manager** در شکل ۷-۴ آمده است. چون این ترد قبل از فراخوانی **M3** کدی را اجرا کرده است، فرض کنید که اشیاء نوع **String** و **Int32** قبلاً ساخته شده‌اند (این امر محتمل است، چون این‌ها، نوع‌های بسیار رایجی هستند). پس من آن‌ها را در شکل نشان نمی‌دهم.

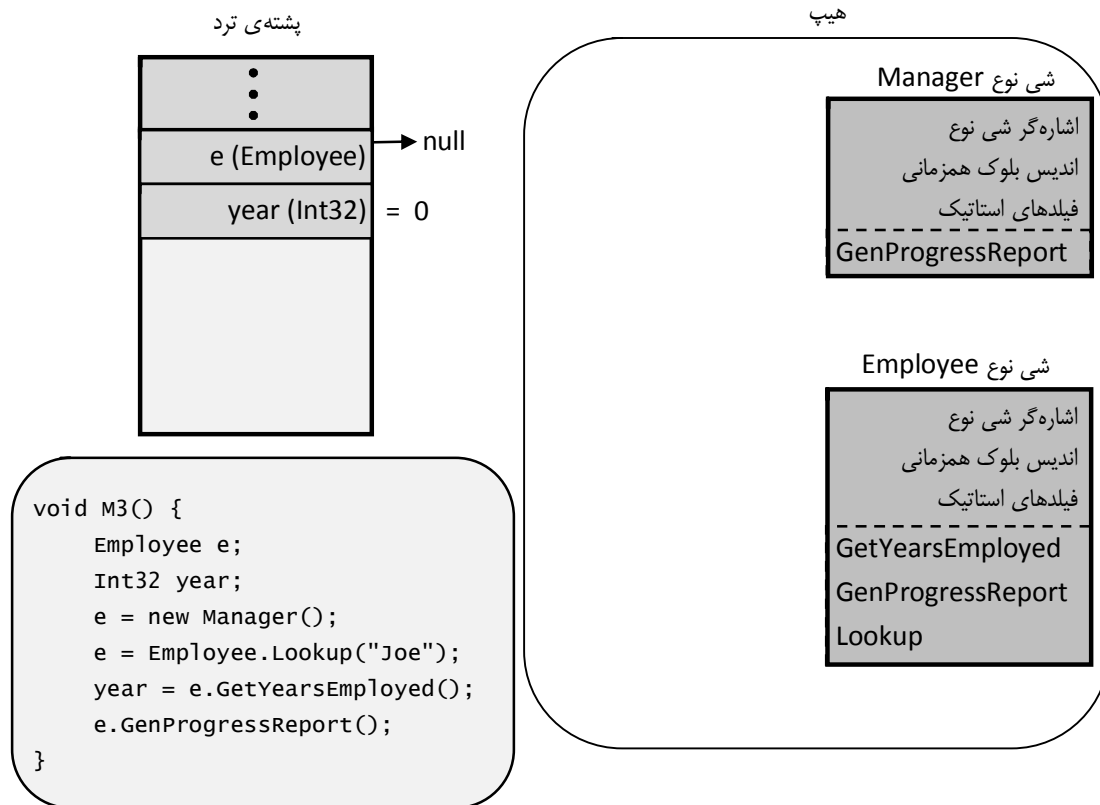
^{۳۵} منظور خود شی نوع (type object) است و نه شی‌ای از یک نوع (object of a type).



شکل ۷-۴ اشیاء نوع **Employee** و **Manager** که هنگام فراخوانی **M3** ایجاد شده اند.

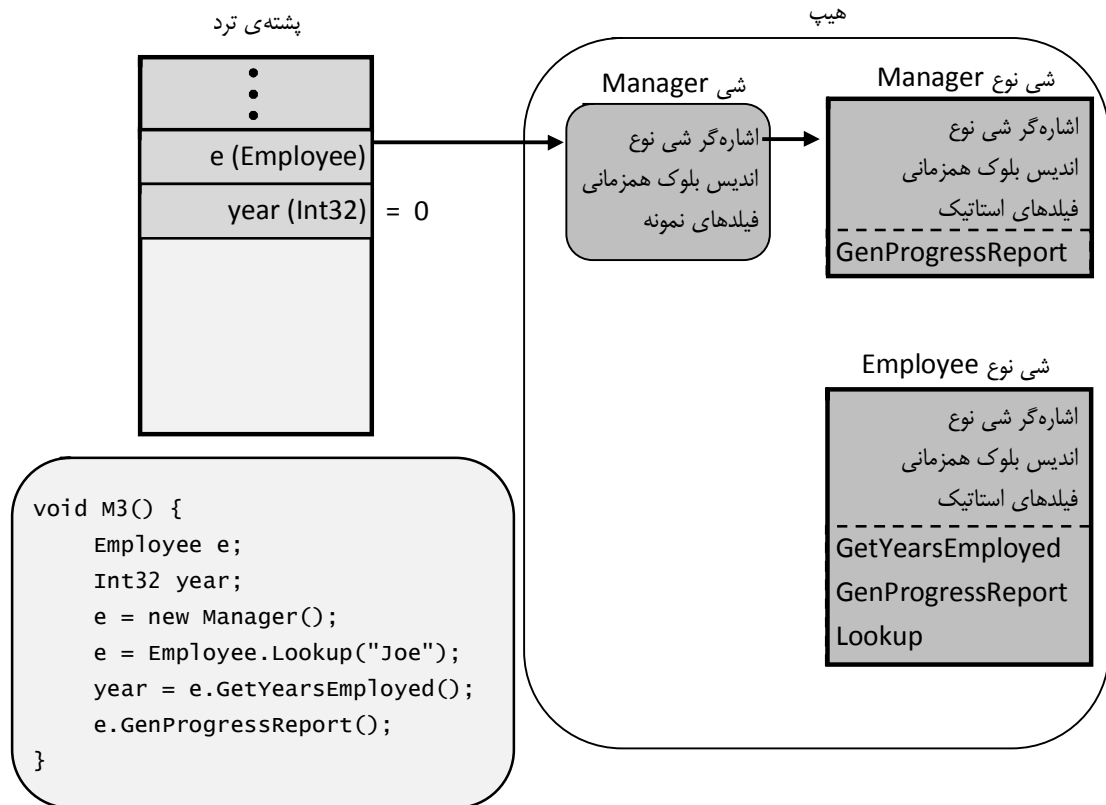
بگذارید کمی پیرامون این اشیاء نوع بحث کنیم. همانطور که قبلا در این فصل گفته شد، همه‌ی اشیاء در هیپ دو عضو اضافی دارند: اشاره‌گر شی نوع **type object pointer** و اندیس بلوک همزمانی **sync block index**. همانطور که می‌بینید، اشیاء نوع **Employee** و **Manager** هر دوی این اعضا را دارند. وقتی شما یک نوع تعریف می‌کنید، می‌توانید فیلدهای داده استاتیک در آن تعریف نمایید. بایت‌های این فیلدهای داده استاتیک درون خود اشیاء نوع تخصیص داده می‌شوند. سرانجام درون هر نوع یک جدول متد با یک ورودی به ازای هر متد تعریف شده در نوع، قرار دارد. این جدول متدی است که در فصل ۱ "مدل اجرایی CLR" بحث شد. چون نوع **Employee** سه متد تعریف می‌کند (**GetYearsEmployed**، **GenProgressReport** و **Lookup**)، سه ورودی در جدول متد متعلق به نوع **Employee** وجود دارد. چون نوع **Manager** یک متد دارد (یک بازنویسی از **GetProgressReport**) پس تنها یک ورودی در جدول متد متعلق به نوع **Manager** وجود دارد.

اکنون، پس از آنکه CLR مطمئن شد تمام اشیاء نوع که متد به آن‌ها نیاز دارد ساخته شده‌اند و کد **M3** کامپایل شده است، CLR به ترد اجازه می‌دهد کد اصلی **M3** را اجرا کند. وقتی کد مقدمه‌ی **M3** اجرا شد، همانطور که در شکل ۸-۴ آمده است، حافظه برای متغیرهای محلی از پشته‌ی ترد باید تخصیص یابد. ضمناً، CLR به صورت خودکار تمام متغیرهای محلی را با **null** یا **0** (صفر) به عنوان بخشی از کد مقدمه‌ی متد، مقداردهی اولیه می‌کند. هر چند، اگر کدی بنویسید که سعی در خواندن از یک متغیر محلی قبل از آنکه در کد خود صریحاً آن را مقداردهی اولیه کنید، داشته باشد، کامپایلر سی‌شارپ پیام خطای "Use of unassigned local variable" را تولید می‌کند.



شکل ۸-۴ تخصیص متغیرهای محلی M3 از پشته‌ی ترد

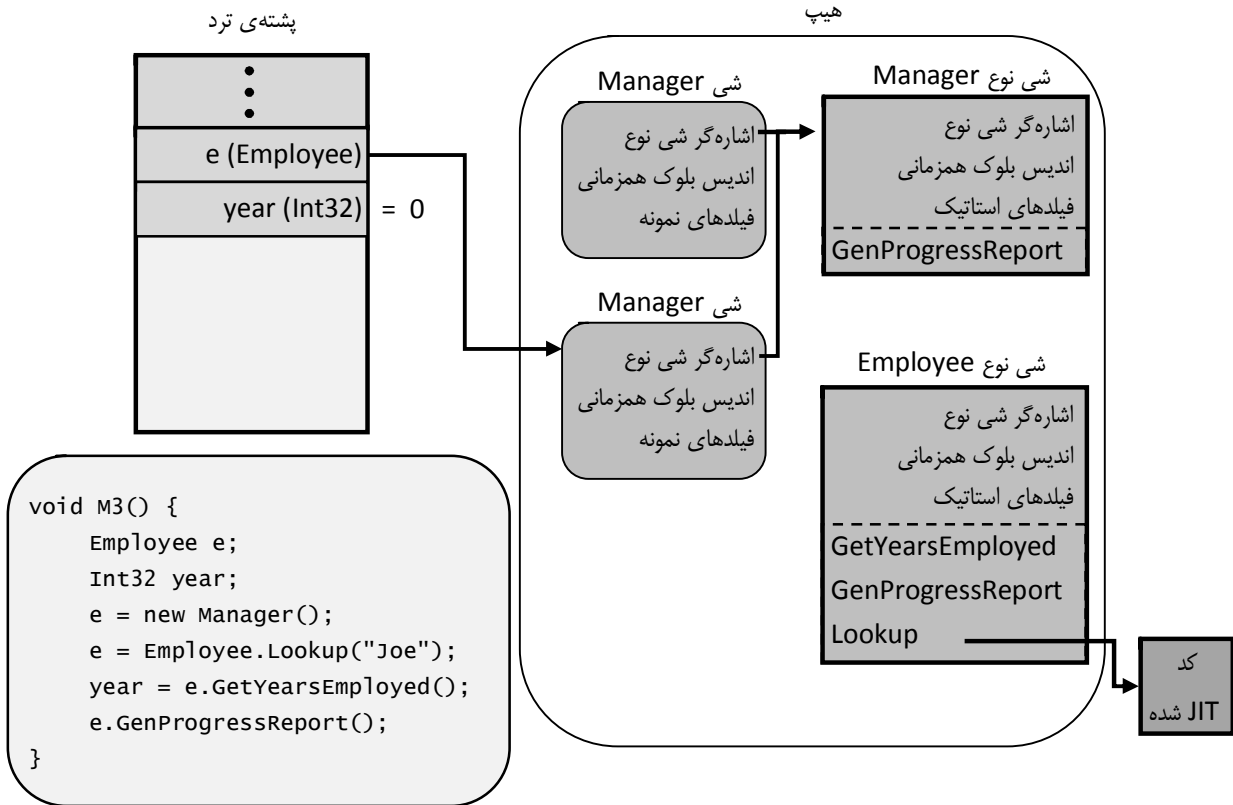
سپس، M3، کد خود را برای ساختن یک شی **Manager** اجرا می‌کند. این باعث می‌شود که یک نمونه از نوع **Manager**، یک شی **Manager**، همانگونه که در شکل ۹-۴ نشان داده شده، در هیپ مدیریت شده ساخته شود. همانگونه که می‌بینید، شی **Manager** مثل هر شی دیگری دارای اشاره‌گر شی نوع و اندیس بلوک همزمانی است. این شی همچنین بایت‌های مورد نیاز برای نگهداری فیلدهای داده‌ای نمونه که توسط نوع **Manager** و همچنین فیلدهای نمونه که توسط هر کلاس پایه‌ی نوع **Manager** (در اینجا، **Employee** و **Object**) ساخته شده‌اند، را داراست. وقتی یک شی جدید در هیپ ساخته می‌شود، CLR به صورت خودکار، اشاره‌گر شی نوع را برای ارجاع به شی نوع متناظر با شی (در این مورد، شی نوع **Manager**) مقداردهی اولیه می‌کند. علاوه بر این، CLR، اندیس بلوک همزمانی و تمام فیلدهای نمونه‌ی شی را به **null** یا **0** (صفر)، پیش از فراخوانی سازنده‌ی نوع، متدی که احتمالاً مقادیر فیلدهای داده‌ای نمونه را تغییر می‌دهد، مقداردهی اولیه می‌کند. عملگر **new** آدرس حافظه‌ی شی **Manager** را برمی‌گرداند که در متغیر **e** (در پشته‌ی ترد) ذخیره می‌شود.



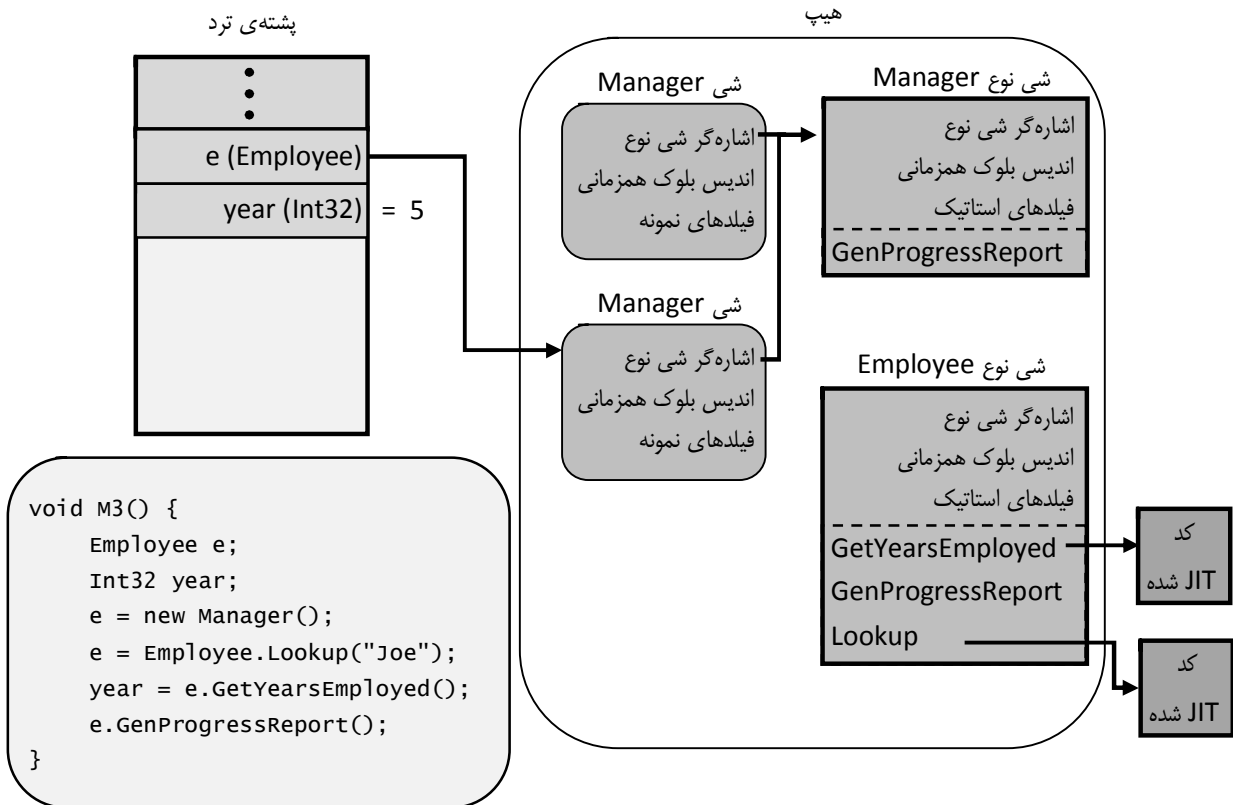
شکل ۹-۴ اختصاص حافظه و مقداردهی اولیه یک شی `Manager`

خط بعدی کد در `M3`، متد `Lookup` از `Employee` را فراخوانی می‌کند. وقتی یک متد استاتیک فراخوانی می‌شود، کامپایلر JIT شی نوع متناظر با شی نوعی که متد استاتیک را تعریف کرده پیدا می‌کند. سپس، کامپایلر JIT ورودی در جدول متد شی نوع که به متد فراخوانی شده اشاره می‌کند را پیدا کرده آن را (در صورت نیاز) JIT می‌کند و کد JIT شده را فراخوانی می‌کند. برای بحث ما، فرض کنید که متد `Lookup` از `Employee` برای یافتن `Joe` یک پایگاه داده را جستجو می‌کند. بگوییم که پایگاه داده تعیین می‌کند که `Joe` یک مدیر است و به صورت داخلی متد `Lookup` یک شی جدید `Manager` در هیپ می‌سازد، آن را برای `Joe` مقداردهی اولیه کرده و آدرس این شی را برمی‌گرداند. آدرس در متغیر محلی `e` ذخیره می‌شود. نتیجه این عملیات در شکل ۹-۴ آمده است.

توجه کنید که `e` دیگر به اولین شی `Manager` ساخته شده اشاره نمی‌کند. در واقع، چون دیگر متغیری به این شی اشاره نمی‌کند، از کاندیداهای اصلی برای جمع آوری در آینده است که حافظه مصرفی توسط این شی را باز پس می‌گیرد (آزاد می‌کند). خط بعدی کد در `M3`، متد نمونه‌ی غیرمجازی `GetYearsEmployed` از `Employee` را فراخوانی می‌کند. وقتی یک متد نمونه‌ی غیرمجازی فراخوانی می‌شود، کامپایلر JIT، شی نوع متناظر با نوع متغیری که متد را فراخوانی کرده است را پیدا می‌کند. در این مورد، متغیر `e` به عنوان `Employee` تعریف شده است. (اگر نوع `Employee` متد فراخوانی شده را تعریف نکرده باشد، کامپایلر JIT در سلسله مراتب وراثت به سمت `Object` برای یافتن متد حرکت می‌کند. آن می‌تواند این کار را انجام دهد چون هر شی نوع دارای فیلدی است که به شی نوع پایه اشاره می‌کند، این اطلاعات در شکل نیامده است). سپس، کامپایلر JIT ورودی در جدول متد شی نوع که به متد فراخوانی شده اشاره دارد را پیدا کرده، متد را (در صورت نیاز) JIT می‌کند و سپس کد JIT شده را فراخوانی می‌کند. برای بحث ما، بگوییم که متد `GetYearsEmployed` از `Employee` مقدار ۵ را برمی‌گرداند چون `Joe` پنج سال است که در شرکت استخدام شده است. عدد صحیح در متغیر محلی `year` ذخیره می‌شود. نتیجه این عملیات در شکل ۹-۵ آمده است.



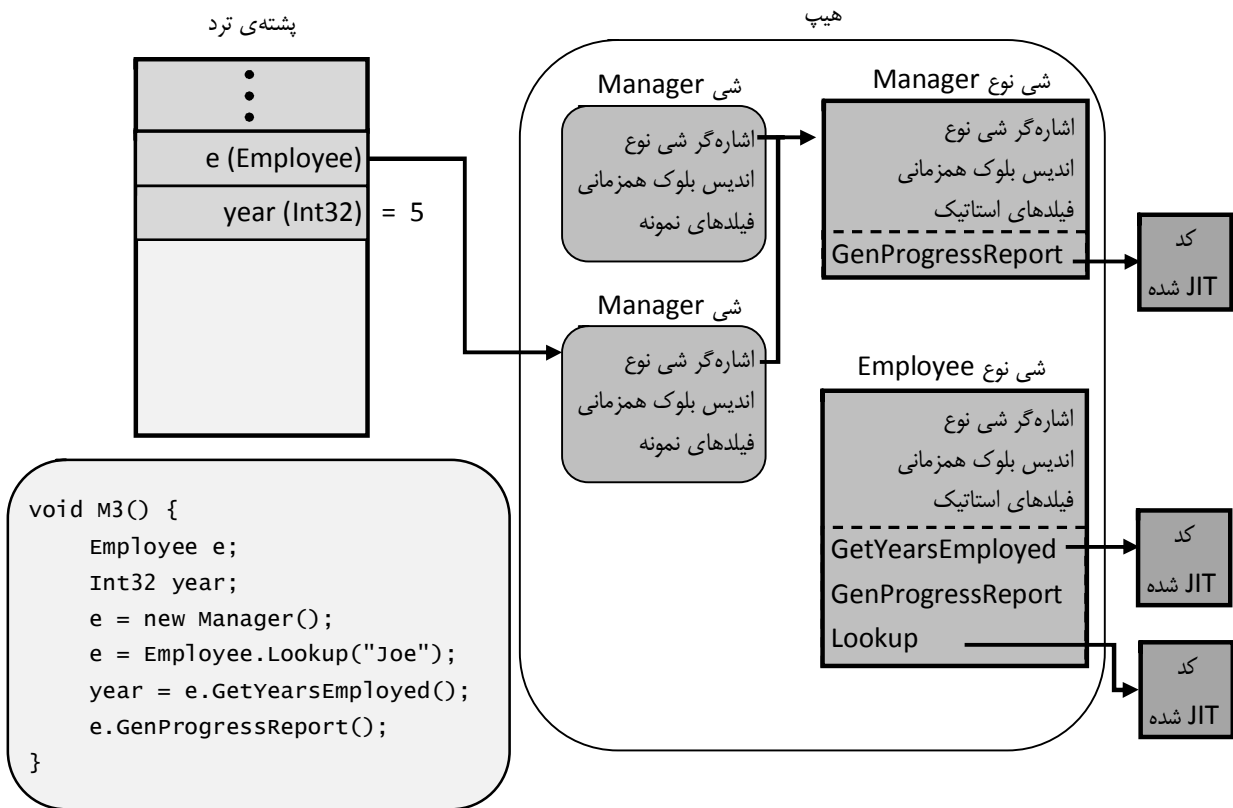
شکل ۱۰-۴ متد استاتیک Lookup از Employee شی Manager را برای Joe تخصیص داده و مقداردهی اولیه می‌کند.



شکل ۱۱-۴ متد نمونه‌ی غیرمجازی **GetYearsEmployed** از **Employee** فراخوانی شده و مقدار 5 را برمی‌گرداند.

خط بعدی کد در **M3** متد مجازی نمونه **GetProgressReport** از **Employee** را فراخوانی می‌کند. هنگام فراخوانی یک متد مجازی نمونه، کامپایلر JIT تعدادی کد اضافی در متد تولید می‌کند که با هر فراخوانی اجرا می‌شوند. این کد ابتدا به متغیری که فراخوانی کرده است و سپس آدرس شی فراخوانی کننده نگاه می‌کند. در این مورد متغیر **e** به شی **Manager** که "Joe" را نشان می‌دهد، اشاره دارد. سپس، کد، عضو درونی اشاره‌گر شی نوع متعلق به شی را بررسی می‌کند. این عضو به نوع واقعی شی اشاره می‌کند. کد، سپس ورودی جدول متد شی نوع که به متد فراخوانی شده اشاره دارد را پیدا کرده، آن را (در صورت نیاز) JIT می‌کند و کد JIT شده را فراخوانی می‌کند. برای بحث ما، پیاده‌سازی‌ای که **Manager** از **GetProgressReport** کرده است، فراخوانی می‌شود، چون **e** به یک شی **Manager** ارجاع می‌کند. نتیجه این عملیات در شکل ۱۲-۴ نشان داده شده است.

نکته اینکه اگر متد **Lookup** از **Employee** تعیین می‌کرد که **Joe** فقط یک **Employee** و نه یک **Manager** است، **Lookup** به صورت داخلی یک شی **Employee** می‌ساخت که اشاره‌گر شی نوع آن به شی نوع **Employee** اشاره کرده و باعث می‌شد پیاده‌سازی‌ای که **Employee** از **GetProgressReport** انجام داده به جای پیاده‌سازی‌ای که **Manager** از آن انجام داده، فراخوانی شود.



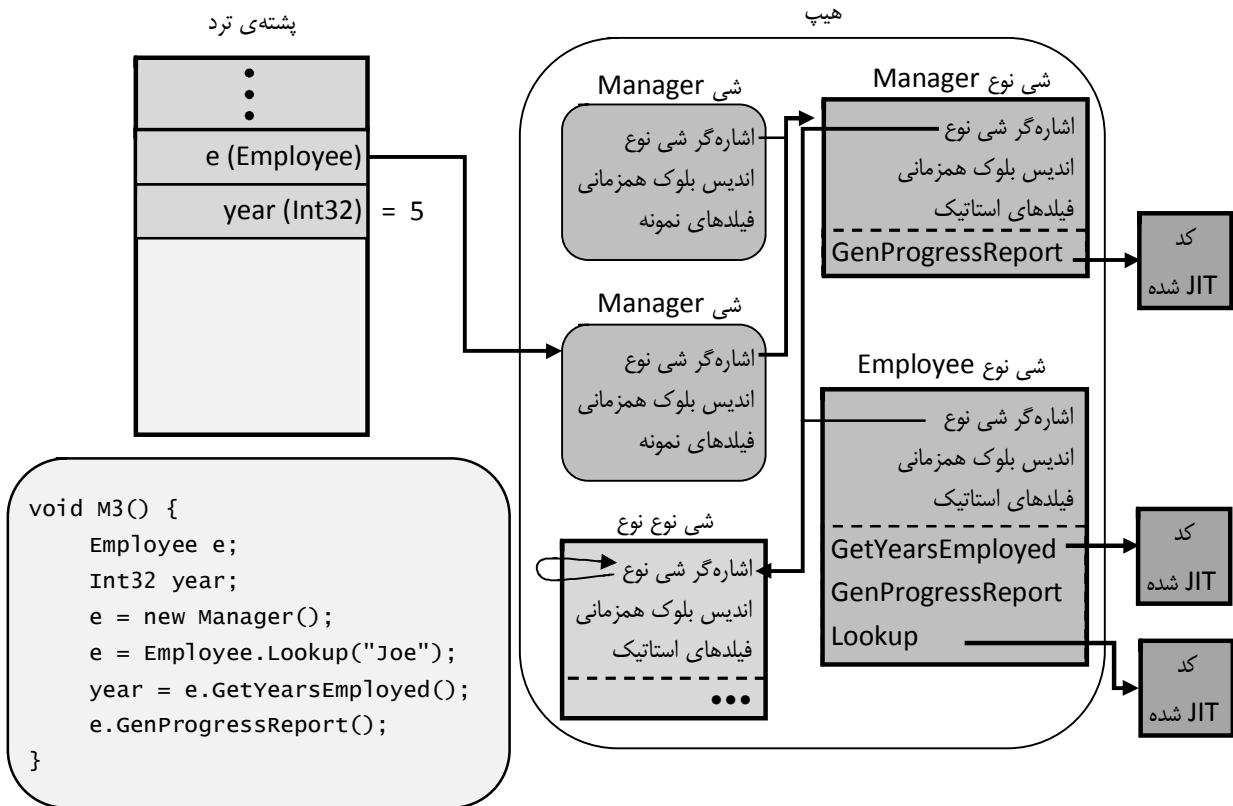
شکل ۱۲-۴ متد مجازی نمونه **GetProgressReport** از **Employee** فراخوانی می‌شود و باعث اجرای نسخه‌ی بازنویسی شده‌ی این متد توسط **Manager** می‌گردد.

در اینجا، ما ارتباط بین سورس کد، IL و کد JIT شده را بحث کردیم. همچنین، پشته‌ی ترد، آرگومان‌ها، متغیرهای محلی و چگونگی ارجاع این آرگومان‌ها و متغیرها به اشیای درون هیپ را نیز صحبت کردیم. همچنین دیدید چگونه اشیاء، اشاره‌گری به شی نوعشان (که شامل فیلدهای استاتیک و جدول متد است) دارند. در ضمن گفتیم چگونه کامپایلر JIT تعیین می‌کند به چه طریق متدهای استاتیک، متدهای غیرمجازی نمونه و متدهای مجازی نمونه را فراخوانی کند. تمام این‌ها باید به شما دید دقیقی از چگونگی عملکرد CLR بدهد و این بینش هنگام معماری و پیاده‌سازی نوع‌ها، کامپوننت‌ها و برنامه‌ها، اتان به شما کمک می‌کند. قبل از پایان فصل، می‌خواهم کمی بینش بیشتری از آنچه درون CLR اتفاق می‌افتد به شما بدهم.

شما متوجه شدید که اشیاء نوع **Employee** و **Manager** هر دو شامل اعضای اشاره‌گر شی نوع هستند. این بدین دلیل است که خود اشیاء نوع، واقعا شی هستند. وقتی CLR، اشیاء نوع را می‌سازد. CLR باید این اعضا را مقداردهی اولیه کند. شاید بپرسید "به چه چیزی؟". خوب وقتی CLR در یک پردازش شروع به اجرا می‌کند، یک شی نوع ویژه برای نوع **System.Type** (که در `MSCorLib.dll` تعریف شده است) می‌سازد. اشیاء نوع **Employee** و

Manager "نمونه" هایی از این نوع هستند و بنابراین، اعضای اشاره‌گر شی نوع در آن‌ها با اشاره به نوع **System.Type** مقداردهی اولیه می‌شوند، درست همانگونه که در شکل ۱۳-۴ آمده است.

البته، شی نوع **System.Type** خودش یک شی است و بنابراین دارای یک عضو اشاره‌گر شی نوع می‌باشد و منطقی است که سوال کنید این عضو به چه چیزی اشاره دارد. این عضو به خودش اشاره دارد، چون شی نوع **System.Type** خود یک "نمونه" از شی نوع است. و اکنون باید سیستم کامل نوع در CLR و نحوه عملکرد آن را بدانید. ضمناً، متد **GetType** از **System.Object** فقط آدرس ذخیره شده در عضو اشاره‌گر شی نوع از شی مشخص شده را برمی‌گرداند. به بیان دیگر متد **GetType**، یک اشاره‌گر به شی نوع از یک شی برمی‌گرداند و این راهیست که شما می‌توانید نوع صحیح هر شی در سیستم (شامل اشیاء نوع) را تعیین کنید.



شکل ۱۳-۴ اشیاء نوع **Employee** و **Manager** نمونه‌هایی از نوع **System.Type** هستند.

پایان فصل ۴

شکر، شکر، شکر، شکر

فصل ۵: نوع های اصلی، ارجاعی و مقداری

در این فصل، گونه های مختلف نوع ها که یک برنامه نویسی دات نت فریمورک میکروسافت با آن ها کار می کند را بحث می کنیم. این برای تمام برنامه نویسان حیاتی است که با رفتارهای متفاوتی که این نوع ها دارند، آشنا باشند. وقتی من اولین بار دات نت فریمورک را یاد گرفتم؛ تفاوت بین نوع های اصلی، ارجاعی و مقداری را کاملاً درک نمی کردم. عدم درک این تمایزها باعث می شد که خطاها و مشکلات عملکردی در کد من بوجود بیاید. با توضیح تفاوت میان نوع ها در اینجا، امیدوارم شما را از سردردهایی که من برای رسیدن به سرعت در برنامه نویسی تجربه کرده ام رها کنم.

نوع های اصلی زبان برنامه نویسی

برخی نوع های داده ای آنقدر فراوان استفاده می شوند که کامپایلرها، نحو ساده شده ای برای دسترسی به این نوع ها فراهم می کنند. برای نمونه شما می توانید یک عدد صحیح را با نحو زیر درست کنید:

```
System.Int32 a = new System.Int32();
```

اما مطمئنم شما هم موافقید تعریف و مقداردهی اولیه یک عدد صحیح با این نحو، نسبتاً سنگین است. خوشبختانه، بسیاری از کامپایلرها (شامل سی شارپ) به شما اجازه می دهند که به جای آن از نحوی شبیه به زیر استفاده کنید:

```
int a = 0;
```

این نحو مطمئناً کد را خواناتر کرده و کد IL ای تولید می کند که معادل کد IL تولیدی وقتی که از **System.Int32** استفاده می شود، است. هر نوع داده ای که کامپایلر به صورت مستقیم پشتیبانی می کند، نوع های اصلی **primitive types** نامیده می شوند. نوع های اصلی مستقیماً بر نوع های موجود در کتابخانه کلاس فریمورک (FCL) منطبق می شوند. برای نمونه در سی شارپ، یک **int** مستقیماً بر نوع **System.Int32** منطبق می گردد. به این خاطر چهار خط کد زیر به درستی کامپایل شده و دقیقاً کد IL یکسانی تولید می کنند:

```
int a = 0; // Most convenient syntax
System.Int32 a = 0; // Convenient syntax
int a = new int(); // Inconvenient syntax
System.Int32 a = new System.Int32(); // Most inconvenient syntax
```

جدول ۱-۵ نوع های FCL که دارای نوع متناظر اصلی در سی شارپ هستند را نشان می دهد. برای نوع هایی که با مشخصات مشترک زبان (CLS) سازگار هستند، دیگر زبان ها نوع های اصلی مشابهی ارائه می کنند. اما، زبان ها نیاز نیست که هیچ گونه پشتیبانی برای نوع های غیرسازگار با CLS فراهم کنند.

جدول ۱-۵ نوع های اصلی سی شارپ با نوع های FCL متناظر

نوع اصلی	نوع FCL	سازگار با CLS	توضیح
sbyte	System.SByte	خیر	مقدار ۸ بیتی علامت دار
byte	System.Byte	بله	مقدار ۸ بیتی بدون علامت
short	System.Int16	بله	مقدار ۱۶ بیتی علامت دار
ushort	System.UInt16	خیر	مقدار ۱۶ بیتی بدون علامت
int	System.Int32	بله	مقدار ۳۲ بیتی علامت دار
uint	System.UInt32	خیر	مقدار ۳۲ بیتی بدون علامت
long	System.Int64	بله	مقدار ۶۴ بیتی علامت دار
ulong	System.UInt64	خیر	مقدار ۶۴ بیتی بدون علامت
char	System.Char	بله	کاراکتر ۱۶ بیتی یونیکد (char) هرگز یک مقدار ۸ بیتی را آنچنان که در C++ مدیریت نشده داشت، نشان نمی دهد.)
float	System.Single	بله	مقدار اعشاری ۳۲ بیتی IEEE
double	System.Double	بله	مقدار اعشاری ۶۴ بیتی IEEE

یک مقدار true یا false	بله	System.Boolean	bool
یک مقدار اعشاری ۱۲۸ بیتی با دقت بالا که برای محاسبات مالی که خطای گرد کردن قابل چشم پوشی نیست کاربرد دارد. از ۱۲۸ بیت، ۱ بیت، علامت مقدار را نشان می‌دهد، ۹۶ بیت، خود مقدار و ۸ بیت، توان ۱۰ که باید مقدار ۹۶ بیت بر آن تقسیم شود را نگهداری می‌کند (هر جا از ۰ تا ۲۸ می‌تواند باشد). بقیه بیت‌ها بدون استفاده هستند.	بله	System.Decimal	decimal
آرایه‌ای از کاراکترها	بله	System.String	string
نوع پایه‌ی همه‌ی نوع‌ها	بله	System.Object	object
برای CLR ، dynamic معادل Object است. هرچند، کامپایلر سی-شارپ اجازه می‌دهد که متغیرهای dynamic در ارسال‌های پویا با نحو ساده استفاده شوند. برای اطلاعات بیشتر بخش "نوع اصلی dynamic " را در پایان این فصل ببینید.	بله	System.Object	dynamic

راه دیگر تفکر پیرامون نوع‌های اصلی این است که کامپایلر سی‌شارپ به صورت خودکار فرض می‌کند شما دستورات **using** (همانطور که در فصل ۴ "مبانی نوع" صحبت شد) زیر را در تمام فایل‌های سورس کد خود وارد کرده اید:

```
using sbyte = System.SByte;
using byte = System.Byte;
using short = System.Int16;
using ushort = System.UInt16;
using int = System.Int32;
using uint = System.UInt32;
...
```

مشخصات زبان سی‌شارپ بیان می‌کند، "به عنوان یک سبک، استفاده از کلمه کلیدی بر استفاده از نام کامل نوع ترجیح دارد". من با مشخصات زبان موافق نیستم. من ترجیح می‌دهم که از نام‌های نوع **FCL** استفاده کرده و کاملاً از نام‌های نوع اصلی خودداری کنم. در واقع، آرزو داشتم که کامپایلرها نام‌های نوع اصلی را ارائه نداده و به جای آن برنامه‌نویسان را مجبور به استفاده از نام‌های نوع **FCL** می‌کردند. دلایل من به این شرح می‌باشند:

- من برنامه‌نویسانی را دیده‌ام که گیج شده‌اند و نمی‌دانند از **string** یا **String** در گذشته استفاده کنند. چون **string** (یک کلمه کلیدی) در سی‌شارپ دقیقاً بر **System.String** (یک نوع **FCL**) منطبق است، تفاوتی بین آن‌ها وجود نداشته و از هر دو می‌توانند استفاده کنند. به طریق مشابه من شنیده‌ام بعضی برنامه‌نویسان می‌گویند که **int** هنگامی که برنامه در سیستم عامل ۳۲ بیتی اجرا می‌شود یک عدد ۳۲ بیتی را نمایش داده و وقتی در سیستم عامل ۶۴ بیتی اجرا شود یک عدد ۶۴ بیتی را نشان می‌دهد. این بیان مطلقاً اشتباه است، در سی‌شارپ یک **int** همیشه بر **System.Int32** منطبق می‌شود و بنابراین بدون توجه به سیستم عاملی که در آن اجرا می‌شود همواره یک عدد ۳۲ بیتی را نشان می‌دهد. اگر برنامه‌نویسان از **Int32** استفاده می‌کردند این سردرگمی احتمالی نیز از بین می‌رفت.
- در سی‌شارپ، **long** بر **System.Int64** منطبق می‌شود اما در یک زبان برنامه‌نویسی متفاوت، **long** می‌تواند بر یک **Int16** یا **Int32** منطبق شود. در واقع، **C++/CLI** با **long** به عنوان یک **Int32** رفتار می‌کند. کسی که به برنامه‌نویسی در یک زبان عادت دارد اگر کدی که به زبان دیگر نوشته شده است را بخواند ممکن است براحتمی آن را اشتباه تفسیر کند. در واقع، اکثر زبان‌ها با **long** به عنوان یک کلمه کلیدی برخورد نمی‌کنند و کدی که از آن استفاده کند را کامپایل نخواهند کرد.
- FCL** دارای متدهای زیادی است که نام نوع به عنوان بخشی از نام متد قرار دارد. برای نمونه، نوع **BinaryReader** متدهای **ReadBoolean**، **ReadInt32**، **ReadSingle** و غیره را ارائه می‌کند و نوع **System.Convert** متدهایی مثل **ToBoolean**، **ToInt32**، **ToSingle** و غیره را ارائه می‌کند. اگرچه نوشتن کد زیر مجاز است اما خط شامل **float** برای من بسیار غیرطبیعی به نظر می‌رسد و واضح نیست که این خط درست باشد:

```
BinaryReader br = new BinaryReader(...);
float val = br.ReadSingle(); // OK, but feels unnatural
```

```
▪ single val = br.ReadSingle(); // OK and feels good
```

▪ بسیاری از برنامه‌نویسانی که فقط از سی‌شارپ استفاده می‌کنند، فراموش می‌کنند که زبان‌های برنامه‌نویسی دیگر را نیز می‌توان برای کار با CLR استفاده کرد و به این خاطر، سی‌شارپ‌یزم در کتابخانه کلاس بوجود آمده است. برای نمونه، FCL مایکروسافت تقریباً منحصر در سی‌شارپ نوشته شده است و برنامه‌نویسان تیم FCL اکنون متدهایی مثل **GetLongLength** از **Array** را معرفی کرده‌اند که یک مقدار **Int64** که در سی‌شارپ **long** است ولی در زبان‌های دیگر (C++/CLI) نیست را برمی‌گرداند. مثال دیگر متد **LongCount** از **System.Linq.Enumerable** است.

به خاطر تمام این دلایل ما من از نوع‌های FCL در این کتاب استفاده می‌کنم. در بسیاری از زبان‌های برنامه‌نویسی، شما انتظار دارید که کد زیر کامپایل شده و به درستی اجرا شود:

```
Int32 i = 5; // A 32-bit value
Int64 l = i; // Implicit cast to a 64-bit value
```

اما، بر طبق مبحث تبدیل‌ها در فصل ۴، شما انتظار نخواهید داشت که این کد کامپایل شود. گذشته از این، **System.Int64** و **System.Int32** نوع‌های متفاوتی بوده و هیچ کدام از دیگری مشتق نشده است. خوب، خوشحال خواهید شد اگر بدانید که کامپایلر سی‌شارپ این کد را به درستی کامپایل کرده و طبق انتظار اجرا می‌کند. چرا؟ علت آن است که کامپایلر سی‌شارپ درباره‌ی نوع‌های اصلی دانش ویژه‌ای دارد و قانون‌های خاص خود را هنگام کامپایل کد اعمال می‌کند. به بیان دیگر، کامپایلر الگوهای برنامه‌نویسی رایج را تشخیص داده و کد IL مورد نیاز را تولید می‌کند تا کد نوشته شده طبق انتظار کار کند. به خصوص، کامپایلر سی‌شارپ از الگوهای مرتبط با تبدیل‌ها، لیترال‌ها و عملگرها همانند مثال زیر پشتیبانی می‌کند.

اول اینکه، کامپایلر قادر به انجام تبدیل‌های ضمنی و صریح بین نوع‌های اصلی است. همانند این:

```
Int32 i = 5; // Implicit cast from Int32 to Int32
Int64 l = i; // Implicit cast from Int32 to Int64
Single s = i; // Implicit cast from Int32 to Single
Byte b = (Byte) i; // Explicit cast from Int32 to Byte
Int16 v = (Int16) s; // Explicit cast from Single to Int16
```

سی‌شارپ اجازه‌ی تبدیل ضمنی را وقتی می‌دهد که تبدیل "امن" باشد، یعنی اینکه از دست رفتن داده ممکن نباشد، مثل تبدیل یک **Int32** به یک **Int64**. اما سی‌شارپ نیاز به تبدیل صریح دارد اگر تبدیل احتمالاً ناامن باشد. برای نوع‌های عددی، "ناامن" یعنی اینکه شما بتوانید دقت یا بزرگی عدد را در نتیجه تبدیل از دست بدهید. برای نمونه تبدیل از **Int32** به **Byte** نیاز به تبدیل صریح دارد چون برای اعداد بزرگ **Int32** دقت ممکن است از دست برود؛ تبدیل از **Single** به **Int16** نیاز به تبدیل صریح دارد چون **Single** می‌تواند اعداد بزرگتری از آنچه **Int16** می‌تواند، را نشان دهد.

بدانید که کامپایلرهای مختلف می‌توانند کدهای مختلفی برای مدیریت این عملیات‌های تبدیل تولید کنند. برای نمونه، هنگام تبدیل یک **Single** با مقدار ۶.۸ به یک **Int32**، بعضی کامپایلرها کدی تولید می‌کنند که ۶ را در **Int32** قرار می‌دهد و بعضی دیگر با گرد کردن به سمت بالا به جواب ۷ می‌رسند. ضمناً، سی‌شارپ نتیجه را کوتاه می‌کند. برای قوانینی که سی‌شارپ برای تبدیل نوع‌های اصلی از آن‌ها پیروی می‌کند به بخش تبدیلات "Conversions" در مشخصات زبان سی‌شارپ مراجعه کنید.

علاوه بر تبدیل، نوع‌های اصلی می‌توانند به عنوان لیترال **literal** نوشته شوند. یک لیترال یک نمونه از نوع خودش در نظر گرفته می‌شود و بنابراین، شما می‌توانید متدهای نمونه را برای آن‌ها همانند زیر استفاده کنید:

```
Console.WriteLine(123.ToString() + 456.ToString()); // "123456"
```

همچنین، اگر عبارتی دارید که از لیترال‌ها تشکیل شده است، کامپایلر قادر است که عبارت را در زمان کامپایل ارزیابی کره و عملکرد برنامه را بهبود بخشد.

```
Boolean found = false; // Generated code sets found to 0
Int32 x = 100 + 20 + 3; // Generated code sets x to 123
String s = "a " + "bc"; // Generated code sets s to "a bc"
```

سرانجام کامپایلر به صورت خودکار می‌داند چگونه و به چه ترتیبی عملگرها (مثل **!**، **~**، **>>**، **<<**، **<=**، **>=**، **<**، **>**، **!**، **==**، **|**، **^**، **&**، **%**، **/**، *****، **+**، **-**) را در کد تفسیر کند:

```
Int32 x = 100; // Assignment operator
Int32 y = x + 23; // Addition and assignment operators
Boolean lessThanFifty = (y < 50); // Less-than and assignment operators
```

عملیات های کنترل شده و کنترل نشده بر روی نوع اصلی

برنامه‌نویسان کاملاً آگاهند که بسیاری از عملیات‌های ریاضی بر روی نوع‌های اصلی منجر به سرریز (overflow) می‌شود:

```
Byte b = 100;
b = (Byte) (b + 200); // b now contains 44 (or 2C in Hex).
```

مهم وقتی عملیات ریاضی فوق انجام می‌شود، در اولین گام نیاز دارد که مقادیر عملوندها به مقادیر ۳۲ بیتی (یا مقادیر ۶۴ بیتی اگر هر یک از عملوندها بیش از ۳۲ بیت نیاز داشته باشند) بسط داده شود. پس **b** و **200** (مقادیری که کمتر از ۳۲ بیت نیاز دارند) ابتدا به مقادیر ۳۲ بیتی تبدیل شده و سپس با هم جمع می‌شوند. نتیجه، یک مقدار ۳۲ بیتی (۳۰۰ در دهدهی و 12C در هگز) است که باید قبل از آنکه در متغیر **b** ذخیره شود به یک **Byte** تبدیل گردد. سی‌شارپ این تبدیل را به صورت ضمنی برای شما انجام نمی‌دهد و به همین دلیل **“(Byte)”** در خط دوم از کد قبلی مورد نیاز است.

در اغلب سناریوهای برنامه‌نویسی، این سرریز بی صدا مطلوب نیست و اگر تشخیص داده نشود باعث می‌شود برنامه به گونه‌ای عجیب و غیر معمول رفتار کند. در برخی سناریوهای نادر برنامه‌نویسی (مثل محاسبه مقدار هس با یک مقدار کنترلی)، این سرریز نه تنها قابل قبول بلکه مطلوب است.

زبان‌های مختلف، سرریز را به روش‌های مختلف مدیریت می‌کنند. C و C++ سرریز را به عنوان خطا در نظر نمی‌گیرند و اجازه می‌دهند عملیات صورت گرفته و برنامه ادامه یابد. ویژوال بیسیک، در سوی دیگر، همیشه سرریز را خطا در نظر گرفته و در صورت رخداد آن یک اکسپشن تولید می‌کند.

CLR، دستورات IL ای ارائه می‌کند که به کامپایلر اجازه می‌دهد رفتار مورد نظر خود را انتخاب کند. CLR دارای دستوری به نام **add** است که دو مقدار را با هم جمع می‌کند. دستور **add** هیچ سرریزی را کنترل نمی‌کند. CLR همچنین دستوری به نام **add.ovf** دارد که دو مقدار را با هم جمع می‌کند. اما **add.ovf** در صورت رخداد سرریز یک **System.OverflowException** تولید می‌کند. علاوه بر این دو دستور IL برای عملیات جمع، CLR دستورات مشابهی برای تفریق (**sub/sub.ovf**)، ضرب (**mul/mul.ovf**) و تبدیل داده (**conv/conv.ovf**) نیز دارد.

سی‌شارپ به برنامه‌نویس اجازه می‌دهد که تصمیم بگیرد چگونه سرریز باید مدیریت شود. به صورت پیش فرض، کنترل سرریز غیرفعال است. این یعنی، کامپایلر کد IL را با نسخه‌ای از دستورات جمع، تفریق، ضرب و تبدیل تولید می‌کند که شامل کنترل سرریز نیستند. در نتیجه‌ی آن، کد سریعتر اجرا می‌شود — اما برنامه‌نویسان باید مطمئن شوند که سرریزها رخ نمی‌دهند یا اینکه کد آن‌ها برای پیش‌بینی این سرریزها طراحی شده است.

یک روش برای آنکه کامپایلر سی‌شارپ سرریزها را کنترل کند آن است که از سوئیچ **+checked** کامپایلر استفاده کنید. این سوئیچ به کامپایلر می‌گوید کدی تولید کند که شامل نسخه‌های کنترل کننده سرریز از دستورات IL جمع، تفریق، ضرب و تبدیل باشد. کد کمی آهسته تر اجرا می‌شود چون CLR این عملیات‌ها را برای تعیین رخداد سرریز، بررسی می‌کند. اگر سرریز رخ دهد، CLR یک **OverflowException** تولید می‌کند.

علاوه بر فعال یا غیرفعال بودن سراسری کنترل سرریز، برنامه‌نویسان می‌توانند بررسی سرریز را در نواحی خاصی از کدشان کنترل کنند. سی‌شارپ این انعطاف پذیری را با عملگرهای **checked** و **unchecked** فراهم می‌کند. یک نمونه که از عملگر **unchecked** استفاده می‌کند:

```
UInt32 invalid = unchecked((UInt32) (-1)); // OK
```

و مثال دیگری که از عملگر **checked** استفاده می‌کند:

```
Byte b = 100;
b = checked((Byte) (b + 200)); // overflowException is thrown
```

در این مثال، ابتدا **b** و **200** به مقادیر ۳۲ بیتی تبدیل شده و سپس با هم جمع می‌شوند که جواب ۳۰۰ می‌شود. سپس ۳۰۰ به خاطر وجود تبدیل، به یک بایت تبدیل می‌شود که این عمل یک **OverflowException** تولید می‌کند. اگر **Byte** بیرون از عملگر **checked** بود، اکسپشن رخ نمی‌داد:

```
b = (Byte) checked(b + 200); // b contains 44; no OverflowException
```

علاوه بر عملگرهای **checked** و **unchecked**، سی‌شارپ همچنین عبارتهای **checked** و **unchecked** را ارائه می‌کند. عبارتهای باعث می‌شوند که تمام دستورات درون یک بلوک کنترل شوند یا کنترل نشوند:

```
checked { // start of checked block
    byte b = 100;
    b = (Byte) (b + 200); // This expression is checked for overflow.
} // End of checked block
```

در واقع، اگر از یک بلوک عبارت **checked** استفاده کنید می‌توانید عملگر **+=** را با **Byte** به کار برید که کمی کد را ساده تر می‌کند:

```
checked {
    // Start of checked block
    byte b = 100;
    b += 200; // This expression is checked for overflow.
} // End of checked block
```

مهم چون تنها اثری که عملگر و عبارت **checked** دارد این است که تعیین می‌کند چه نسخه‌ای از دستورات **IL** جمع، تفریق، ضرب و تبدیل داده تولید شوند، فراخوانی یک متد درون عملگر یا عبارت **checked** اثری بر متد ندارد؛ همانند نمونه کد زیر:

```
checked {
    // Assume SomeMethod tries to load 400 into a Byte.
    SomeMethod(400);
    // SomeMethod might or might not throw an OverflowException.
    // It would if SomeMethod were compiled with checked instructions.
}
```

در تجربه خودم، محاسباتی زیادی را دیده‌ام که نتایج تعجب‌آوری تولید می‌کنند. معمولاً، این به خاطر ورودی اشتباه کاربر است، اما می‌تواند به خاطر مقادیر برگشتی از بخش‌هایی از سیستم که برنامه‌نویس انتظار ندارد، باشد. و بنابراین، من برنامه‌نویسان را به انجام کارهای زیر توصیه می‌کنم:

- هر جا ممکن است به جای نوع‌های داده‌ای بدون علامت (مثل **UInt32** و **UInt64**) از نوع‌های داده‌ای علامت دار (مثل **Int32** و **Int64**) استفاده کنید. این به کامپایلر اجازه می‌دهد تا خطاهای سرریز/ زیرریز (**underflow**) بیشتری را تشخیص دهد. به علاوه، بخش‌های مختلف کتابخانه کلاس (مثل ویژگی **Length** از **String** و **Array**) به گونه‌ای نوشته شده‌اند که مقادیر علامت دار بازگرداندند تا هنگام استفاده از این مقادیر در کدتان به تبدیل کمتری نیاز داشته باشید. تبدیلات کمتر کد شما را تمیزتر کرده و نگهداری آن را آسان می‌سازد. افزون بر این، نوع‌های عددی بدون علامت، منطبق بر **CLS** نیستند.
- هنگام کدنویسی، اطراف بلوک‌هایی که احتمال رخداد سرریز به خاطر ورودی اشتباه کاربر مثل پردازش یک درخواست از کاربر نهایی یا یک ماشین مشتری وجود دارد، از **checked** استفاده کنید شاید بخواهید **OverflowException** را هم بگیرید تا برنامه‌تان به آرامی از این شکست‌ها احیا شود.
- هنگام کدنویسی، اطراف بلوک‌هایی که رخداد سرریز مشکل ندارد، مثل محاسبه‌ی یک مقدار کنترلی (**checksum**)، صریحاً از **unchecked** استفاده کنید.
- برای هر کدی که از **checked** و **unchecked** استفاده نمی‌کنید، فرض بر آنست که می‌خواهید هنگام سرریز، اکسپشن تولید شود، برای نمونه در محاسبه‌ی چیزی (مثل اعداد اول) که ورودی‌ها شناخته شده‌اند و سرریز یک باگ است.

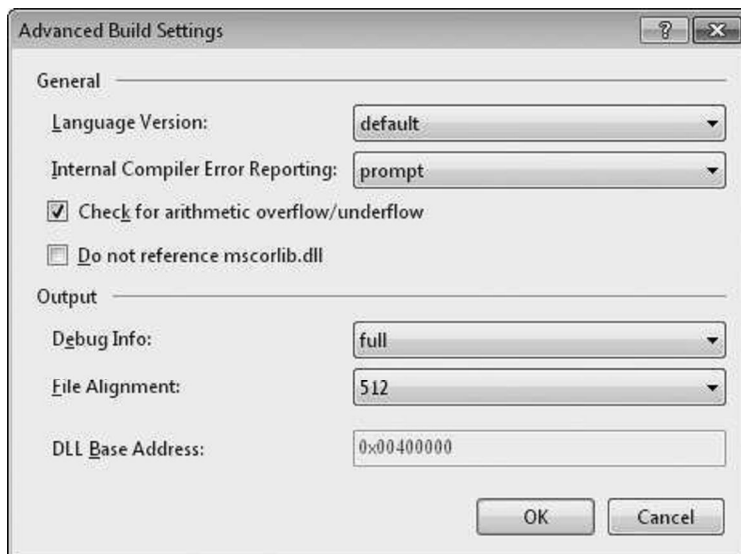
اکنون، هنگام توسعه‌ی برنامه‌ی خود برای ساخت‌های خطایابی^{۳۶} **checked+ /** کامپایلر را روشن کنید. برنامه شما بسیار آهسته تر اجرا می‌شود، چون سیستم هر کدی که صریحاً با **checked** یا **unchecked** معلوم نشده باشد را برای کنترل سرریز بررسی می‌کند. اگر اکسپشن رخ دهد، به سادگی آن را پیدا کرده و می‌توانید در کد خود آنرا برطرف کنید. برای ساخت‌های نهایی^{۳۷} (آماده‌ی پخش و توزیع) برنامه‌ی خود، از سویج **checked- /** کامپایلر استفاده کنید تا کد شما سریعتر اجرا شده و اکسپشن‌های سرریز تولید نشود. برای تغییر تنظیمات **Checked** در ویژوال استودیو، **Properties** پروژه‌ی خود را باز کرده، تب **Build** را انتخاب و **Advanced** را کلیک کنید و سپس گزینه‌ی **“Check For Arithmetic Overflow/underflow”** را همانند شکل ۵-۱ تیک بزنید.

اگر برنامه شما می‌تواند کاهش سرعت جزئی ناشی از عملیات‌های کنترل شده را تحمل کند، پس به شما توصیه می‌کنم که حتی برای ساخت نهایی از سویج **checked /** استفاده کنید چرا که مانع ادامه‌ی اجرای برنامه شما با داده‌های خراب و حفره‌های امنیتی احتمالی می‌شود. برای نمونه، شاید شما برای

³⁶ debug build

³⁷ release build

محاسبه‌ی اندیس یک آرایه ضربی انجام دهید؛ بسیار بهتر است که به جای دسترسی به عنصر اشتباه از آرایه به خاطر گردش اندیس، یک **OverflowException** دریافت کنید.



شکل ۱-۵ تغییر تنظیمات پیش فرض کامپایلر برای انجام عملیات کنترل شده ریاضی توسط پنجره‌ی **Advanced Build Settings** در ویژوال استودیو

مهم نوع **System.Decimal** نوعی بسیار خاصی است. اگرچه بسیاری از زبان‌های برنامه‌نویسی (شامل سی‌شارپ و ویژوال بیسیک)، **Decimal** را یک نوع اصلی در نظر می‌گیرند، CLR این کار را نمی‌کند. این یعنی CLR، دستورات IL برای دستکاری یک مقدار **Decimal** ندارد. اگر نوع **Decimal** را در SDK دات‌نت فریمورک نگاه کنید، می‌بینید دارای متدهای استاتیک عمومی به نام **Add**، **Subtract**، **Multiply**، **Divide** و غیره است. به علاوه، نوع **Decimal** متدهای سربارگذاری عملگرها برای **+**، **-**، *****، **/** و غیره را ارائه می‌کند.

وقتی کدی که از نوع **Decimal** استفاده می‌کند را کامپایل می‌کنید، کامپایلر کد لازم برای فراخوانی اعضای **Decimal** را به منظور انجام عمل واقعی تولید می‌کند. این یعنی کار با **Decimal** کندتر از کار با نوع‌های اصلی CLR است. همچنین، چون دستور IL ای برای دستکاری مقادیر **Decimal** وجود ندارد، عملگرها و عبارتهای **checked** و **unchecked** و سویچ‌های کامپایلر اثری ندارند. عملیات بر روی **Decimal** در صورتی که نتواند به صورت امن انجام شود همیشه یک **OverflowException** تولید می‌کند.

به طریق مشابه، نوع **System.Numerics.BigInteger** هم نوع خاصی است چون به صورت داخلی از آرایه‌ای از **UInt32** ها برای نمایش یک عدد بزرگ دلخواه که مقدارش حد بالا یا پایینی ندارد، استفاده می‌کند. بنابراین، عملیات بر روی یک **BigInteger** هرگز منجر به یک **OverflowException** نمی‌شود. هرچند، یک عمل **BigInteger** ممکن است یک **OutOfMemoryException** تولید کند اگر مقدار خیلی بزرگ شده و حافظه‌ی کافی برای تغییر اندازه آرایه موجود نباشد.

نوع‌های ارجاعی و نوع‌های مقداری

CLR دو گونه از نوع‌ها را پشتیبانی می‌کند: نوع‌های ارجاعی **reference types** و نوع‌های مقداری **value types**. در حالیکه اکثر نوع‌ها در FCL نوع‌های ارجاعی هستند، نوع‌هایی که برنامه‌نویسان اغلب استفاده می‌کنند، نوع‌های مقداری هستند. نوع‌های ارجاعی همیشه از هیپ مدیریت شده اختصاص داده می‌شوند و عملگر **new** سی‌شارپ آدرس حافظه‌ی شی - آدرس حافظه‌ای که به بیت‌های شی اشاره می‌کند - را بر می‌گرداند. هنگام استفاده از نوع‌های ارجاعی باید برخی ملاحظات عملکردی را در ذهن داشته باشید. اول به این حقایق توجه کنید:

- حافظه باید از هیپ مدیریت شده تخصیص داده شود.
- هر شی تخصیص داده شده در هیپ دارای اعضای اضافی همراه آن است که باید مقداردهی اولیه شوند.

- دیگر بایت‌های شی (برای فیلدها) همیشه صفر می‌شوند.
- اختصاص یک شی از هیپ مدیریت شده می‌تواند باعث یک جمع آوری زباله شود.

اگر هر نوعی یک نوع ارجاعی بود، کارایی برنامه به شدت کم می‌شد، تصور کنید اگر با هر استفاده از یک مقدار **Int32**، یک تخصیص حافظه رخ می‌داد، چقدر کارایی پایین می‌آمد. به منظور بهبود کارایی، CLR برای نوع‌های ساده و غالباً مورد استفاده، نوع‌های سبکی به نام نوع‌های مقداری را ارائه می‌کند. نمونه‌های نوع مقداری معمولاً در پشت‌پرده ترد اختصاص داده می‌شوند (هرچند آن‌ها می‌توانند به عنوان یک فیلد در یک شی از نوع ارجاعی، جاسازی شوند). متغیری که نمونه را نمایش می‌دهد، حاوی یک اشاره‌گر به یک نمونه نیست، متغیر، حاوی فیلدهای خود نمونه است. چون متغیر حاوی فیلدهای نمونه است، برای دسترسی به فیلدهای نمونه، نیاز به دنبال کردن یک اشاره‌گر نیست. نمونه‌های نوع مقداری تحت کنترل جمع آوری کننده‌ی زباله نبوده، پس استفاده از آن‌ها، بار هیپ مدیریت شده را کاهش می‌دهد و تعداد جمع آوری‌ها در طول یک برنامه را نیز کم می‌کند.

SDK دات‌نت فریم‌ورک به وضوح بیان می‌کند کدام نوع‌ها، نوع‌های ارجاعی و کدام نوع‌ها مقداری هستند. هنگامی که یک نوع را در SDK دات‌نت نگاه می‌کنید، هر نوعی که یک کلاس **class** نامیده شود، یک نوع ارجاعی است. برای نمونه کلاس **System.Exception**، کلاس **System.IO.FileStream**، و کلاس **System.Random**، همگی نوع‌های ارجاعی هستند. در سوی دیگر، این مستندات، از هر **structure** یا **enumeration** به عنوان یک نوع مقداری یاد می‌کند. برای نمونه، ساختار **System.Int32**^{۳۸}، ساختار **System.Boolean**، ساختار **System.Decimal**، ساختار **System.TimeSpan**، شمارشی^{۳۹} **System.DayOfWeek**، شمارشی **System.IO.FileAttributes** و شمارشی **System.Drawing.FontStyle** همه نوع‌های مقداری هستند.

اگر دقیق‌تر به مستندات نگاه کنید، متوجه خواهید شد که تمام ساختارها، بلافاصله از نوع خلاصه^{۴۰} **System.ValueType** مشتق می‌شوند. خود **System.ValueType** بلافاصله از نوع **System.Object** مشتق می‌شود. طبق تعریف، تمام نوع‌های مقداری باید از **System.ValueType** مشتق شوند. تمام شمارشی‌ها از نوع مطلق **System.Enum** مشتق می‌شوند که خود آن از **System.ValueType** مشتق می‌شود. CLR و تمام زبان‌های برنامه‌نویسی با شمارشی‌ها به گونه‌ای خاص رفتار می‌کنند. برای اطلاعات بیشتر درباره‌ی نوع‌های شمارشی به فصل ۱۵ "نوع‌های شمارشی و پرچم‌های بیتی" مراجعه کنید.

اگرچه شما نمی‌توانید هنگام تعریف نوع مقداری خود، یک نوع پایه انتخاب کنید، اگر بخواهید یک نوع مقداری می‌تواند یک یا بیشتر رابط^{۴۱} را پیاده‌سازی کند. به علاوه، تمام نوع‌های مقداری مهر شده اند^{۴۲} که مانع از آن می‌شوند که یک نوع مقداری به عنوان نوع پایه برای هر نوع ارجاعی یا مقداری دیگر استفاده شود. پس برای نمونه، ممکن نیست نوعی تعریف کنید که از **Char**، **Boolean**، **Int32**، **UInt64**، **Single**، **Double**، **Decimal** و غیره به عنوان نوع پایه استفاده کند.

مهم برای بسیاری از برنامه‌نویسان (مثل برنامه‌نویسان C/C++ مدیریت نشده)، نوع‌های ارجاعی و نوع‌های مقداری در ابتدا عجیب به نظر می‌رسند. در C/C++ مدیریت نشده، شما یک نوع تعریف می‌کنید و سپس کدی که از نوع استفاده می‌کند، باید تصمیم بگیرد که یک نمونه از نوع را در پشت‌پرده ترد یا هیپ برنامه اختصاص دهد. در کد مدیریت شده، برنامه‌نویسی که نوع را تعریف کرده تعیین می‌کند که نمونه‌های نوع در کجا اختصاص داده شوند، در نتیجه برنامه‌نویسی که از نوع استفاده می‌کند، کنترلی بر این مورد ندارد.

کد زیر و شکل ۲-۵ تفاوت میان نوع‌های ارجاعی و نوع‌های مقداری را نشان می‌دهد.

```
// Reference type (because of 'class')
class SomeRef { public Int32 x; }
```

```
// Value type (because of 'struct')
struct SomeVal { public Int32 x; }
```

```
static void valueTypeDemo() {
```

```

38 structure
39 enumeration
40 abstract
41 interface
42 sealed
```



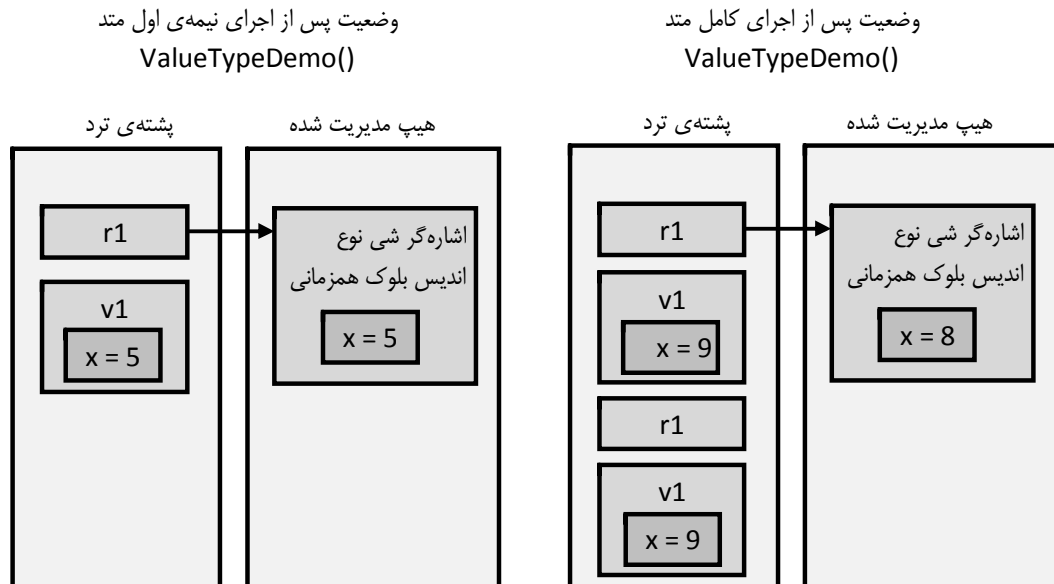
```

SomeRef r1 = new SomeRef(); // Allocated in heap
SomeVal v1 = new SomeVal(); // Allocated on stack
r1.x = 5; // Pointer dereference
v1.x = 5; // Changed on stack
Console.WriteLine(r1.x); // Displays "5"
Console.WriteLine(v1.x); // Also displays "5"
// The left side of Figure 5-2 reflects the situation
// after the lines above have executed.

SomeRef r2 = r1; // Copies reference (pointer) only
SomeVal v2 = v1; // Allocate on stack & copies members
r1.x = 8; // Changes r1.x and r2.x
v1.x = 9; // Changes v1.x, not v2.x
Console.WriteLine(r1.x); // Displays "8"
Console.WriteLine(r2.x); // Displays "8"
Console.WriteLine(v1.x); // Displays "9"
Console.WriteLine(v2.x); // Displays "5"
// The right side of Figure 5-2 reflects the situation
// after ALL of the lines above have executed.
}

```

در این کد، نوع **SomeVale** به عنوان **struct** به جای گونه‌ی رایج تر **class** تعریف شده است. در سی‌شارپ، نوع‌هایی که با **struct** تعریف می‌شوند، نوع‌های مقداری هستند و نوع‌هایی که با **class** تعریف می‌شوند، نوع‌های ارجاعی هستند. همانطور که می‌بینید، رفتار نوع‌های ارجاعی با نوع‌های مقداری بسیار تفاوت دارد. هنگام استفاده از نوع‌ها در کد خود، باید بدانید که نوع یک نوع مقداری یا یک نوع ارجاعی است چون نحوه‌ی آنچه می‌خواهید پیاده‌سازی کنید را بسیار تحت تاثیر قرار می‌دهد.



شکل ۲-۵ نمایش حافظه هنگام اجرای کد

در کد قبلی، شما این خط را دیدید:

```
SomeVal v1 = new SomeVal(); // Allocated on stack
```

نحوه‌ی نوشتن این خط به گونه‌ایست که به نظر می‌رسد یک نمونه‌ی **SomeVal** در هیپ مدیریت شده اختصاص می‌یابد. اما، کامپایلر سی‌شارپ می‌داند که **SomeVal** یک نوع مقداری است و کدی تولید می‌کند که نمونه‌ی **SomeVal** در پشت‌پشته‌ی ترد اختصاص می‌یابد. همچنین سی‌شارپ مطمئن می‌شود که تمام فیلدها در نمونه‌ی نوع مقدار صفر شوند.

خط قبلی می‌توانست شبیه این خط نوشته شود:

```
SomeVal v1; // Allocated on stack
```

این خط کد IL ای تولید می‌کند که نمونه را در پشت‌پشته‌ی ترد اختصاص داده و فیلدهای آن را صفر کند. تنها فرق این است که اگر شما از عملگر **new** استفاده کنید سی‌شارپ "فکر می‌کند" نمونه، مقداری دهی اولیه شده است. کد زیر این نکته را روشن می‌کند:

```
// These two lines compile because C# thinks that
```

```
// v1's fields have been initialized to 0.
```

```
SomeVal v1 = new SomeVal();
```

```
Int32 a = v1.x;
```

```
// These two lines don't compile because C# doesn't think that
```

```
// v1's fields have been initialized to 0.
```

```
SomeVal v1;
```

```
Int32 a = v1.x; // error CS0170: Use of possibly unassigned field 'x'
```

هنگام طراحی نوع‌های خود، به این نکته که آیا نوع‌های خود را به عنوان نوع‌های مقداری به جای نوع‌های ارجاعی تعریف کنید، توجه داشته باشید. در برخی موارد، نوع‌های مقداری، کارایی بهتری دارند. به خصوص، در صورتی که تمام شرایط زیر برقرار باشد، نوع را باید نوع مقداری تعریف کنید:

- نوع به عنوان نوع اصلی عمل می‌کند، به ویژه، این یعنی، آن یک نوع خیلی ساده بدون هیچ عضوی که مقادیر فیلدهایش را تغییر دهد، است. وقتی یک نوع هیچ عضوی برای تغییر فیلدهایش ارائه نکند، می‌گوییم که نوع تغییرناپذیر **immutable** است. در واقع توصیه می‌شود که بسیاری از نوع‌های مقداری تمام فیلدهای خود را با **readonly** (که در فصل ۷ "ثابت‌ها و فیلدها" بحث می‌شود) علامت بزنند.
- نوع، نیازی به ارث بری از نوعی دیگر ندارد.
- نوع، هیچ نوع دیگری که از آن مشتق شود، ندارد.

اندازه‌ی نمونه‌های نوع شما نیز عاملی است که باید لحاظ شود. چون طبق پیش فرض، آرگومان‌ها با مقدار ارسال می‌شوند که باعث کپی شدن فیلدها در نمونه‌های نوع‌های مقداری می‌شود و منجر به کاهش کارایی می‌گردد. پس علاوه بر شرایط قبلی، در صورت صحت شرایط زیر، نوع خود را نوع مقداری تعریف کنید:

- نمونه‌های نوع کوچک هستند (تقریباً ۱۶ بایت یا کمتر).
- نمونه‌های نوع بزرگ (بزرگتر از ۱۶ بایت) هستند و به عنوان پارامتر متدها یا مقدار برگشتی متدها استفاده نمی‌شوند.
- مزیت اصلی نوع‌های مقداری آن است که به عنوان شی در هیپ مدیریت شده اختصاص داده نمی‌شوند. البته، نوع‌های مقداری در قیاس با نوع‌های ارجاعی، محدودیت‌های خود را دارند. برخی از تفاوت‌های نوع‌های مقداری و ارجاعی عبارتند از:
 - اشیاء نوع مقداری دو نمایش دارند: یک فرم بسته‌بندی نشده و یک فرم بسته‌بندی شده (که در بخش بعدی توضیح داده می‌شود). نوع‌های ارجاعی همیشه در فرم بسته‌بندی شده هستند.
 - نوع‌های مقداری از **System.ValueType** مشتق می‌شوند. این نوع همان متدهای تعریف شده توسط **System.Object** را ارائه می‌کند. هر چند، **System.ValueType** متد **Equals** را بازنویسی کرده و در صورت برابری فیلدهای دو شی، **true** برمی‌گرداند. به علاوه، **System.ValueType** متد **GetHashCode** را برای تولید کد هش به کمک یک الگوریتم که مقادیر فیلدهای نمونه‌ی شی را در محاسبه کد هش دخیل می‌کند، بازنویسی می‌کند. به خاطر کارایی ضعیف این متدهای پیش فرض، هنگام تعریف نوع مقداری خود، باید صریحاً متدهای **Equals** و **GetHashCode** را بازنویسی کنید. متدهای **Equals** و **GetHashCode** را در پایان این فصل توضیح می‌دهم.
 - چون شما نمی‌توانید یک نوع مقداری جدید یا یک نوع ارجاعی جدید با استفاده از یک نوع مقداری به عنوان نوع پایه، تعریف کنید، نباید هیچ متد مجازی جدیدی در یک نوع مقداری تعریف کنید. هیچ متدی نمی‌تواند خلاصه (**abstract**) باشد و تمام متدها به صورت ضمنی مهر شده‌اند (قابل بازنویسی نیستند).

- متغیرهای نوع ارجاعی حاوی آدرس حافظه‌ی اشیاء در هیپ می‌باشند. به صورت پیش فرض، وقتی که متغیر نوع ارجاعی ساخته می‌شود، به **null** مقداردهی اولیه می‌شود که بیان می‌کند متغیر نوع ارجاعی اکنون به یک شی معتبر اشاره ندارد. سعی در استفاده از یک متغیر نوع ارجاعی **null** منجر به تولید **NullReferenceException** می‌گردد. در مقابل، متغیرهای نوع مقداری همواره دارای مقداری از نوعشان هستند و تمام اعضای نوع مقداری با صفر مقداردهی اولیه می‌شوند. چون یک متغیر نوع مقداری یک اشاره‌گر نیست، هنگام دسترسی به نوع مقداری امکان تولید **NullReferenceException** وجود ندارد. CLR ویژگی خاصی ارائه می‌کند که قابلیت تهی بودن را برای یک نوع مقداری فراهم می‌کند. این ویژگی، نوع‌های تهی پذیر **nullable types** نامیده می‌شود که در فصل ۱۹ "نوع‌های مقداری تهی پذیر" بحث می‌شود.
- هنگامیکه یک متغیر نوع مقداری را به متغیر نوع مقداری دیگری نسبت می‌دهید یک کپی فیلد به فیلد انجام می‌شود. هنگام انتساب یک متغیر نوع ارجاعی به متغیر نوع ارجاعی دیگر، تنها آدرس حافظه کپی می‌شود.
- به دلیل نکته قبلی، دو یا بیشتر متغیر نوع ارجاعی می‌توانند به یک شی در هیپ اشاره کنند که اجازه می‌دهد عملیات‌های یک متغیر، بر شی ارجاع داده شده توسط دیگر متغیرها اثر گذارد. در سوی دیگر، متغیرهای نوع مقداری اشیاء مجزایی هستند و اثرگذاری عملیات یک متغیر نوع مقداری بر دیگری غیرممکن است.
- چون نوع‌های مقداری بسته‌بندی نشده، در هیپ تخصیص نمی‌یابند، فضای تخصیص داده شده به آن‌ها به محض غیرفعال شدن متدی که آن‌ها را تعریف کرده، آزاد می‌شود. این یعنی یک نمونه‌ی نوع مقداری، خبری (از طریق متد **Finalize**) هنگام بازپس‌گیری حافظه‌اش دریافت نمی‌کند.

نکته در واقع، تعریف یک نوع مقداری با متد **Finalize** خیلی عجیب و غریب است چون متد فقط بر روی نمونه‌های بسته‌بندی شده فراخوانی می‌شود. به همین دلیل، بسیاری از کامپایلرهای (شامل سی‌شارپ، C++/CLI و ویژوال بیسیک) اجازه‌ی تعریف متد **Finalize** برای یک نوع مقداری را نمی‌دهند. اگرچه CLR اجازه‌ی تعریف متد **Finalize** برای یک نوع مقداری را می‌دهد، اما CLR این متد را هنگام جمع‌آوری یک نمونه بسته‌بندی نشده از نوع مقداری فراخوانی نخواهد کرد.

چگونه CLR ترتیب فیلدهای یک نوع را کنترل می کند

برای بهبود کارایی، CLR قادر است فیلدهای یک نوع را به هر صورتی که بخواهد مرتب کند. برای نمونه ممکن است CLR فیلدها را در حافظه به گونه‌ای مرتب کند که ارجاعات دسته‌بندی شده و فیلدها کنار هم مرتب شوند. هرچند وقتی شما یک نوع تعریف می‌کنید، می‌توانید به CLR بگویید که فیلدهای نوع به همان ترتیبی که برنامه‌نویس تعیین کرده بماند یا می‌تواند آن‌ها را برطبق مصلحت مرتب کند.

شما برای آنکه به CLR بگویید چه کاری انجام دهد، صفت `System.Runtime.InteropServices.StructLayoutAttribute` را بر کلاس یا ساختاری که تعریف می‌کنید، اعمال می‌نمایید. برای سازنده‌ی این صفت، `LayoutKind.Auto` را ارسال کنید تا CLR فیلدها را مرتب کند، `LayoutKind.Sequential` تا CLR ترتیب فیلدهای شما را نگه دارد یا `LayoutKind.Explicit` تا صریحاً فیلدها را با آفست در حافظه مرتب کنید. اگر شما صریحاً `StructLayoutAttribute` را برای یک نوع که تعریف می‌کنید، تعیین نکنید، کامپایلر شما بهترین ترتیب را انتخاب می‌کند.

باید بدانید که کامپایلر سی‌شارپ مایکروسافت `LayoutKind.Auto` را برای نوع‌های ارجاعی (کلاس‌ها) و `LayoutKind.Sequential` را برای نوع‌های مقداری (ساختارها) انتخاب می‌کند. واضح است که تیم کامپایلر سی‌شارپ باور داشته‌اند که ساختارها اغلب در ارتباط با کد مدیریت نشده استفاده می‌شوند و برای عملی شدن آن، فیلدها باید به همان ترتیبی باشند که برنامه‌نویس تعریف کرده است. اما اگر یک نوع مقداری می‌سازید که ارتباطی با کد مدیریت نشده ندارد، شما احتمالاً بخواهید که پیش‌فرض کامپایلر سی‌شارپ را تغییر دهید. یک نمونه ببینیم:

```
using System;
using System.Runtime.InteropServices;

// Let the CLR arrange the fields to improve
// performance for this value type.
[StructLayout(LayoutKind.Auto)]
internal struct SomeValueType {
    private readonly Byte m_b;
    private readonly Int16 m_x;
    ...
}
```

`StructLayoutAttribute` همچنین به شما اجازه می‌دهد که صریحاً آفست هر فیلدی را با ارسال `LayoutKind.Explicit` `System.Runtime.InteropServices.FieldOffsetAttribute` به سازنده‌اش، تعیین کنید. سپس یک نمونه از صفت `Int32` به عنوان آفست (به بایت) اولین فیلد از نقطه‌ی شروع نمونه، به سازنده‌ی صفت می‌فرستید. را به هر فیلد اعمال کرده و یک `Int32` به عنوان آفست (به بایت) اولین فیلد از نقطه‌ی شروع نمونه، به سازنده‌ی صفت می‌فرستید. ترتیب صریح اغلب برای شبیه‌سازی چیزی شبیه به یک اتحاد `union` در `C/C++` مدیریت نشده، استفاده می‌شود چون می‌توانید چندین نوع فیلد داشته باشید که از یک آفست در حافظه شروع شوند. یک نمونه در اینجا آمده است:

```
using System;
using System.Runtime.InteropServices;

// The developer explicitly arranges the fields of this value type.
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValueType {
    [FieldOffset(0)]
    private readonly Byte m_b; // The m_b and m_x fields overlap each
```

```
[FieldOffset(0)]
private readonly Int16 m_x; // other in instances of this type
}
```

باید یادآوری شود که تعریف نوعی که در آن یک نوع ارجاعی با یک نوع مقداری هم پوشانی دارند، غیرمجاز است. تعریف نوعی که در آن، چند نوع ارجاعی در آفست شروع یکسان هم پوشانی دارند ممکن است اما این نوع، دیگر قابل بازبینی و بررسی نیست. تعریف نوعی که در آن چند نوع مقداری هم پوشانی دارند، مجاز است؛ هرچند تمام بایت‌های هم پوشانی باید توسط فیلدهای عمومی دسترس پذیر باشند تا نوع قابل بازبینی و بررسی باشد.

بسته بندی و باز کردن نوع های مقداری

نوع های مقداری از نوع های ارجاعی سبکتر هستند چون به عنوان شی در هیپ مدیریت شده، تخصیص نمی یابند، جمع آوری نمی شوند و توسط اشاره گرها ارجاع نمی شوند. هرچند، در بسیاری موارد، شما یک ارجاع به یک نمونه از یک نوع مقداری دریافت می کنید. برای نمونه، فرض کنیم شما می خواهید یک شی **ArrayList** (یک نوع تعریف شده در فضای نام **System.Collections**) برای نگهداری مجموعه ای از ساختارهای **Point** بسازید. کد شبیه به این خواهد بود:

```
// Declare a value type.
struct Point {
public Int32 x, y;
}

public sealed class Program {
public static void Main() {
ArrayList a = new ArrayList();
Point p; // Allocate a Point (not in the heap).
for (Int32 i = 0; i < 10; i++) {
p.x = p.y = i; // Initialize the members in the value type.
a.Add(p); // Box the value type and add the
// reference to the ArrayList.
}
...
}
}
```

با هر بار اجرای حلقه، فیلدهای یک نوع مقداری **Point** مقداردهی اولیه می شوند، سپس **Point** در **ArrayList** ذخیره می شود. اما بگذارید کمی درباره سی این فکر کنیم. واقعا چه چیزی در **ArrayList** ذخیره می شود؟ ساختار **Point**، آدرس ساختار **Point** یا کلا چیز دیگری؟ برای رسیدن به جواب باید متد **Add** از **ArrayList** را نگاه کنید و ببینید پارامترش از چه نوعی تعریف شده است. در این مورد، متد **Add** به شکل زیر است:

```
public virtual Int32 Add(Object value);
```

از روی این، به وضوح می بینید که **Add** یک **Object** را به عنوان پارامتر دریافت می کند که یعنی **Add** برای پارامتر نیاز به یک ارجاع (یا اشاره گر) به یک شی در هیپ مدیریت شده دارد. اما در کد قبلی، من **p** را ارسال کردم، که یک **Point** بوده و نوع مقداری است. برای آنکه کد کار کند، نوع مقداری **Point** باید به یک شی که در هیپ مدیریت می شود تبدیل گشته و ارجاعی به این شی به دست بیاید.

می توان یک نوع مقداری را به یک نوع ارجاعی با مکانیزی به نام بسته بندی **boxing** تبدیل کرد، آنچه هنگام بسته بندی شدن یک نوع مقداری در داخل رخ می دهد، به این شرح است:

۱. حافظه از هیپ مدیریت شده تخصیص می یابد. مقدار حافظه تخصیص یافته برابر با اندازه ی مورد نیاز فیلدهای نوع مقداری به اضافه ی دو عنصر اضافی (اشاره گر شی نوع و اندیس بلوک همزمانی) که تمام شی ها در هیپ به این دو نیاز دارند، است.

۲. فیلدهای نوع مقداری به حافظه‌ی هیپ جدیداً تخصیص یافته کپی می‌شوند.

۳. آدرس شی برگشت داده می‌شود. این آدرس اکنون یک ارجاع به یک شی است و نوع مقداری اکنون یک نوع ارجاعی است.

کامپایلر سی‌شارپ به صورت خودکار کد IL لازم برای بسته‌بندی یک نمونه‌ی نوع مقداری را تولید می‌کند اما شما هنوز باید آنچه در درون رخ می‌دهد را درک کنید تا از مشکلات کارایی و اندازه‌ی کد آگاه باشید.

در کد قبلی، کامپایلر سی‌شارپ تشخیص داد که من یک نوع مقداری به متدی که نیاز به نوع ارجاعی دارد، ارسال کرده‌ام و به صورت خودکار کدی برای بسته‌بندی شی تولید نمود. پس در زمان اجرا، فیلدهای موجود در نمونه‌ی **p** از نوع مقداری **Point** به شی جدیداً تخصیص یافته‌ی **Point** کپی می‌شوند. آدرس شی **Point** بسته‌بندی شده (که اکنون یک نوع ارجاعی است)، برگردانده می‌شود و سپس به متد **Add** ارسال می‌شود. شی **Point** تا زمان جمع-آوری در هیپ باقی می‌ماند. متغیر نوع مقداری **Point**، **(p)** می‌تواند مجدداً استفاده شود چون **ArrayList** هرگز چیزی درباره‌ی آن نمی‌داند. توجه کنید که دوره حیات نوع مقداری بسته‌بندی شده بیش از دوره حیات نوع مقداری بسته‌بندی نشده است.

نکته این نکته باید گفته شوند که اکنون **FCL** دارای مجموعه‌های جدیدی از کلاس‌های مجموعه‌های جنریک (**generic**) است که کلاس‌های مجموعه‌های غیرجنریک را منسوخ کرده اند. برای نمونه شما باید از کلاس **<T>System.Collections.Generic.List** به جای کلاس **System.Collections.ArrayList** استفاده کنید. کلاس‌های مجموعه‌های جنریک بسیار بهتر از معادل‌های غیرجنریک‌شان هستند. برای نمونه، **API** تمیز شده و کارایی آن افزایش یافته است و کارایی کلاس‌های مجموعه‌ها نیز بسیار بهتر شده است. اما یکی از بزرگترین بهبودها این است که کلاس‌های مجموعه‌های جنریک به شما اجازه کار با مجموعه‌ی نوع‌های مقداری بدون آنکه نیاز باشد اقلام مجموعه، بسته بندی/باز شوند، را می‌دهند. این به تنهایی کارایی را بسیار افزایش می‌دهد چون اشیاء کمتری در هیپ ساخته شده و در نتیجه تعداد جمع‌آوری زباله در برنامه شما کاهش می‌یابد. علاوه بر این شما از امنیت نوع در زمان کامپایل بهره خواهید برد و کد شما به خاطر تبدیل کمتر تمیزتر می‌شود. تمام این‌ها در فصل ۱۲ "جنریک‌ها" گفته می‌شود.

حال که می‌دانید بسته‌بندی چگونه انجام می‌شود، اجازه دهید درباره‌ی باز کردن بحث کنیم. فرض کنید می‌خواهید اولین عنصر از **ArrayList** را با کد زیر بدست آورید:

```
Point p = (Point) a[0];
```

در اینجا، شما ارجاع (یا اشاره‌گر) موجود در عنصر **0** از **ArrayList** را برداشتید و سعی کردید آن را در یک نمونه‌ی نوع مقداری **Point**، **p**، قرار دهید. برای آنکه این کد کار کند، تمام فیلدهای موجود در شی **Point** بسته‌بندی شده، باید به درون متغیر نوع مقداری، **p**، که در پشتی ترد است، کپی شوند. CLR این کار را در دو مرحله انجام می‌دهد: اول، آدرس فیلدهای **Point** در شی **Point** بسته‌بندی شده بدست می‌آید. این فرآیند، باز کردن **unboxing** نامیده می‌شود. سپس مقادیر این فیلدها از هیپ به نمونه‌ی نوع مقداری که در پشتی است، کپی می‌شود.

باز کردن دقیقاً مخالف بسته‌بندی کردن نیست. عمل باز کردن از عمل بسته‌بندی کردن بسیار کم هزینه تر است. باز کردن، فقط بدست آوردن یک اشاره‌گر به مقدار خام نوع (فیلدهای داده) درون یک شی است. اشاره‌گر به بخش بسته‌بندی نشده در نمونه‌ی بسته‌بندی شده اشاره دارد. پس، برخلاف بسته‌بندی، باز کردن نیاز به کپی هیچ بایتی در حافظه ندارد. با بیانی چنین صریح، بیان این نکته هم مهم است که یک عمل باز کردن معمولاً با کپی فیلدها همراه است.

واضح است که عملیات بسته‌بندی و باز کردن/کپی کردن کارایی برنامه شما را از هر دو لحاظ سرعت و حافظه کاهش می‌دهد، پس شما باید آگاه باشید چه زمانی کامپایلر به صورت خودکار کدی برای انجام این عملیات‌های تولید می‌کند و سعی نکنید کدی بنویسید که این تولید کد را کاهش دهد.

در درون، هنگامی که یک نمونه نوع مقداری بسته‌بندی شده، باز می‌شود، این اتفاقات می‌افتد:

۱. اگر متغیر حاوی اشاره‌گر به نمونه نوع مقداری بسته‌بندی شده، **null** است، یک **NullReferenceException** تولید می‌شود.

۲. اگر ارجاع (اشاره‌گر) به یک شی که یک نمونه‌ی بسته‌بندی شده از نوع مورد نظر است، اشاره ندارد، یک **InvalidReferenceException** تولید می‌شود.^{۳۳}

مورد دوم یعنی کد زیر آنچنان که انتظار دارید، کار نمی‌کند:

```
public static void Main() {
    Int32 x = 5;
```

^{۳۳} CLR اجازه می‌دهد که شما یک نوع مقداری را به نسخه تهی پذیر از همان نوع باز کنید. این بحث در فصل ۱۹ آمده است.

```

Object o = x;           // Box x; o refers to the boxed object
Int16 y = (Int16) o;    // Throws an InvalidCastException
}

```

منطقاً درست است که **Int32** بسته‌بندی شده که **o** به آن اشاره دارد را به یک **Int16** تبدیل کنیم اما هنگام باز کردن یک شی، تبدیل **o** باید به نوع مقداری بسته‌بندی نشده – در اینجا **Int32** – باشد. روش صحیح نوشتن کد این است:

```

public static void Main() {
    Int32 x = 5;
    Object o = x;           // Box x; o refers to the boxed object
    Int16 y = (Int16)(Int32) o; // Unbox to the correct type and cast
}

```

پیش از این اشاره کردم که یک عمل باز کردن اغلب بلافاصله با یک کپی فیلد همراه است. بگذارید به کد سی‌شارپ زیر که عملیات بسته‌بندی و باز کردن را با هم انجام می‌دهد، نگاه کنیم:

```

public static void Main() {
    Point p;
    p.x = p.y = 1;
    Object o = p; // Boxes p; o refers to the boxed instance
    p = (Point) o; // Unboxes o AND copies fields from boxed
                  // instance to stack variable
}

```

در خط آخر، کامپایلر سی‌شارپ یک کد **IL** برای باز کردن **o** (بدست آوردن آدرس فیلدها در نمونه‌ی بسته‌بندی شده) و کد **IL** دیگری برای کپی فیلدها از هیپ به متغیر **p** در پشته، تولید می‌کند.

حال به این کد نگاه کنید:

```

public static void Main() {
    Point p;
    p.x = p.y = 1;
    Object o = p; // Boxes p; o refers to the boxed instance

    // Change Point's x field to 2
    p = (Point) o; // Unboxes o AND copies fields from boxed

    // instance to stack variable
    p.x = 2; // Changes the state of the stack variable
    o = p; // Boxes p; o refers to a new boxed instance
}

```

کد بخش پایینی فقط برای تغییر فیلد **x** از **Point** از مقدار ۱ به ۲ می‌باشد. برای انجام این کار، یک عملیات باز کردن باید انجام شود، که در پی آن فیلدها کپی می‌شوند و به دنبال آن فیلد (در پشته) تغییر می‌کند و در پایان یک عملیات بسته‌بندی (که یک نمونه‌ی بسته‌بندی شده کاملاً جدید را در هیپ مدیریت-شده می‌سازد) انجام می‌شود. امیدوارم اثر عملیات‌های بسته‌بندی و باز کردن/کپی کردن را بر کارایی برنامه خود ببینید.

برخی زبان‌ها، مثل **C++/CLI** به شما اجازه‌ی باز کردن یک نوع مقداری بسته‌بندی شده را بدون کپی فیلدها می‌دهند. باز کردن، آدرس بخش بسته‌بندی نشده از یک شی بسته‌بندی شده را می‌دهد (با نادیده گرفتن اشاره‌گر شی نوع و اندیس بلوک همزمانی از شی). شما اکنون می‌توانید از این اشاره‌گر برای دستکاری فیلدهای نمونه‌ی باز شده (که درون یک شی بسته‌بندی شده در هیپ است) استفاده کنید. برای مثال، اگر کد قبلی در **C++/CLI** نوشته می‌شد بسیار کارتر می‌بود چون شما می‌توانستید مقدار فیلد **x** از **Point** درون نمونه‌ی **Point** که قبلاً بسته‌بندی شده است، را تغییر دهید. این کار از تخصیص یک شی جدید در هیپ و دو بار کپی کردن تمام فیلدها جلوگیری می‌کرد.

مهم اگر شما در مورد کارایی برنامه‌ی خود بسیار نگرانید، باید بدانید که کامپایلر چه هنگام کدی تولید می‌کند که این عملیات‌ها را انجام می‌دهد. متأسفانه، بسیاری از کامپایلرها به صورت ضمنی کدی تولید می‌کنند که اشیاء را بسته بندی کند، پس وقتی کدی می‌نویسید واضح نیست که بسته بندی دارد رخ می‌دهد. اگر من درباره کارایی یک الگوریتم خاص نگران باشم، همیشه از ابزاری مثل `ILDasm.exe` برای دیدن کد IL استفاده می‌کنم تا ببینم دستورات IL بسته‌بندی (**box**) در کجا هستند.

بگذارید به مثال‌های بیشتری پیرامون بسته‌بندی و باز کردن بپردازیم:

```
public static void Main() {
    Int32 v = 5;           // Create an unboxed value type variable.
    Object o = v;         // o refers to a boxed Int32 containing 5.
    v = 123;              // Changes the unboxed value to 123
    Console.WriteLine(v + ", " + (Int32) o); // Displays "123, 5"
}
```

در این کد، آیا می‌توانید حدس بزنید چند عملیات بسته‌بندی رخ می‌دهد؟ شاید شما شگفت زده شوید اگر بدانید جواب سه است. بگذارید با دقت کد را بررسی کنیم تا درک کنید واقعا چه اتفاقی می‌افتد. برای آنکه بهتر درک کنید، کد IL تولیدی برای متد **Main** در کد قبلی، را آورده‌ام. من کد را با توضیحات گذاشته‌ام تا با راحتی تک تک عملیات‌ها را ببینید:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      45 (0x2d)
    .maxstack 3
    .locals init (int32 v_0,
                  object v_1)
    // Load 5 into v.
    IL_0000: ldc.i4.5
    IL_0001: stloc.0

    // Box v and store the reference pointer in o.
    IL_0002: ldloc.0
    IL_0003: box          [mscorlib]System.Int32
    IL_0008: stloc.1

    // Load 123 into v.
    IL_0009: ldc.i4.s     123
    IL_000b: stloc.0

    // Box v and leave the pointer on the stack for Concat.
    IL_000c: ldloc.0
    IL_000d: box          [mscorlib]System.Int32

    // Load the string on the stack for Concat.
    IL_0012: ldstr        ", "

    // Unbox o: Get the pointer to the In32's field on the stack.
    IL_0017: ldloc.1
    IL_0018: unbox.any    [mscorlib]System.Int32
```



```

// Box the Int32 and leave the pointer on the stack for Concat.
IL_001d: box          [mscorlib]System.Int32

// Call Concat.
IL_0022: call         string [mscorlib]System.String::Concat(object,
                                object,
                                object)

// The string returned from Concat is passed to WriteLine.
IL_0027: call         void [mscorlib]System.Console::WriteLine(string)

// Return from Main terminating this application.
IL_002c: ret
} // end of method App::Main

```

ابتدا، یک نمونه‌ی بسته‌بندی نشده نوع مقداری **Int32 (v)** در پشته ساخته و با **5** مقداردهی اولیه می‌شود. سپس یک متغیر **(o)** از نوع **Object** ساخته شده و با اشاره به **v** مقداردهی اولیه می‌گردد. اما چون متغیرهای نوع ارجاعی همیشه باید به اشیاء درون هیپ اشاره کنند، سی‌شارپ کد **IL** مناسب برای بسته‌بندی و ذخیره کردن آدرس کپی بسته‌بندی شده از **v**، به درون **o** را تولید می‌کند. اکنون مقدار **۱۲۳** در نمونه‌ی بسته‌بندی نشده‌ی نوع مقداری **v** قرار می‌گیرد؛ این کار اثری بر مقدار بسته‌بندی شده **Int32** ندارد و مقدار خودش یعنی **5** را نگهداری می‌کند.

بعد از آن، فراخوانی متد **WriteLine** است. **WriteLine**، انتظار یک شی **String** را به عنوان پارامتر دارد، اما هیچ شی رشته‌ای وجود ندارد. به جای آن، این سه مورد وجود دارند: یک نمونه‌ی بسته‌بندی نشده‌ی نوع مقداری **Int32 (v)**، یک رشته (که یک نوع ارجاعی است) و یک ارجاع به نمونه‌ی بسته‌بندی شده نوع مقداری **Int32 (o)** که در حال تبدیل به یک **Int32** باز شده می‌باشد. این‌ها باید به طریقی ترکیب شده و یک **String** بسازند.

برای ساخت یک **String**، کامپایلر سی‌شارپ کدی تولید می‌کند که متد استاتیک **Concat** از **String** را فراخوانی کند. چندین نسخه‌ی سربارگذاری شده از متد **Concat** وجود دارد که همه یکسان کار می‌کنند و تنها تفاوت در تعداد پارامترهاست. چون یک رشته از اتصال سه چیز می‌خواهد درست شود، کامپایلر نسخه‌ی زیر از متد **Concat** را انتخاب می‌کند:

```
public static String Concat (Object arg0, Object arg1, Object arg2);
```

برای اولین پارامتر، **arg0**، متغیر **v** ارسال می‌شود. اما **v** یک پارامتر مقداری بسته‌بندی نشده است و **arg0** یک **Object** است، پس **v** باید بسته‌بندی شود و آدرس **v** ی بسته‌بندی شده برای **arg0** ارسال شود. برای پارامتر **arg1**، رشته‌ی **","** به عنوان یک ارجاع به یک شی **String** ارسال می‌شود. سرانجام برای پارامتر **arg2**، **o** (یک ارجاع به یک **Object**) به یک **Int32** تبدیل می‌شود. این کار نیاز به یک عملیات باز کردن (بدون عمل کپی) دارد که آدرس **Int32** بسته‌بندی نشده درون **Int32** بسته‌بندی شده را به دست می‌آورد. این نمونه‌ی **Int32** بسته‌بندی نشده باید مجدداً بسته‌بندی شود و آدرس حافظه‌ی نمونه‌ی بسته‌بندی جدید برای پارامتر **arg2** به **Concat** ارسال شود.

متد **Concat** متد **Tostring** مربوط به هر شی را فراخوانی کرده و نمایش رشته‌ای هر رشته را به هم متصل می‌کند. شی **String** برگشتی از **Concat** به متد **WriteLine** برای نمایش نتیجه نهایی ارسال می‌شود.

باید اشاره کنم که کد **IL** تولیدی در صورتی که متد **WriteLine** به صورت زیر نوشته شود، بسیار کارا تر خواهد بود.

```
Console.WriteLine(v + ", " + o); // Displays "123, 5"
```

این خط معادل نسخه‌ی قبلی است به جز آنکه من تبدیل (**Int32**) قبل از متغیر **o** را حذف کردم. این کد کارا تر است چون **o** از قبل یک نوع ارجاعی به یک **Object** است و به سادگی می‌توان آدرس آن را به متد **Concat** ارسال نمود. پس حذف تبدیل، دو عمل را برای ما صرفه جویی کرد، یک باز کردن و یک بسته‌بندی. براحتی می‌توان این صرفه جویی را با ساخت مجدد برنامه و بررسی کد **IL** تولیدی، بررسی کرد:

```

.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      35 (0x23)
    .maxstack 3

```

```

.locals init (int32 v_0,
              object v_1)
// Load 5 into v.
IL_0000: ldc.i4.5
IL_0001: stloc.0

// Box v and store the reference pointer in o.
IL_0002: ldloc.0
IL_0003: box          [mscorlib]System.Int32
IL_0008: stloc.1

// Load 123 into v.
IL_0009: ldc.i4.s     123
IL_000b: stloc.0

// Box v and leave the pointer on the stack for Concat.
IL_000c: ldloc.0
IL_000d: box          [mscorlib]System.Int32

// Load the string on the stack for Concat.
IL_0012: ldstr       ", "

// Load the address of the boxed Int32 on the stack for Concat.
IL_0017: ldloc.1

// Call Concat.
IL_0018: call        string [mscorlib]System.String::Concat(object,
                                                            object,
                                                            object)

// The string returned from Concat is passed to WriteLine.
IL_001d: call        void [mscorlib]System.Console::WriteLine(string)

// Return from Main terminating this application.
IL_0022: ret
} // end of method App::Main

```

یک مقایسه سریع کد IL این دو نسخه از متد **Main** نشان می‌دهد که نسخه‌ی بدون تبدیل (**Int32**)، ۱۰ بایت کوچکتر از نسخه‌ی با تبدیل است. مراحل اضافی بسته‌بندی/بازکردن در نسخه‌ی اول به وضوح کد بیشتری تولید می‌کند. نگرانی بزرگتر این است که مرحله‌ی بسته‌بندی اضافی، اشیاء بیشتری در هیپ مدیریت شده ایجاد می‌نماید که باید در آینده جمع آوری شوند. مطمئناً هر دو نسخه جواب یکسانی می‌دهند و تفاوت در سرعت قابل توجه نیست، اما عملیات‌های غیر ضروری بسته‌بندی اگر در یک حلقه باشند، به شدت باعث کاهش سرعت و افزایش مصرف حافظه‌ی برنامه شما می‌شوند. شما می‌توانید عملکرد کد قبلی را با فراخوانی **WriteLine** به صورت زیر، حتی بیشتر بهبود بخشید.

```
Console.WriteLine(v.ToString() + ", " + o); // Displays "123, 5"
```

اکنون **ToString** بر روی نمونه‌ی بسته‌بندی نشده‌ی نوع مقداری **v** فراخوانی می‌شود و یک **String** برمی‌گردد. اشیاء رشته‌ای نوع‌های ارجاعی هستند و به راحتی به متد **Concat** بدون هیچ گونه بسته‌بندی ارسال می‌شوند.

بگذارید مثال دیگری در بحث بسته‌بندی و باز کردن را بررسی کنیم.

```
public static void Main() {
```

```

    Int32 v = 5;           // Create an unboxed value type variable.
    Object o = v;        // o refers to the boxed version of v.

    v = 123;             // Changes the unboxed value type to 123
    Console.WriteLine(v); // Displays "123"

    v = (Int32) o;       // Unboxes and copies o into v
    Console.WriteLine(v); // Displays "5"
}

```

چند عملیات بسته‌بندی در این کد می‌بینید؟ جواب یک است. دلیل این است که کلاس **System.Console** یک متد **WriteLine** تعریف می‌کند که یک **Int32** به عنوان پارامتر دریافت می‌نماید:

```
public static void WriteLine(Int32 value);
```

در دو فراخوانی بالا به **WriteLine**، متغیر **v** یک نمونه‌ی بسته‌بندی نشده‌ی نوع مقداری **Int32**، با مقدار ارسال می‌شود. حال ممکن است که متد **WriteLine** در درون خود این **Int32** را بسته‌بندی کند، اما شما کنترلی بر آن ندارید. نکته مهم این است که شما تلاش خود برای حذف بسته‌بندی از کد خود را انجام داده اید.

اگر به FCL نگاه کنید، متوجه تعداد زیادی متدهای سربارگذاری شده می‌شوید که فقط بر اساس نوع پارامتر متفاوتند. برای نمونه، نوع **System.Console** چندین نسخه‌ی سربارگذاری شده از متد **WriteLine** را ارائه می‌کند.

```

public static void WriteLine(Boolean);
public static void WriteLine(Char);
public static void WriteLine(Char[]);
public static void WriteLine(Int32);
public static void WriteLine(UInt32);
public static void WriteLine(Int64);
public static void WriteLine(UInt64);
public static void WriteLine(Single);
public static void WriteLine(Double);
public static void WriteLine(Decimal);
public static void WriteLine(Object);
public static void WriteLine(String);

```

همچنین مجموعه‌های مشابهی از متدهای سربارگذاری شده برای متد **Write** از **System.Console**، متد **Write** از **System.IO.BinaryWriter**، متدهای **Write** و **WriteLine** از **System.IO.TextWriter**، متد **AddValue** از **System.Runtime.Serialization.SerializationInfo**، متد **Append** و **Insert** از **System.Text.StringBuilder** و غیره را پیدا می‌کنید. اکثر این متدها، نسخه‌های سربارگذاری شده را فقط برای کاهش تعداد عملیات بسته‌بندی برای نوع‌های مقداری رایج ارائه می‌کنند.

اگر شما نوع مقداری خودتان را تعریف می‌کنید، کلاس‌های FCL نسخه‌ی سربارگذاری شده از این متدها که نوع مقداری شما را دریافت کنند، ندارد. گذشته از این، تعداد زیادی از نوع‌های مقداری تعریف شده در FCL وجود دارند که سربارگذاری شده‌ی این متدها برای آن نوع‌ها وجود ندارد. اگر متدی را فراخوانی کنید که یک سربارگذاری برای نوع مقداری ارسال شده نداشته باشد، همیشه نسخه‌ی سربارگذاری شده‌ای از متد که یک **Object** دریافت می‌کند را فراخوانی کرده‌اید. ارسال یک نمونه‌ی نوع مقداری به عنوان یک **Object**، منجر به بسته‌بندی می‌شود که به شدت کارایی را کاهش خواهد داد. اگر شما کلاس خودتان را تعریف می‌کنید، می‌توانید متدهای کلاس را جنریک (generic) تعریف کنید (که احتمالاً نوع پارامترها را به نوع مقداری محدود می‌کنید). جنریک‌ها به شما راهی برای تعریف یک متد که بدون بسته‌بندی هر نوعی را دریافت کند، ارائه می‌کنند. جنریک‌ها در فصل ۱۲ بحث می‌شوند.

یک نکته آخر پیرامون بسته‌بندی: اگر می‌دانید کدی که می‌نویسید مکرراً کامپایلر را مجبور به بسته‌بندی یک نوع مقداری می‌کند، اگر خودتان به صورت دستی، نوع مقداری را بسته‌بندی کنید، کد کوچکتر و سریعتر می‌شود. یک نمونه ببینید:

```
using System;
```

```
public sealed class Program {
```

```

public static void Main() {
    Int32 v = 5; // Create an unboxed value type variable.

    #if INEFFICIENT
        // when compiling the following line, v is boxed
        // three times, wasting time and memory.
        Console.WriteLine("{0}, {1}, {2}", v, v, v);
    #else
        // The lines below have the same result, execute
        // much faster, and use less memory.
        Object o = v; // Manually box v (just once).
        // No boxing occurs to compile the following line.
        Console.WriteLine("{0}, {1}, {2}", o, o, o);
    #endif
}
}

```

اگر نماد **INEFFICIENT** تعریف شده باشد و این کد با این نماد کامپایل شود، کامپایلر کدی تولید می‌کند که **v** را سه بار بسته‌بندی می‌کند و باعث تخصیص سه شی در هیپ می‌شود. این کار بسیار بی‌فایده است چون هر شی دقیقاً مقدار یکسان **5** را دارد. اگر کد بدون آنکه نماد **INEFFICIENT** تعریف شده باشد کامپایل شود، **v** فقط یکبار بسته‌بندی می‌شود، پس فقط یک شی از هیپ تخصیص داده می‌شود. پس، در فراخوانی **Console.WriteLine**، ارجاع به یک شی بسته‌بندی شده، سه بار ارسال می‌شود. این نسخه دوم بسیار سریعتر اجرا شده و حافظه کمتری مصرف می‌کند.

در این مثال‌ها، واقعا شناسایی این که چه هنگام یک نمونه از یک نوع مقداری نیاز به بسته‌بندی دارد، آسان است. اساساً، اگر شما نیاز به یک ارجاع به یک نمونه از یک نوع مقداری دارید، نمونه باید بسته‌بندی شود. معمولاً این اتفاق رخ می‌دهد، چون شما یک نمونه‌ی نوع مقداری دارید که می‌خواهید آن را به مدتی ارسال کنید که انتظار یک نوع ارجاعی را دارد. هر چند، این مورد تنها موردی نیست که شما نیاز به بسته‌بندی نمونه‌ای از یک نوع مقداری پیدا می‌کنید.

به خاطر بسپارید که به دو دلیل نوع‌های مقداری بسته‌بندی نشده، نوع‌های سبک وزن تری از نوع‌های ارجاعی هستند:

- آن‌ها در هیپ مدیریت شده تخصیص داده نمی‌شوند.

- اعضای اضافی که هر شی درون هیپ دارد را ندارند: یک اشاره‌گر شی نوع و اندیس بلوک همزمانی.

چون نوع‌های مقداری بسته‌بندی نشده، یک اندیس بلوک همزمانی ندارند، نمی‌توانید چندین ترد داشته باشید که دسترسی شان به نمونه را با استفاده از متدهای نوع **System.Threading.Monitor** (یا با عبارت **lock** سی‌شارپ) همزمان کنند.

با وجود اینکه نوع‌های مقداری بسته‌بندی نشده یک اشاره‌گر شی نوع ندارند، باز هم می‌توانید متدهای مجازی (مثل **GetHashCode**، **Equals** یا **ToString**) ارث برده شده یا بازنویسی شده توسط نوع را فراخوانی کنید. اگر نوع مقداری شما یکی از این متدهای مجازی را بازنویسی کند، CLR می‌تواند متد را به صورت غیرمجازی به کار برد چون نوع‌های مقداری به صورت ضمنی مهر شده‌اند و هیچ نوعی نمی‌تواند از آن‌ها مشتق شود. به علاوه، نمونه‌ی نوع مقداری که می‌خواهد متد مجازی را صدا بزند بسته‌بندی شده نیست. هر چند، اگر نسخه بازنویسی شده‌ی شما از متدهای مجازی، پیاده‌سازی نوع پایه از متد را فراخوانی کند، نمونه‌ی نوع مقداری هنگام این فراخوانی بسته‌بندی می‌شود تا یک ارجاع به یک شی در هیپ برای اشاره‌گر **this** در متد پایه ارسال شود.

هر چند فراخوانی یک متد غیرمجازی ارث برده شده (مثل **GetType** یا **MemberwiseClone**) همیشه نیاز دارد که نوع مقداری بسته‌بندی شود چون این متدها توسط **System.Object** تعریف شده‌اند، پس متد انتظار دارد، آرگومان **this** اشاره‌گری باشد که به یک شی در هیپ اشاره می‌کند.

افزون بر این، تبدیل یک نمونه‌ی بسته‌بندی نشده از یک نوع مقداری به یکی از رابط‌های نوع، نیاز به بسته‌بندی نمونه دارد چون متغیرهای رابط همیشه باید ارجاعی به یک شی در هیپ داشته باشند. (در فصل ۱۳ "رابط‌ها" درباره‌ی رابط‌ها صحبت می‌کنم). کد زیر این مطلب را نشان می‌دهد:

```

using System;

internal struct Point : IComparable {
    private readonly Int32 m_x, m_y;
}

```

```

// Constructor to easily initialize the fields
public Point(Int32 x, Int32 y) {
    m_x = x;
    m_y = y;
}

// Override ToString method inherited from System.ValueType
public override String ToString() {
    // Return the point as a string
    return String.Format("{0}, {1}", m_x, m_y);
}

// Implementation of type-safe CompareTo method
public Int32 CompareTo(Point other) {
    // Use the Pythagorean Theorem to calculate
    // which point is farther from the origin (0, 0)
    return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
        - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
}

// Implementation of IComparable's CompareTo method
public Int32 CompareTo(Object o) {
    if (GetType() != o.GetType()) {
        throw new ArgumentException("o is not a Point");
    }
    // Call type-safe CompareTo method
    return CompareTo((Point) o);
}
}

public static class Program {
    public static void Main() {
        // Create two Point instances on the stack.
        Point p1 = new Point(10, 10);
        Point p2 = new Point(20, 20);

        // p1 does NOT get boxed to call ToString (a virtual method).
        Console.WriteLine(p1.ToString()); // "(10, 10)"

        // p DOES get boxed to call GetType (a non-virtual method).
        Console.WriteLine(p1.GetType()); // "Point"

        // p1 does NOT get boxed to call CompareTo.
        // p2 does NOT get boxed because CompareTo(Point) is called.
        Console.WriteLine(p1.CompareTo(p2)); // "-1"
    }
}

```

```

// p1 DOES get boxed, and the reference is placed in c.
IComparable c = p1;
Console.WriteLine(c.GetType());// "Point"

// p1 does NOT get boxed to call CompareTo.
// Since CompareTo is not being passed a Point variable,
// CompareTo(Object) is called which requires a reference to
// a boxed Point.
// c does NOT get boxed because it already refers to a boxed Point.
Console.WriteLine(p1.CompareTo(c));// "0"

// c does NOT get boxed because it already refers to a boxed Point.
// p2 does get boxed because CompareTo(Object) is called.
Console.WriteLine(c.CompareTo(p2));// "-1"

// c is unboxed, and fields are copied into p2.
p2 = (Point) c;

// Proves that the fields got copied into p2.
Console.WriteLine(p2.ToString());// "(10, 10)"
}
}

```

این کد سناریوهای مختلفی درباره‌ی بسته‌بندی و باز کردن را نشان می‌دهد.

- **فراخوانی ToString** در فراخوانی به **ToString**، **p1** نیاز به بسته‌بندی ندارد. در ابتدا فکر می‌کنید که **p1** نیاز به بسته‌بندی دارد چون **ToString** یک متد مجازی ارث برده شده از نوع پایه **System.ValueType** است. در حالت عادی، برای فراخوانی یک متد مجازی، CLR نیاز به تعیین نوع شی برای یافتن جدول متد نوع دارد. چون **p1** یک نوع مقداری بسته‌بندی نشده است، هیچ اشاره‌گر شی نوعی وجود ندارد. اما، کامپایلر فقط در لحظه (JIT) می‌بیند که **Point** متد **ToString** را بازنویسی نموده است و کدی تولید می‌کند که **ToString** را به صورت مستقیم (غیر مجازی) بدون هرگونه بسته‌بندی، فراخوانی کند. کامپایلر می‌داند که چندریختی (polymorphism) وجود ندارد چون **Point** یک نوع مقداری است و هیچ نوعی از آن مشتق نمی‌شود تا پیاده‌سازی دیگری از این متد مجازی ارائه کند. توجه کنید اگر متد **ToString** از **Point** در درون خود، **base.ToString()** را فراخوانی کند، آنگاه نمونه نوع مقداری هنگام فراخوانی **ToString** از **System.ValueType**، بسته‌بندی خواهد شد.
- **فراخوانی GetType** در فراخوانی متد غیر مجازی **GetType**، **p1** باید بسته‌بندی شود. علت آن است که نوع **Point**، **GetType** را از **System.Object** به ارث می‌برد. پس برای فراخوانی **GetType**، CLR باید از اشاره‌گری به یک شی نوع استفاده کند که این با بسته‌بندی **p1** بدست می‌آید.
- **فراخوانی CompareTo (اولین بار)** در اولین فراخوانی به **CompareTo**، **p1** نیاز به بسته‌بندی شدن ندارد چون **Point**، متد **CompareTo** را پیاده‌سازی کرده و کامپایلر می‌تواند آن را مستقیماً صدا بزند. توجه کنید که یک متغیر **Point (p2)** به **CompareTo** ارسال شده است، پس کامپایلر نسخه‌ی سربارگذاری شده از **CompareTo** که یک پارامتر **Point** قبول می‌کند را صدا می‌زند. این یعنی، **p2** با مقدار به **CompareTo** ارسال می‌شود و هیچ بسته‌بندی نیاز نیست.
- **تبدیل به IComparable** هنگام تبدیل **p1** به یک متغیر (c) که از نوع رابط است، **p1** باید بسته‌بندی شود، چون رابطه طبق تعریف نوع‌های ارجاعی هستند. پس **p1** بسته‌بندی می‌شود و اشاره‌گر به این شی بسته‌بندی شده در متغیر **c** ذخیره می‌شود. فراخوانی بعدی به **GetType** اثبات می‌کند که **c** به یک **Point** بسته‌بندی شده در هیپ اشاره دارد.
- **فراخوانی CompareTo (بار دوم)** در فراخوانی دوم به **CompareTo**، **p1** نیازی به بسته‌بندی ندارد چون **Point** متد **CompareTo** را پیاده‌سازی کرده و کامپایلر می‌تواند آن را مستقیماً صدا بزند. توجه کنید که متغیر **IComparable (c)** به

CompareTo ارسال شده است و بنابراین کامپایلر نسخه‌ی سربرگذاری شده از **CompareTo** که یک پارامتر **Object** دریافت می‌کند را صدا می‌زند. این یعنی آرگومان ارسالی باید اشاره‌گر به شی‌ای در هیپ باشد. خوشبختانه، **c** به **Point** بسته‌بندی شده اشاره دارد، آدرس حافظه در **c** به **CompareTo** ارسال می‌شود و هیپ بسته‌بندی اضافی نیاز نیست.

- **فراخوانی CompareTo (بار سوم)** در سومین فراخوانی به **CompareTo.c** قبلاً به یک شی بسته‌بندی شده **Point** در هیپ اشاره دارد. چون **c** از نوع رابط **IComparable** است، شما فقط می‌توانید نسخه‌ای از متد **CompareTo** از رابط را صدا بزنید که نیاز به یک پارامتر **Object** داشته باشد. این یعنی، آرگومان ارسالی باید اشاره‌گری به یک شی در هیپ باشد. پس **p2** بسته‌بندی شده و اشاره‌گری به این شی بسته‌بندی شده به **CompareTo** ارسال می‌شود.
- **تبدیل به Point** هنگام تبدیل **c** به یک **Point**، شی در هیپ که توسط **c** به آن اشاره می‌شود، باز شده و فیلدهایش از هیپ به **p2** کپی می‌شوند و یک نمونه از نوع **Point** در پشته جای می‌گیرد.

من درک می‌کنم که همه‌ی این اطلاعات پیرامون نوع‌های ارجاعی نوع‌های مقداری و بسته‌بندی در نگاه اول سنگین و گیج کننده است. اما درک قوی این مفاهیم برای موفقیت دراز مدت هر برنامه‌نویس دات‌نت فریمورک حیاتی است. به من اطمینان کنید: فهم دقیق این مفاهیم به شما اجازه می‌دهند که برنامه‌هایی کارا، با سرعت و به آسانی بسازید.

تغییر فیلدها در یک نوع مقداری بسته بندی شده به کمک رابط‌ها (و چرا شما نباید این کار را انجام دهید)

برای کمی سرگرمی و آزمایش آنچه پیرامون نوع‌های مقداری، بسته‌بندی و باز کردن درک کرده اید، کد زیر را بررسی می‌کنیم تا ببینیم آیا می‌توانید خروجی را تعیین کنید:

```
using System;

// Point is a value type.
internal struct Point {
    private Int32 m_x, m_y;
    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    public void Change(Int32 x, Int32 y) {
        m_x = x; m_y = y;
    }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}

public sealed class Program {
    public static void Main() {
        Point p = new Point(1, 1);
        Console.WriteLine(p);
        p.Change(2, 2);
        Console.WriteLine(p);
        Object o = p;
```

```

        Console.WriteLine(o);
        ((Point) o).Change(3, 3);
        Console.WriteLine(o);
    }
}

```

به سادگی، **Main** یک نمونه (**p**) از نوع مقداری **Point** در پشته می‌سازد و فیلدهای **m_x** و **m_y** آن را با مقدار **1** تنظیم می‌کند. سپس **p** قبل از اولین فراخوانی به **WriteLine** بسته‌بندی می‌شود تا **Tostring** بر روی **Point** بسته‌بندی شده فراخوانی شود و طبق انتظار **(1, 1)** نمایش داده شود. سپس، **p** برای فراخوانی متد **Change** استفاده می‌شود که مقادیر فیلدهای **m_x** و **m_y** از **p** را در پشته به **2** تغییر می‌دهد. فراخوانی دوم به **WriteLine** نیاز به بسته‌بندی مجدد **p** دارد و طبق انتظار **(2, 2)** را نشان می‌دهد.

اکنون، **p** برای بار سوم بسته‌بندی می‌شود و **o** به شی بسته‌بندی شده **Point** اشاره می‌کند، سومین فراخوانی به **WriteLine** مجدداً طبق انتظار **(2, 2)** را نشان می‌دهد. سرانجام، من می‌خواهم متد **Change** را برای آپدیت فیلدهای شی بسته‌بندی شده **Point** فراخوانی کنم. اما، **Object** (نوع متغیر **o**) چیزی درباره‌ی متد **Change** نمی‌داند پس باید ابتدا **o** را به یک **Point** تبدیل کنم.

تبدیل **o** به یک **Point**، **o** را باز کرده و فیلدهای **Point** بسته‌بندی شده را در یک **Point** موقتی در پشته‌ی ترد کپی می‌کند! فیلدهای **m_x** و **m_y** از این **Point** موقتی به **3** و **3** تغییر پیدا می‌کنند اما **Point** بسته‌بندی شده با این فراخوانی به **Change**، تغییر نمی‌کند. هنگامیکه **WriteLine** برای بار چهارم فراخوانی می‌شود، مجدداً **(2, 2)** نمایش داده می‌شود. بسیاری از برنامه‌نویسان انتظار چنین چیزی را ندارند.

بعضی زبان‌ها مثل **C++/CLI** به شما اجازه‌ی تغییر فیلدهای یک نوع مقداری بسته‌بندی شده را می‌دهند، اما سی‌شارپ این کار را نمی‌کند. هر چند، شما می‌توانید با استفاده از یک رابط، سی‌شارپ را گول بزنید. کد زیر نسخه‌ی تغییر یافته کد قبلی است:

```

using System;

// Interface defining a Change method
internal interface IChangeBoxedPoint {
    void Change(Int32 x, Int32 y);
}

// Point is a value type.
internal struct Point : IChangeBoxedPoint {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    public void Change(Int32 x, Int32 y) {
        m_x = x; m_y = y;
    }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}

public sealed class Program {
    public static void Main() {

```



```

Point p = new Point(1, 1);

Console.WriteLine(p);

p.Change(2, 2);
Console.WriteLine(p);

Object o = p;
Console.WriteLine(o);

((Point) o).Change(3, 3);
Console.WriteLine(o);

// Boxes p, changes the boxed object and discards it
((IChangeBoxedPoint) p).Change(4, 4);
Console.WriteLine(p);

// Changes the boxed object and shows it
((IChangeBoxedPoint) o).Change(5, 5);
Console.WriteLine(o);
}
}

```

این کد تقریباً معادل کد قبلی است. تفاوت اصلی اینست که متد **Change** توسط رابط **IChangedBoxedPoint** تعریف شده و اکنون نوع **Point**، رابط را پیاده‌سازی کرده است. درون **Main**، چهار فراخوانی اول به **WriteLine** یکسان بوده و همان نتایج قبلی (طبق انتظار) را تولید می‌کنند. هر چند، من دو مثال دیگر به انتهای **Main** افزوده‌ام.

در اولین مثال، **Point** بسته‌بندی نشده، **p**، به یک **IChangeBoxedPoint** تبدیل می‌شود. این تبدیل باعث می‌شود مقدار درون **p** بسته‌بندی شود. **Change** بر روی مقدار بسته‌بندی شده فراخوانی می‌شود که مقدار فیلدهای **m_x** و **m_y** آنرا به **4** و **4** تغییر می‌دهد، اما پس از آنکه **Change** برمی‌گردد، شی بسته‌بندی شده بلافاصله آماده جمع‌آوری به عنوان زباله است. پس پنجمین فراخوانی به **WriteLine**، **(2, 2)** را نمایش می‌دهد. بسیاری از برنامه‌نویسان انتظار چنین جوابی را ندارند.

در آخرین مثال، **Point** بسته‌بندی شده که توسط **o** ارجاع می‌شود به یک **IChangeBoxedPoint** تبدیل می‌شود. هیچ بسته‌بندی در اینجا نیاز نیست چون **o** قبلاً یک **Point** بسته‌بندی شده است. پس **Change** فراخوانی می‌شود که فیلدهای **m_x** و **m_y** از **Point** بسته‌بندی شده را تغییر می‌دهد. متد **Change** از رابط به من اجازه داد که فیلدهای یک شی **Point** بسته‌بندی شده را تغییر دهم! اکنون وقتی **WriteLine** فراخوانی می‌شود، **(5, 5)** را طبق انتظار نشان می‌دهد. هدف از کل این مثال بیان این بود که چگونه یک متد رابط توانایی تغییر فیلدهای یک نوع مقداری بسته‌بندی شده را دارد. در سی‌شارپ این کار بدون یک رابط ممکن نیست.

مهم قبلا در این فصل، بیان کردم که نوع‌های مقداری باید تغییر ناپذیر باشند: یعنی اینکه، آن‌ها نباید هیچ عضوی تعریف کنند که هر یک از فیلدهای نمونه‌ی نوع را تغییر دهد. در واقع، من توصیه کردم که نوع‌های مقداری، فیلدهایشان را با **readonly** علامت بزنند تا کامپایلر هنگامی که تصادفاً کدی می‌نویسد که سعی در تغییر یک فیلد را دارد، اعلام خطا کند. مثال قبلی، به وضوح نشان داد چرا نوع‌های مقداری باید تغییر ناپذیر باشند. رفتار غیر منتظره که در مثال قبل دیدیم تماماً هنگامی رخ می‌دهد که سعی در فراخوانی متدی دارید که فیلدهای نمونه‌ی نوع مقداری را تغییر می‌دهد. اگر پس از ساخت یک نوع مقداری هیچ متدی که وضعیت آن را تغییر دهد، فراخوانی نکنید، وقتی بسته‌بندی و باز کردن/کپی فیلدها رخ می‌دهد، دچار سردرگمی نمی‌شوید. اگر نوع مقداری تغییر ناپذیر باشد، شما فقط می‌توانید وضعیت آنرا از یکجا به جای دیگر کپی کنید و از هیچ یک از رفتارهای مشاهده شده تعجب نخواهید کرد. تعدادی از برنامه‌نویسان فصل‌های این کتاب را مرور کرده‌اند. پس از خواندن بعضی از نمونه کدهای من (همانند مثال قبلی)، این افراد به من گفتند که نوع‌های مقداری را دیگر استفاده نمی‌کنند. من باید بگویم که این نکات ظریف از نوع‌های مقداری چندین روز وقت مرا برای خطایابی معطل کرده‌اند که دلیل آنکه آن‌ها را در کتاب آورده‌ام همین است. من امیدوارم شما این نکات را به خاطر بسپارید و هنگام مواجه شدن با آن‌ها، کاملاً آماده و مجهز باشید. مطمئناً، شما نباید از نوع‌های مقداری بترسید. آن‌ها مفید هستند و جایگاه خود را دارند. گذشته از این، یک برنامه به **Int32** کوچک و دوست داشتنی در حال حاضر و در آینده نیاز دارد. فقط به خاطر بسپارید که نوع‌های مقداری و نوع‌های ارجاعی رفتاری کاملاً متفاوت بسته به چگونگی استفاده از آن‌ها، دارند. در واقع شما باید کد قبلی را برداشته و نوع **Point** را به جای **struct**، یک **class** تعریف کنید تا رفتار متفاوت را ببینید. سرانجام شاد می‌شوید اگر بدانید که نوع‌های مقداری اصلی که در FCL عرضه می‌شوند – **Byte**، **Int32**، **UInt32**، **Int64**، **Single**، **Double**، **Decimal**، **BigInteger**، **Complex**، تمام **enum** ها و غیره، همه تغییر ناپذیر هستند، پس هنگام استفاده از آن‌ها رفتاری غیر منتظره نخواهید دید.

برابری و هویت شی

اغلب، برنامه‌نویسان کدی برای مقایسه‌ی اشیاء با یکدیگر می‌نویسند. این کار به ویژه در قرار دادن اشیاء در مجموعه‌ها و مرتب کردن، جستجو یا مقایسه‌ی اقلام یک مجموعه، کاربرد دارد. در این بخش، برابری و هویت شی را بحث می‌کنم و در ضمن می‌گویم چگونه یک نوع را تعریف کنید تا برابری شی را به درستی پیاده‌سازی کند.

نوع **System.Object** یک متد مجازی به نام **Equals** تعریف می‌کند که هدفش برگرداندن **true** در صورتی که دو شی مقدار یکسانی داشته باشند است. پیاده‌سازی متد **Equals** از **Object** شبیه به این است:

```
public class Object {
    public virtual Boolean Equals(Object obj) {

        // If both references point to the same object,
        // they must have the same value.
        if (this == obj) return true;

        // Assume that the objects do not have the same value.
        return false;
    }
}
```

در ابتدا، این به نظر یک پیاده‌سازی پیش فرض معقول برای **Equals** است: اگر آرگومان‌های **this** و **obj** دقیقاً به شی یکسانی اشاره کنند، **true** برمی‌گرداند. این معقولانه به نظر می‌رسد چون **Equals** می‌داند که یک شی باید مقداری برابر با خودش داشته باشد. اما اگر آرگومان‌ها به اشیاء مختلف اشاره کنند، **Equals** نمی‌تواند مطمئن شود که اشیاء مقدار یکسانی دارند و بنابراین **false** برمی‌گرداند. به بیان دیگر، پیاده‌سازی پیش فرض **Equals** از **Object**، برابری هویت (دو ارجاع که به یک شی اشاره دارند) و نه برابری مقداری را پیاده‌سازی می‌کند.

متاسفانه، همانطور که معلوم شد، متد **Equals** از **Object**، پیش فرض معقولانه‌ای نیست و هرگز نمی‌بایست به این طریق پیاده‌سازی می‌شد. شما به سرعت این مشکل را می‌بینید، هنگامیکه پیرامون سلسله مراتب وراثت کلاس و نحوه صحیح بازنویسی **Equals** فکر کنید. نحوه‌ی صحیح پیاده‌سازی متد **Equals** این گونه است:

۱. اگر **obj**، **null** است، **false** برگردد چون وقتی متد غیراستاتیک **Equals** فراخوانی می‌شود، شی جاری که با **this** شناخته می‌شود مطمئناً **null** نیست.
۲. اگر آرگومان‌های **this** و **obj**، به شی یکسانی اشاره دارند، **true** برگردد. این مرحله، می‌تواند هنگام بررسی اشیاء با فیلدهای زیاد، منجر به کاهش کارایی شود.
۳. اگر آرگومان‌های **this** و **obj**، به اشیایی از نوع‌های مختلف اشاره دارند، **false** برگرداند. به وضوح، بررسی اینکه یک شی **String** برابر با یک شی **FileStream** است باید جواب **false** داشته باشد.
۴. برای هر فیلد نمونه تعریف شده توسط نوع، مقدار شی **this** را با مقداری شی **obj** مقایسه کند. اگر یکی از فیلدها برابر نبود، **false** برگرداند.
۵. متد **Equals** از کلاس پایه را برای مقایسه هر فیلد تعریف شده توسط آن، فراخوانی کند. اگر متد **Equals** از کلاس پایه، **false** برگرداند، **false** برگردد و در غیر این صورت **true** برگردد.

پس مایکروسافت می‌بایست **Equals** از **Object** را شبیه به این پیاده‌سازی می‌کرد:

```
public class Object {
    public virtual Boolean Equals(Object obj) {
        // The given object to compare to can't be null
        if (obj == null) return false;

        // If objects are different types, they can't be equal.
        if (this.GetType() != obj.GetType()) return false;

        // If objects are same type, return true if all of their fields match
        // Since System.Object defines no fields, the fields match
        return true;
    }
}
```

اما چون مایکروسافت، **Equals** را اینگونه تعریف نکرده است، قوانین برای پیاده‌سازی **Equals** بسیار پیچیده تر از آن است که فکر می‌کنید. وقتی یک نوع، **Equals** را بازنویسی می‌کند، این بازنویسی باید پیاده‌سازی کلاس پایه‌اش از **Equals** را فراخوانی کند مگر اینکه پیاده‌سازی **Object** از **Equals** را فراخوانی نماید. همچنین این یعنی، چون یک نوع می‌تواند متد **Equals** از **Object** را بازنویسی کند، این متد **Equals**، دیگر برای تشخیص هویت نمی‌تواند فراخوانی شود. برای حل این، **Object** متد استاتیک **ReferenceEquals** را ارائه می‌کند که شبیه به این پیاده‌سازی شده است:

```
public class Object {
    public static Boolean ReferenceEquals(Object objA, Object objB) {
        return (objA == objB);
    }
}
```

همیشه اگر می‌خواهید هویت (آیا دو ارجاع به یک شی اشاره دارند) را بررسی کنید، باید **ReferenceEquals** را فراخوانی کنید. شما نباید از عملگر **==** سی‌شارپ استفاده کنید (مگر آنکه ابتدا هر دو عملوند را به **Object** تبدیل کنید) چون یکی از نوع‌های عملوند می‌تواند عملگر **==** را سربارگذاری کرده و معنی دیگری جز هویت به آن دهد.

همانطور که می‌بینید، دات‌نت فریمورک داستان بسیار گیج‌کننده‌ای پیرامون برابری و هویت شی دارد. ضمناً، **System.ValueType** (کلاس پایه‌ی تمام نوع‌های مقداری)، متد **Equals** از **Object** را بازنویسی کرده و به درستی برای انجام تست برابری اشیاء (و نه تست هویت) آن را پیاده‌سازی کرده است. در درون، **Equals** از **ValueType** به این روش پیاده‌سازی شده است:

۱. اگر آرگومان **obj**، **null** است، **false** برگرداند.

۲. اگر آرگومان‌های **this** و **obj** به اشیایی از نوع‌های متفاوت اشاره دارند، **false** برگرداند.
۳. برای هر فیلد نمونه تعریف شده توسط نوع، مقدار شی **this** را با مقدار شی **obj** با فراخوانی متد **Equals** آن فیلد، مقایسه کند. اگر فیلدی برابر نبود، **false** برگرداند.
۴. **true** برگرداند. متد **Equals** از **Object** توسط متد **Equals** از **ValueType** فراخوانی نمی‌شود.
- در درون، متد **Equals** از **ValueType** برای انجام مرحله شماره ۳ در بالا از رفلکشن (در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن") استفاده می‌کند. چون مکانیزم رفلکشن کند است، هنگام تعریف نوع مقداری خود، شما باید **Equals** را بازنویسی کرده و پیاده‌سازی خود را ارائه کنید تا سرعت مقایسه‌های برابری مقدار که از نمونه‌های نوع شما استفاده می‌کنند، افزایش یابد.
- هنگام تعریف نوع خود، اگر تصمیم گرفتید **Equals** را بازنویسی کنید، باید مطمئن شوید که با چهار خاصیت برابری مطابق است:
- **Equals** باید بازتابی باشد؛ یعنی، **x.Equals(x)** باید **true** برگرداند.
 - **Equals** باید متقارن باشد؛ یعنی، **x.Equals(y)** باید مقدار یکسانی با **y.Equals(x)** برگرداند.
 - **Equals** باید متعددی (تراگذر) باشد؛ یعنی، اگر **x.Equals(y)**، **true** برگرداند و **y.Equals(z)**، **true** برگرداند آنگاه **x.Equals(z)** نیز باید **true** برگرداند.
 - **Equals** باید ثابت و یکنواخت باشد. مشروط به اینکه در دو مقدار مقایسه شده تغییری حاصل نشود، **Equals** باید یکنواخت **true** یا **false** برگرداند.
- اگر پیاده‌سازی شما از **Equals** تمام این قوانین را رعایت نکند، برنامه شما به نحوی عجیب و پیش‌بینی نشده رفتار خواهد کرد.
- هنگام بازنویسی متد **Equals**، چند چیز دیگر وجود دارد که شما احتمالاً انجام می‌دهید:
- نوع، متد **Equals** از رابط **System.IEquatable<T>** را پیاده‌سازی کند این رابط جنریک به شما اجازه‌ی تعریف یک متد **Equals** نوع-امن را می‌دهد. معمولاً، شما متد **Equals** ای را پیاده‌سازی می‌کنید که یک پارامتر **Object** می‌گیرد و در درون خود متد **Equals** نوع-امن را فراخوانی می‌کند.
 - **متدهای عملگر == و != را سربارگذاری کنید** معمولاً، شما این متدهای عملگر را به گونه‌ای پیاده‌سازی می‌کنید که در درون خود، متد **Equals** نوع-امن را فراخوانی کنند.
- علاوه بر این، اگر فکر می‌کنید که نمونه‌های نوع شما برای مرتب شدن مقایسه خواهند شد، شما می‌خواهید نوعتان متد **CompareTo** از **System.IComparable** و متد **CompareTo** نوع-امن از **System.IComparable<T>** را نیز پیاده‌سازی کند. اگر این متدها را پیاده‌سازی کنید، همچنین خواهید خواست که متدهای عملگر مقایسه‌ای مختلف (**>**، **=**، **<**) را بازنویسی کرده و در درون متد نوع-امن **CompareTo** را فراخوانی کنند.

کدهای هش‌شی

طراحان FCL تصمیم گرفتند که اگر هر نمونه از هر شی می‌توانست در یک مجموعه جدول هش قرار بگیرد، به صورت باور نکردنی اتفاق مفیدی می‌بود. برای همین، **System.Object**، متد مجازی **GetHashCode** را ارائه می‌کند تا بتوان برای هر شی یک کد هش **Int32** بدست آورد.

اگر شما نوعی تعریف کرده و متد **Equals** را بازنویسی کنید، باید متد **GetHashCode** را نیز بازنویسی نمایید. در واقع، کامپایلر سی‌شارپ مایکروسافت، اگر شما نوعی تعریف کنید که **Equals** را بدون بازنویسی **GetHashCode**، هشدار زیر را اعلام می‌کند:

“warning CS0659: ‘Program’ overrides Object.Equals(object o) but does not override Object.GetHashCode()”

```
public sealed class Program {
    public override Boolean Equals(Object obj) { ... }
}
```

علت آنکه نوعی که **Equals** را تعریف می‌کند می‌بایست **GetHashCode** را نیز تعریف کند این است که پیاده‌سازی نوع **System.Collections.HashTable**، نوع **System.Collections.Generic.Dictionary** و برخی مجموعه‌های دیگر، نیاز دارند که دو شی دارای مقدار یکسان، دارای کد هش یکسان نیز باشند. پس اگر شما، **Equals** را بازنویسی می‌کنید، شما باید **GetHashCode** را نیز بازنویسی کنید تا مطمئن شوید الگوریتمی که برای محاسبه‌ی برابری استفاده می‌کنید با الگوریتمی که برای محاسبه‌ی کد هش شی استفاده می‌کنید، تطابق داشته باشد.

اساساً، شما وقتی یک جفت کلید/مقدار به یک مجموعه اضافه می‌کنید، ابتدا یک کد هش برای کلید شی بدست می‌آید. این کد هش تعیین می‌کند که جفت کلید/مقدار باید در کدام "جعبه" ذخیره شود. هنگامیکه یک مجموعه نیاز به یافتن یک کلید دارد، کد هش از شی کلید تعیین شده را بدست می‌آورد. این کد جعبه‌ای را نشان می‌دهد که اکنون به صورت سریالی برای یافتن یک شی کلید ذخیره شده که معادل با شی کلید تعیین شده باشد، جستجو می‌شود. استفاده از این الگوریتم برای مرتب کردن و یافتن کلیدها بدین معنی است که اگر یک شی کلید را در یک مجموعه تغییر دهید، مجموعه دیگر نمی‌تواند شی را پیدا کند. اگر قصد تغییر شی کلید در یک جدول هش را دارید، باید جفت کلید/مقدار اصلی را حذف کرده، شی کلید را تغییر دهید و جفت کلید/مقدار جدید را به جدول هش اضافه کنید.

تعریف یک متد **GetHashCode** می‌تواند ساده و آسان باشد. اما بسته به نوع‌های داده‌ای شما و توزیع داده در آن‌ها، یافتن یک الگوریتم هش که بازه‌ای از مقادیر خوب توزیع شده را برگرداند، می‌تواند سخت باشد. نمونه‌ی زیر احتمالاً برای اشیاء **Point** خوب کار می‌کند:

```
internal sealed class Point {
    private readonly Int32 m_x, m_y;
    public override Int32 GetHashCode() {
        return m_x ^ m_y; // m_x XOR'd with m_y
    }
    ...
}
```

هنگام انتخاب یک الگوریتم برای محاسبه‌ی کدهای هش برای نمونه‌های خودتان، راهنمایی‌های زیر را در نظر داشته باشید:

- از الگوریتمی استفاده کنید که توزیع تصادفی خوبی همراه با بهترین سرعت، برای جدول هش به شما دهد.
- الگوریتم شما همچنین می‌تواند متد **GetHashCode** از نوع پایه را صدا زده و از مقدار برگشتی آن استفاده کند. اما، شما معمولاً اقدام به فراخوانی متد **GetHashCode** از **Object** یا **ValueType** نمی‌کنید چون، پیاده‌سازی هر کدام از این متدها منجر به یک الگوریتم هش با کارایی بالا نمی‌شوند.
- الگوریتم شما باید حداقل از یکی از فیلدهای نمونه استفاده کند.
- به صورت ایده‌آل، فیلدهای مورد استفاده در الگوریتم شما باید تغییر ناپذیر باشند. این یعنی، فیلدها هنگام ساخت شی مقداردهی اولیه شوند و هرگز در دوره حیات شی تغییر نکنند.
- الگوریتم شما باید با بالاترین سرعت ممکن اجرا شود.
- اشیاء با مقادیر یکسان باید کد هش یکسانی برگردانند. برای نمونه، دو شی **String** با متن یکسان باید کد هش یکسانی برگردانند.

پیاده‌سازی **System.Object** از **GetHashCode** چیزی درباره‌ی نوع‌های مشتق شده و هرگونه فیلدی در این نوع‌ها نمی‌داند. به همین علت، متد **GetHashCode** از **Object**، عددی برمی‌گرداند که به صورت تضمینی در درون **AppDomain**، انحصاراً شی را شناسایی می‌کند. تضمین می‌شود که این عدد در دوره‌ی حیات شی تغییر نکند. بعد از آنکه شی به عنوان زباله جمع‌آوری شد، به هر حال، عدد منحصر بفرد آن می‌تواند به عنوان کد هش برای یک شی جدید استفاده شود.

نکته اگر نوعی، متد **GetHashCode** از **Object** را بازنویسی کند، شما نمی‌توانید دیگر برای گرفتن یک شناسه‌ی منحصر بفرد از آن استفاده کنید. اگر می‌خواهید یک شناسه منحصر بفرد (درون یک **AppDomain**) برای یک شی بدست آورید، **FCL** متدی برای فراخوانی شما فراهم می‌کند. در فضای نام **System.Runtime.CompilerServices** کلاس **RuntimeHelpers**، متد استاتیک عمومی **GetHashCode** که یک شناسه منحصر بفرد برای یک شی حتی اگر نوع شی، متد **GetHashCode** از **Object** را بازنویسی کرده باشد، برمی‌گرداند را نگاه کنید. این متد نامش را به ارث برده است اما بهتر می‌بود که مایکروسافت نام آن را چیزی شبیه به **GetUniqueObjectID** می‌گذاشت.

پیاده‌سازی **System.ValueType** از **GetHashCode** با رفلکشن (که کند است) و XOR کردن فیلدهای نمونه‌ی نوع با همدیگر، کار می‌کند. این یک پیاده‌سازی ساده است که شاید برای بعضی از نوع‌های مقداری خوب باشد اما من هنوز توصیه می‌کنم که شما **GetHashCode** خود را پیاده‌سازی کنید چون به خوبی می‌دانید که چه کاری انجام می‌دهد و پیاده‌سازی شما سریعتر از پیاده‌سازی **ValueType** خواهد بود.

مهم اگر شما به هر دلیلی مجموعه جدول هاش خود را پیاده‌سازی می‌کنید یا کدی را استفاده می‌کنید که در آن **GetHashCode** فراخوانی می‌شود، شما هرگز و تحت هیچ شرایطی نباید مقادیر کدهای هاش را ذخیره کنید. علت آن است که مقادیر کد هاش در حال تغییر هستند. برای نمونه یک نسخه آتی از یک نوع ممکن است از الگوریتم متفاوتی برای محاسبه‌ی کد هاش شی استفاده کند.

شرکتی بود که به این هشدار مهم توجه نکرد. در وب سایت آن‌ها، کاربران می‌توانستند با انتخاب یک نام کاربری و رمز عبور، اکانت جدید بسازند. وب سایت، رمز عبور **String** را بر می‌داشت، **GetHashCode** را صدا می‌زد و مقدار کد هاش را در پایگاه داده ذخیره می‌نمود. وقتی کاربران مجدداً وارد وب سایت می‌شدند رمز عبور خود را وارد می‌کردند، وب سایت مجدداً **GetHashCode** را صدا می‌زد و مقدار کد هاش را با مقدار ذخیره شده در پایگاه داده مقایسه می‌نمود. اگر کدهای هاش برابر بودند، کاربر اجازه وارد شدن می‌گرفت. متأسفانه، وقتی شرکت با نسخه‌ی جدید از **CLR** بروزرسانی شد، متد **GetHashCode** از **String**، تغییر یافته بود و اکنون مقدار کد هاش متفاوتی برمی‌گرداند. نتیجه نهایی این بود که هیچ کاربری دیگر نمی‌توانست به وب سایت وارد شود.

نوع اصلی dynamic

سی‌شارپ یک زبان برنامه‌نویسی نوع-امن است. این یعنی تمام عبارتها به یک نمونه از یک نوع تبدیل می‌شوند و کامپایلر تنها کدی تولید می‌کند که عملیاتی انجام دهد که برای این نوع معتبر باشد. مزیت یک زبان برنامه‌نویسی نوع-امن بر یک زبان برنامه‌نویسی غیر نوع-امن اینست که بسیاری از خطاهای برنامه‌نویس در زمان کامپایل یافت می‌شود و کمک می‌کند تا قبل از اجرای آن مطمئن شوید که کد صحیح است. به علاوه، زبان‌های زمان کامپایل، کدی کوچکتر و سریعتر تولید می‌کنند چون در زمان کامپایل فرضیاتی را در نظر می‌گیرند و این فرضیات را در **IL** و متادیتای خروجی لحاظ می‌کنند.

هر چند، مواقع بسیاری هست که یک برنامه باید با اطلاعاتی کار کند که تا زمان اجرا از آن‌ها بی‌خبر است. در حالیکه شما می‌توانید از زبان‌های برنامه‌نویسی نوع-امن (مثل سی‌شارپ) برای کار با این اطلاعات استفاده کنید، اما نحو مورد استفاده بسیار بد خواهد شد. مخصوصاً چون می‌خواهید با رشته‌های بسیاری کار کنید، سرعت نیز پایین می‌آید. اگر یک برنامه فقط سی‌شارپی می‌نویسید، آنگاه تنها راه شما برای کار کردن با اطلاعاتی که در زمان اجرا مشخص می‌شوند، استفاده از رفلکشن (بحث شده در فصل ۲۳) است. به هر حال، برنامه‌نویسان زیادی از سی‌شارپ برای ارتباط با کامپونتهایی که در سی-شارپ پیاده‌سازی نشده‌اند، استفاده می‌کنند. برخی از این کامپونتها می‌توانند زبان‌های پویای دات‌نت مثل **Python** یا **Ruby** یا اشیاء **COM** که رابط **IDispatch** را پشتیبانی می‌کنند (که احتمالاً در **C++** اصلی پیاده‌سازی شده اند)، یا اشیاء **HTML Document Object Model (DOM)** (پیاده‌سازی شده توسط زبان‌ها و تکنولوژی‌های مختلف) باشند. ارتباط با اشیاء **HTML DOM** بخصوص هنگام ساخت برنامه‌های **Microsoft Silverlight** کاربرد دارد.

برای آسان کردن کار برنامه‌نویسان در استفاده از رفلکشن یا ارتباط با دیگر کامپونتها، کامپایلر سی‌شارپ روشی برای علامت زدن نوع یک عبارت به عنوان **dynamic** ارائه می‌کند. شما همچنین می‌توانید نتیجه یک عبارت را در متغیری بگذارید و نوع متغیر را **dynamic** تعریف کنید. سپس عبارت/متغیر **dynamic** می‌تواند برای درخواست یک عضو مثل یک فیلد، یک ویژگی/ایندکسر، یک متد، نماینده (**delegate**) و عملگرهای یکنانه/دوتایی/تبدیل، مورد استفاده قرار بگیرد. وقتی کد شما با استفاده از یک عبارت/متغیر پویا، یک عضو را درخواست می‌کند، کامپایلر کد **IL** خاصی که عملکرد مورد نظر را

توصیف کند، تولید می‌نماید. این کد خاص، با عنوان **payload** اطلاق می‌شود. در زمان اجرا، این کد، دقیقاً عملیاتی که باید بر روی نوع واقعی شی که عبارت/متغیر **dynamic** به آن اشاره دارد را تعیین می‌کند.

کد زیر آنچه گفته شد را نشان می‌دهد:

```
Private static class DynamicDemo {
    public static void Main() {
        for (Int32 demo = 0; demo < 2; demo++) {
            dynamic arg = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
            dynamic result = Plus(arg);
            M(result);
        }
    }

    private static dynamic Plus(dynamic arg) { return arg + arg; }

    private static void M(Int32 n) { Console.WriteLine("M(Int32): " + n); }
    private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

من وقتی **Main** را اجرا می‌کنم، خروجی زیر را می‌گیرم:

```
M(Int32): 10
M(String): AA
```

برای درک آنچه رخ داد، با نگاهی به متد **Plus** آغاز می‌کنیم. این متد، پارامترش را از نوع **dynamic** تعریف کرده است و درون متد، آرگومان به عنوان دو عملوند برای عملگر باینری **+** استفاده شده است. چون **dynamic arg** است، کامپایلر سی‌شارپ کد پیلودی را تولید می‌کند که نوع واقعی **arg** را در زمان اجرا تعیین کرده و مشخص می‌کند عملگر **+** واقعا چه کاری باید انجام دهد.

وقتی برای اولین بار **Plus** فراخوانی می‌شود، **5** (یک **Int32**) بدان ارسال می‌شود. پس **Plus** مقدار **10** (باز هم یک **Int32**) را به فراخوانی کننده‌اش برمی‌گرداند. این عملیات نتیجه را در متغیر **result** (که از نوع **dynamic** تعریف شده است) ذخیره می‌کند. سپس متد **M** فراخوانی می‌شود و **result** به آن ارسال می‌گردد. برای فراخوانی **M**، کامپایلر کد پیلودی را تولید می‌کند که در زمان اجرا، نوع واقعی مقدار ارسالی به **M** را تعیین کرده و مشخص می‌کند کدام سربارگذاری از متد **M** باید صدا زده شود. وقتی **result** حاوی یک **Int32** است، سربارگذاری از **M** که یک پارامتر **Int32** دارد، صدا زده می‌شود.

برای بار دوم که **Plus** فراخوانی می‌شود، **"A"** (یک **String**) ارسال می‌شود پس **Plus** مقدار **"AA"** (نتیجه اتصال **A** با خودش) را به فراخوانی کننده‌اش برمی‌گرداند. که این نتیجه را در متغیر **result** قرار می‌دهد. سپس متد **M** مجدداً صدا زده می‌شود و **result** به آن ارسال می‌گردد. این بار، کد پیلودی نوع واقعی پارامتر ارسالی به **M** را **String** تشخیص می‌دهد و سربارگذاری‌ای از **M** که یک پارامتر **String** دریافت می‌کند را صدا می‌زند.

وقتی نوع یک فیلد، پارامتر متد، نوع برگشتی متد یا متغیرهای محلی، **dynamic** تعیین می‌شود، کامپایلر این نوع را به نوع **System.Object** تبدیل می‌کند و یک نمونه از **System.Runtime.CompilerServices.DynamicAttribute** را بر فیلد، پارامتر یا نوع برگشتی در متادیتا اعمال می‌کند. اگر یک متغیر محلی، **dynamic** تعیین شود، نوع متغیر از نوع **Object** خواهد بود اما **DynamicAttribute** بر متغیر محلی اعمال نمی‌شود چون کاربردش فقط داخل متد است. چون **dynamic** در حقیقت همان **Object** است، شما نمی‌توانید متدهایی بنویسید که تفاوت امضای آن‌ها فقط بین **dynamic** و **object** باشد.

هنگام تعیین آرگومان‌های نوع جنریک برای یک کلاس جنریک (نوع ارجاعی)، یک ساختار (یک نوع مقداری)، یک رابط، یک نماینده، یا یک متد می‌توانید از **dynamic** استفاده کنید. وقتی این کار را می‌کنید، کامپایلر، **dynamic** را به **Object** تبدیل کرده و هر جا در متادیتا لازم باشد، **DynamicAttribute** را اعمال می‌کند. توجه داشته باشید که کد جنریکی که از آن استفاده می‌کنید، قبلاً کامپایل شده است و نوع را **Object** در نظر می‌گیرد. هیچ عملیات اضافی پویایی انجام نمی‌شود چون کامپایلر کد پیلودی در کد جنریک تولید نکرده است.

هر عبارت را می‌توان صریحا به **dynamic** تبدیل کرد، چون تمام عبارتها منجر به جوابی می‌شوند که از **Object** مشتق شده اند.^{۴۴} به صورت عادی، کامپایلر اجازه‌ی نوشتن کدی که یک عبارت را به صورت ضمنی از **Object** به نوع دیگری تبدیل کند را نمی‌دهد؛ شما باید از نحو تبدیل صریح استفاده کنید. هر چند، کامپایلر اجازه‌ی تبدیل یک عبارت از **dynamic** به نوع دیگر را با استفاده از نحو تبدیل ضمنی می‌دهد.

```
Object o1 = 123;           // OK: Implicit cast from Int32 to Object (boxing)
Int32 n1 = o;             // Error: No implicit cast from Object to Int32
Int32 n2 = (Int32) o;     // OK: Explicit cast from Object to Int32 (unboxing)
```

```
dynamic d1 = 123;        // OK: Implicit cast from Int32 to dynamic (boxing)
Int32 n3 = d;           // OK: Implicit cast from dynamic to Int32 (unboxing)
```

در حالیکه کامپایلر اجازه‌ی حذف تبدیل صریح هنگام تبدیل از **dynamic** به نوع دیگر را می‌دهد، CLR برای تامین امنیت نوع، تبدیل را در زمان اجرا بازبینی می‌کند. اگر نوع شی با تبدیل سازگار نباشد، CLR یک اکسپشن **InvalidCastException** تولید می‌کند.

توجه داشته باشید که نتیجه ارزیابی یک عبارت **dynamic**، یک عبارت پویاست. این کد را بررسی کنید:

```
dynamic d = 123;
var result = M(d); // Note: 'var result' is the same as 'dynamic result'
```

در اینجا، کامپایلر به کد اجازه کامپایل شدن را می‌دهد چون در زمان کامپایل نمی‌داند کدام متد **M** فراخوانی می‌شود. بنابراین، نوع نتیجه‌ی برگشتی از **M** را هم نمی‌داند. پس، کامپایلر فرض می‌کند که خود **result** از نوع **dynamic** است. شما می‌توانید این نکته را با نگه داشتن موس خود روی **var** در ویرایشگر ویزوال استودیو امتحان کنید؛ پنجره‌ی IntelliSense بیان می‌کند که:

'dynamic: Represents an object whose operation will be resolved at runtime.'

اگر متد **M** که در زمان اجرا درخواست می‌شود، نوع برگشتی **void** داشته باشد آنگاه هیچ اکسپشنی تولید نشده و به جای آن مقدار **null** در **result** قرار می‌گیرد.

مهم dynamic را با **var** قاطی نکنید. تعریف یک متغیر محلی با عنوان **var** تنها یک میانبر نحوی است که کامپایلر، نوع را از روی عبارت تعیین می‌کند. کلمه کلیدی **var** تنها برای تعریف متغیرهای محلی درون یک متد کاربرد دارد در حالیکه کلمه کلیدی **dynamic** برای متغیرهای محلی، فیلدها و آرگومان‌ها استفاده می‌شود. شما نمی‌توانید یک عبارت را به **var** تبدیل کنید اما می‌توانید یک عبارت را به **dynamic** تبدیل کنید. شما باید صریحا یک متغیر که با **var** تعریف شده است را مقداردهی اولیه کنید، در حالیکه مجبور نیستید یک متغیر تعریف شده با **dynamic** را مقداردهی اولیه کنید. برای اطلاعات بیشتر درباره‌ی **var** سی‌شارپ، به بخش "متغیرهای محلی تعریف شده‌ی ضمنی" در فصل ۹ "پارامترها" مراجعه کنید.

هر چند، هنگام تبدیل از **dynamic** به نوع ثابت دیگر، نوع نتیجه البته نوع ثابت است. به طریق مشابه، هنگام ساخت یک نوع با ارسال یک یا بیشتر آرگومان **dynamic** به سازنده اش، نتیجه از نوع شی در حال ساخت است:

```
dynamic d = 123;
var x = (Int32) d;           // Conversion: 'var x' is the same as 'Int32 x'
var dt = new DateTime(d);    // Construction: 'var dt' is the same as 'DateTime dt'
```

اگر یک عبارت **dynamic** به عنوان مجموعه در یک عبارت **foreach** یا به عنوان یک منبع در عبارت **using** تعیین شده باشد، کامپایلر کدی تولید می‌کند که تلاش می‌کند عبارت را به ترتیب به رابط غیرجنریک **System.IEnumerable** یا رابط **System.IDisposable** تبدیل کند. اگر تبدیل موفقیت آمیز بود، عبارت استفاده می‌شود و کد به خوبی اجرا خواهد شد. اگر تبدیل شکست خورد، یک اکسپشن **Microsoft.CSharp.RuntimeBinder.RuntimeBinderExpression** تولید می‌شود.

^{۴۴} و همچون همیشه نوع‌های مقداری، بسته بندی می‌شوند.

مهم یک عبارت **dynamic** واقعا از همان نوع **System.Object** است. کامپایلر فرض می‌کند هر عملیاتی که بر روی عبارت انجام می‌دهید مجاز است، پس خطا یا هشدار تولید نمی‌کند. هر چند، اگر شما سعی در انجام عملیات غیر مجاز داشته باشید، اکسپشن‌ها در زمان اجرا تولید می‌شوند. به علاوه، ویژوال استودیو نمی‌تواند هیچ پشتیبانی IntelliSense برای کمک به شما در کدنویسی یک عبارت **dynamic** ارائه کند. شما نمی‌توانید یک متد گسترشی (extension method) (بحث شده در فصل ۸ "متدها") که **dynamic** را گسترش دهد تعریف کنید اگرچه می‌توانید برای گسترش **Object** این کار را بکنید. و شما نمی‌توانید یک عبارت لامبدا یا متد ناشناس (هر دو در فصل ۱۷ "نماینده‌ها" بحث می‌شوند) را به عنوان آرگومان به یک متد دارای آرگومان **dynamic** ارسال کنید چون کامپایلر نمی‌تواند نوع مورد استفاده را تعیین کند.

نمونه‌ای از کد سی‌شارپ که از **COM IDispatch** برای ساخت یک **workbook** در اکسل مایکروسافت و ذخیره‌ی یک رشته در خانه **A1** استفاده می‌کند را در زیر می‌بینید:

```
using Microsoft.Office.Interop.Excel;
```

```
...
```

```
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    ((Range)excel.Cells[1, 1]).Value = "Text in cell A1"; // Put this string in cell A1
}
```

بدون نوع **dynamic**. مقدار برگشتی از **excel.Cells[1, 1]** از نوع **Object** است که باید به **Range** تبدیل شود پیش از آنکه ویژگی **Value** از آن قابل استفاده باشد. هر چند، هنگام تولید یک اسمبلی پوشاننده قابل فراخوانی برای یک شی **COM**، هر استفاده از **VARIANT** در متد **COM** به **dynamic** تبدیل می‌شود. این کار، پویا سازی dynamification نامیده می‌شود. بنابراین چون **excel.Cells[1, 1]** از نوع **dynamic** است پیش از آنکه به ویژگی **Value** دسترسی داشته باشد، نیاز ندارید آن را صریحا به نوع **Range** تبدیل نمایید. پویاسازی، کد را هنگام کار با اشیاء **COM** به مراتب ساده تر می‌کند. در اینجا کد ساده تر را می‌بینیم:

```
using Microsoft.Office.Interop.Excel;
```

```
...
```

```
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    excel.Cells[1, 1].Value = "Text in cell A1"; // Put this string in cell A1
}
```

کد زیر نشان می‌دهد چگونه از رفلکشن برای فراخوانی یک متد ("Contains") بر روی یک **String** هدف ("Jeffery Richter") و ارسال یک آرگومان **String** ("ff") و ذخیره‌ی نتیجه **Int32** در یک متغیر محلی (**result**) استفاده کنید.

```
Object target = "Jeffery Richter";
Object arg = "ff";
```

```
// Find a method on the target that matches the desired argument types
Type[] argTypes = new Type[] { arg.GetType() };
MethodInfo method = target.GetType().GetMethod("Contains", argTypes);
```

```
// Invoke the method on the target passing the desired arguments
Object[] arguments = new Object[] { arg };
Boolean result = Convert.ToBoolean(method.Invoke(target, arguments));
```

با استفاده از نوع **dynamic** سی‌شارپ، این کد با نحو بسیار بهتر این گونه می‌شود:

```
dynamic target = "Jeffrey Richter";
dynamic arg = "ff";
Boolean result = target.Contains(arg);
```

قبلاً اشاره کردم که کامپایلر سی شارپ، کد پیلودی تولید می‌کند که در زمان اجرا نوع عملیاتی که باید انجام شود را طبق نوع واقعی یک شی تعیین می‌کند. این کد پیلود، از کلاسی که با عنوان متصل کننده‌ی زمان اجرا **runtime binder** شناخته می‌شود، استفاده می‌کند. زبان‌های برنامه‌نویسی مختلف، متصل کننده‌های زمان اجرای خود را تعریف می‌کنند که قوانین آن زبان را در بردارد. کد متصل کننده‌ی زمان اجرای سی شارپ در اسمبلی **Microsoft.CSharp.dll** قرار دارد و شما باید هنگام استفاده از کلمه کلیدی **dynamic**، این اسمبلی را ارجاع دهید. این اسمبلی در فایل جواب پیش فرض کامپایلر **CSC.rsp** ارجاع داد شده است. کد درون این اسمبلی است که می‌داند (در زمان اجرا) کدی تولید کند که عملگر **+** وقتی بر دو شی **Int32** اعمال می‌شود، آن‌ها را با هم جمع کند و وقتی بر دو شی **String** اعمال می‌شود آن‌ها را با هم متصل نماید.

در زمان اجرا، اسمبلی **Microsoft.CSharp.dll** باید درون **AppDomain** بارگذاری شود که سرعت برنامه را کاهش و مصرف حافظه‌ی آن را افزایش می‌دهد. اگر از **dynamic** استفاده می‌کنید تا در تبادل با کامپوننت‌های **COM** به شما کمک کند، اسمبلی **System.Dynamic.dll** نیز بارگذاری خواهد شد. و وقتی کد پیلود اجرا می‌شود، کد پویا را در زمان اجرا تولید می‌کند، این کد در یک اسمبلی در حافظه به نام "اسمبلی متدهای پویا که به صورت ناشناس میزبانی می‌شوند" "**Anonymously Hosted Dynamic Methods Assembly**" خواهد بود. هدف این کد افزایش سرعت توزیع پویا در سناریوهایی است که یک سایت فراخوانی خاص، درخواست‌های زیادی با آرگومان‌های پویای دارای نوع زمان اجرایی یکسان، دارد. به خاطر تمام عملیات‌های اضافی همراه با ویژگی ارزیابی پویا در سی شارپ شما باید با هوشیاری تصمیم بگیرید که نحو کد شما با ویژگی **dynamic** به قدر کافی ساده شده است که ارزش کاهش سرعت برای بارگذاری تمام این اسمبلی‌ها و حافظه مصرفی اضافی را داشته باشد. اگر تنها در چند جای برنامه به رفتار **dynamic** نیاز دارید، بسیار بهتر است که از روش قدیمی با فراخوانی متدهای رفلکشن (برای اشیاء مدیریت شده) یا تبدیل دستی (برای اشیاء **COM**) استفاده کنید.

در زمان اجرا، متصل کننده‌ی زما اجرای سی شارپ، یک عبارت پویا را بر اساس نوع شی در زمان اجرا تحلیل می‌کند. متصل کننده اول بررسی می‌کند آیا نوع، رابط **IDynamicMetaObjectProvider** را پیاده‌سازی کرده است. اگر نوع این رابط را پیاده‌سازی نموده، سپس متد **GetMetaObject** از رابط فراخوانی می‌شود که یک نوع مشتق شده از **DynamicMetaObject** را برمی‌گرداند. این نوع می‌تواند تمام اعضا، متدها و اتصال‌های عملگر را برای شی پردازش کند. هر دو رابط **IDynamicMetaObjectProvider** و کلاس پایه **DynamicMetaObject** در فضای نام **System.Dynamic** تعریف شده‌اند و هر دو در اسمبلی **System.Core.dll** قرار دارند.

زبان‌های پویا، مثل **Python** و **Roby** به نوع‌هایشان، نوع‌های مشتق شده از **DynamicMetaObject** را می‌دهند تا آن‌ها به روشی مناسب توسط دیگر زبان‌های برنامه‌نویسی (مثل سی شارپ) در دسترس باشند. به طریق مشابه، هنگام دسترسی به یک کامپوننت **COM**، متصل کننده زمان اجرای سی-شارپ از یک نوع مشتق شده از **DynamicMetaObject** استفاده می‌کند که می‌داند چگونه با یک کامپوننت **COM** ارتباط برقرار کند. نوع **COM** مشتق شده از **DynamicMetaObject** در اسمبلی **System.Dynamic.dll** تعریف شده است.

اگر نوع شی مورد استفاده در عبارت پویا، رابط **IDynamicMetaObjectProvider** را پیاده‌سازی نکند، آنگاه کامپایلر سی شارپ با شی به عنوان یک نمونه از یک نوع معمولی تعریف شده توسط سی شارپ رفتار کرده و عملیات بر روی شی را با رفلکشن انجام خواهد داد.

فصل ۶: مبانی نوع و عضو

در فصل ۴ و ۵، بر روی نوع‌ها و اینکه چه عملیاتی بر روی تمام نمونه‌های یک نوع به صورت تضمینی وجود دارد، تمرکز کردم. همچنین توضیح دادم چگونه همه‌ی نوع‌ها در دو دسته قرار می‌گیرند: نوع‌های ارجاعی و نوع‌های مقداری. در این فصل و فصل‌های آتی در این بخش از کتاب، نشان خواهم داد که چگونه نوع‌هایی با گونه‌های مختلف از اعضای قابل تعریف در یک نوع، طراحی کنید.

در فصل ۷ تا ۱۱، اعضای مختلف را با جزییات بررسی خواهم کرد.

گونه‌های مختلف اعضای نوع

یک نوع می‌تواند صفر یا بیشتر گونه‌های زیر از اعضا را داشته باشد:

- **ثابت‌ها** یک ثابت یک نماد است که یک مقدار داده‌ای بدون تغییر را شناسایی می‌کند. این نمادها اغلب برای خواناتر شدن و نگهداری آسانتر کد استفاده می‌شوند. ثابت‌ها همواره با یک نوع و نه با یک نمونه از نوع همراه هستند. منطقی‌تر است ثابت‌ها همیشه عضوهای استاتیک هستند. در فصل ۷ "ثابت‌ها و فیلدها" بحث می‌شوند.
- **فیلدها** یک فیلد یک مقدار داده‌ای فقط خواندنی یا خواندنی/نوشتنی را نمایش می‌دهد. یک فیلد می‌تواند استاتیک باشد که در این حالت به عنوان بخشی از وضعیت نوع در نظر گرفته می‌شود. یک فیلد می‌تواند نمونه (غیر استاتیک) باشد که در این حالت به عنوان بخشی از وضعیت شی در نظر گرفته می‌شود. من قویا شما را تشویق می‌کنم که فیلدهای خود را خصوصی کنید تا وضعیت نوع یا شی توسط کدی خارج از نوع تعریف کننده آن، خراب نشود. فیلدها در فصل ۷ بحث می‌شوند.
- **سازنده‌های نمونه** یک سازنده‌ی نمونه یک متد خاص برای مقداردهی اولیه فیلدهای نمونه‌ی شی جدید به یک وضعیت اولیه صحیح می‌باشد. این‌ها در فصل ۷ "متدها" بحث می‌شوند.
- **سازنده‌های نوع** یک سازنده‌ی نوع یک متد خاص برای مقداردهی اولیه فیلدهای استاتیک یک نوع به یک وضعیت اولیه صحیح می‌باشد. در فصل ۸ بحث می‌شوند.
- **متدها** یک متد تابعی است که عملیاتی انجام می‌دهد که وضعیت یک نوع (متد استاتیک) یا یک شی (متد نمونه) را تغییر داده یا از آن اطلاع پیدا می‌کند. متدها معمولا فیلدهای نوع یا شی را نوشته و می‌خوانند. در فصل ۸ بحث می‌شوند.
- **سربارگذاری عملگرها** یک سربارگذاری عملگر یک متد است که تعریف می‌کند چگونه یک شی هنگامیکه عملگرهای خاصی بر شی اعمال می‌شود باید دستکاری گردد. چون همه‌ی زبان‌های برنامه‌نویسی، سربارگذاری عملگرها را پشتیبانی نمی‌کنند، متدهای سربارگذاری عملگر بخشی از مشخصات مشترک زبان (CLS) نیست. در فصل ۸ بحث می‌شوند.
- **عملگرهای تبدیل** یک عملگر تبدیل یک متد است که تعریف می‌کند چگونه ضمنی یا صریحا یک شی را از نوعی به نوع دیگر تبدیل کند. همانند متدهای سربارگذاری عملگرها، همه‌ی زبان‌های برنامه‌نویسی، عملگرهای مقایسه را پشتیبانی نمی‌کنند، پس آن‌ها بخشی از CLS نیستند. در فصل ۸ بحث می‌شوند.
- **ویژگی‌ها** یک ویژگی مکانیزی است که به صورت آسان با نحو شبیه به فیلدها، اجازه خواندن و نوشتن وضعیت منطقی یک نوع (ویژگی استاتیک) یا شی (ویژگی نمونه) را در حالی می‌دهد که مطمئن می‌شود وضعیت خراب نمی‌شود. ویژگی‌ها می‌توانند بدون پارامتر (بسیار رایج) یا با پارامتر (بسیار رایج، اما اغلب با کلاس‌های مجموعه استفاده می‌شوند) باشند. در فصل ۱۰ "ویژگی‌ها" بحث می‌شوند.
- **رویدادها** یک رویداد استاتیک مکانیزی است که به یک نوع اجازه‌ی ارسال خبر به یک یا بیشتر از متدهای نمونه یا استاتیک آن را می‌دهد. یک رویداد نمونه (غیر استاتیک) مکانیزی است که به یک شی اجازه‌ی ارسال خبر به یک یا بیشتر متدهای نمونه یا استاتیک آن را می‌دهد. رویدادها اغلب در پاسخ به یک تغییر وضعیت در شی یا نوع ارائه کننده‌ی رویداد، فعال می‌شوند. یک رویداد از دو متد تشکیل شده است که اجازه می‌دهد متدهای استاتیک یا نمونه، علاقه‌ی خود به رویداد را ثبت کرده و این علاقه را حذف کنند. علاوه بر این دو متد، رویدادها معمولا از یک فیلد نماینده برای نگهداری مجموعه‌ی متدهای ثبت شده استفاده می‌کنند. در فصل ۱۱ "رویدادها" بحث می‌شوند.

▪ **نوع‌ها** یک نوع می‌تواند نوع‌های دیگری درون خود تعریف کند. این دیدگاه معمولاً برای شکستن یک نوع بزرگ و پیچیده به بلوک‌های سازنده‌ی کوچک جهت ساده کردن پیاده‌سازی، صورت می‌گیرد.

مجدداً، هدف از این فصل توضیح اعضای مختلف با جزئیات نیست بلکه نقاط مشترک این اعضا را بیان می‌کند.

بدون توجه به زبان برنامه‌نویسی مورد استفاده شما، کامپایلر متناظر باید کد شما را پردازش کرده و متادیتا و کد IL را برای هر گونه از اعضای لیست قبلی تولید کند. قالب متادیتا بدون توجه به زبان برنامه‌نویسی استفاده شده، یکسان است و این ویژگی چیزی است که CLR را یک اجرا کننده‌ی زبان مشترک می‌کند. متادیتا اطلاعات مشترکی است که تمام زبان‌ها تولید و مصرف می‌کنند، کد نوشته شده در یک زبان برنامه‌نویسی را به آسانی قادر به دسترسی به کد نوشته شده در زبان برنامه‌نویسی کاملاً متفاوت می‌کند.

این قالب مشترک متادیتا توسط CLR نیز استفاده می‌شود که تعیین می‌کند ثابت‌ها، فیلدها، سازنده‌ها، متدها، ویژگی‌ها و رویدادها چگونه در زمان اجرا رفتار کنند. به بیان ساده، متادیتا کلید تمام پلتفرم‌های برنامه‌نویسی دات‌نت فریم‌ورک مایکروسافت است که یکپارچگی زبان‌ها، نوع‌ها و اشیاء را به آسانی فراهم می‌کند.

کد سی‌شارپ زیر تعریف یک نوع که تمام اعضای ممکن را دارد نشان می‌دهد. کد نشان داده شده در اینجا (همراه با هشدار) کامپایل می‌شود، اما نمایانگر یک نوع که شما معمولاً می‌سازید نیست چون اکثر متدها کار خاصی انجام نمی‌دهند. اکنون، من فقط می‌خواهم به شما نشان دهم کامپایلر چگونه این نوع و اعضایش را به متادیتا ترجمه می‌کند. یکبار دیگر، من تک تک اعضا را در چند فصل آتی توضیح خواهم داد.

```
using System;
```

```
public sealed class SomeType { // 1

    // Nested class
    private class SomeNestedType { } // 2

    // Constant, read-only, and static read/write field
    private const Int32 c_SomeConstant = 1; // 3
    private readonly String m_SomeReadOnlyField = "2"; // 4
    private static Int32 s_SomeReadWriteField = 3; // 5

    // Type constructor
    static SomeType() { } // 6

    // Instance constructors
    public SomeType(Int32 x) { } // 7
    public SomeType() { } // 8

    // Instance and static methods
    private String InstanceMethod() { return null; } // 9
    public static void Main() {} // 10

    // Instance property
    public Int32 SomeProp { // 11
        get { return 0; } // 12
        set { } // 13
    }

    // Instance parameterful property (indexer)
    public Int32 this[String s] { // 14
        get { return 0; } // 15
    }
}
```

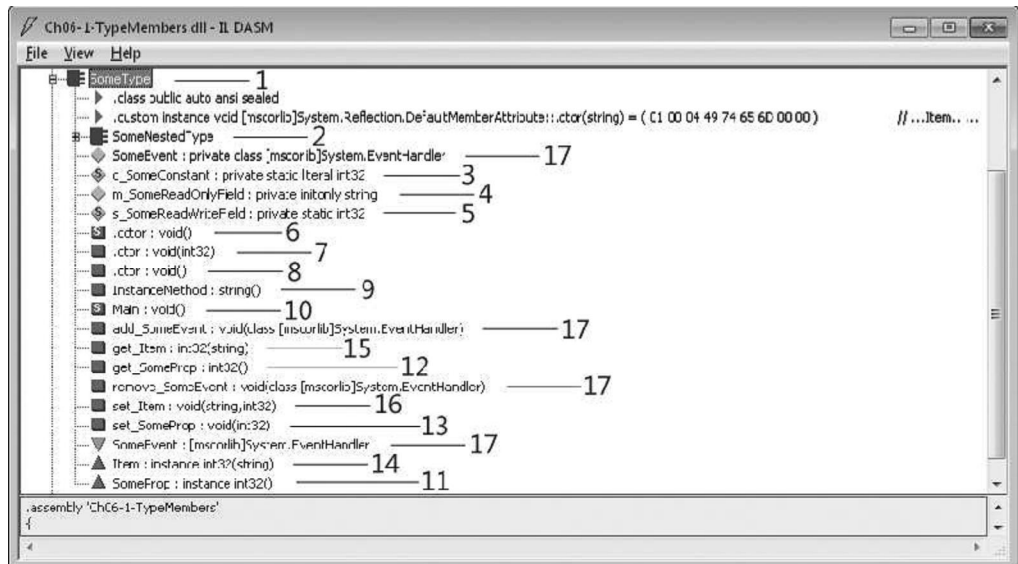
```

set { } // 16
}

// Instance event
public event EventHandler SomeEvent; // 17
}

```

اگر شما این نوع را کامپایل کنید و متادیتای آن را با ILDasm.exe بررسی کنید، خروجی نمایش داده شده در شکل ۱-۶ را خواهید دید.



شکل ۱-۶ ILDasm.exe در حال نمایش متادیتای کد قبلی

توجه کنید تمام اعضای تعریف شده در سورس کد باعث می‌شوند کامپایلر برخی اطلاعات متادیتا تولید کند. در واقع، برخی از اعضا باعث می‌شوند کامپایلر، اعضای اضافی علاوه بر متادیتای اضافی نیز تولید کند. برای نمونه، عضو رویداد (۱۷) باعث می‌شود کامپایلر یک فیلد دو متد و برخی اطلاعات متادیتای اضافی تولید کند. من انتظار ندارم شما کاملاً آنچه اینجا می‌بینید را درک کنید. اما به تدریج که فصل‌های آینده را بخوانید، من شما را تشویق به بازگشت به این مثال می‌کنم تا ببینید چگونه عضو تعریف می‌شود و چه اثری بر متادیتای تولیدی توسط کامپایلر دارد.

پدیداری نوع

وقتی یک نوع را در میدان^{۴۵} یک فایل (در مقابل تعریف یک نوع تودرتو درون نوعی دیگر) تعریف می‌کنید، شما می‌توانید پدیداری نوع^{۴۶} را با **public** یا **internal** تعیین کنید. یک نوع **public** برای تمام کد درون اسمبلی تعریف کننده و همچنین کد نوشته شده در دیگر اسمبلی‌ها قابل دیدن است. یک نوع **internal** در معرض دید تمام کد نوشته شده درون اسمبلی تعریف کننده است ولی نوع در معرض دید کد نوشته شده در دیگر اسمبلی‌ها قرار ندارد. اگر هنگام تعریف یک نوع شما صریحاً یکی از این دو را تعیین نکنید، کامپایلر سی‌شارپ پدیداری نوع را **internal** (پدیداری محدودتر) تعیین می‌کند. نمونه‌هایی را ببینیم:

```
using System;
```

```
// The type below has public visibility and can be accessed by code
// in this assembly as well as code written in other assemblies.
public class ThisIsAPublicType { ... }
```

```
// The type below has internal visibility and can be accessed by code
```

⁴⁵ scope

⁴⁶ Type Visibility

```
// in this assembly only.
internal class ThisIsAnInternalType { ... }

// The type below is internal because public/internal
// was not explicitly stated
class ThisIsAlsoAnInternalType { ... }
```

اسمبلی های دوست

سناریوی زیر را در نظر بگیرید: یک شرکت، یک تیم به نام **TeamA** دارد که تعدادی نوع‌های کاربردی را در یک اسمبلی تعریف می‌کند و آن‌ها انتظار دارند که این نوع‌ها توسط اعضای از تیم دیگر، **TeamB**، استفاده شوند. به دلایل مختلف مثل زمانبندی یا موقعیت جغرافیایی این دو تیم نمی‌توانند همه‌ی نوع-هایشان را در یک اسمبلی جای دهند و به جای آن هر تیم اسمبلی خودش را تولید می‌کند.

برای آنکه اسمبلی **TeamB** بتواند از نوع‌های **TeamA** استفاده کند، **TeamA** باید تمام نوع‌های کاربردی خود را **public** تعریف کند. اما این یعنی نوع‌های آن‌ها در معرض دید هر اسمبلی دیگری قرار دارد و برنامه‌نویسان یک شرکت دیگر می‌توانند کدی بنویسند که از نوع‌های کاربردی عمومی استفاده کند و این اصلاً مطلوب نیست. شاید نوع‌های کاربردی فرضیاتی را لحاظ کنند که **TeamB** هنگام نوشتن کدی که از نوع‌های **TeamA** استفاده می‌کند، از آن مطمئن است. آنچه می‌خواهیم اینست که به روشی، **TeamA** نوع‌هایش را **internal** تعریف کند در حالیکه به **TeamB** اجازه‌ی دسترسی به نوع-هایش را بدهد. CLR و سی‌شارپ این ویژگی را از طریق اسمبلی‌های دوست **friend assemblies** پشتیبانی می‌کنند. این ویژگی اسمبلی دوست، همچنین هنگامیکه که می‌خواهید یک اسمبلی حاوی کدی باشد که برای اجرای تست روی نوع‌های درونی در اسمبلی دیگری استفاده شود، مفید است.

وقتی یک اسمبلی ساخته می‌شود، می‌تواند اسمبلی‌هایی که "دوست" در نظر می‌گیرد را با استفاده از صفت **InternalsVisibleTo** تعریف شده در فضای نام **System.Runtime.CompilerServices**، تعیین کند. این صفت دارای یک پارامتر رشته‌ای است که نام و کلید عمومی اسمبلی دوست را می‌پذیرد (رشته‌ای که شما به صفت ارسال می‌کنید نباید شامل یک نسخه، فرهنگ یا معماری پردازنده باشد). توجه کنید که اسمبلی‌های دوست می‌توانند به تمام نوع‌های **internal** یک اسمبلی و همچنین تمام اعضای **internal** این نوع‌ها دسترسی داشته باشند. نمونه‌ای را در اینجا می‌بینید که یک اسمبلی چگونه می‌تواند دو اسمبلی قوی‌نام دیگر به نام "Microsoft" و "Wintellect" را به عنوان اسمبلی‌های دوست تعیین کند:

```
using System;
using System.Runtime.CompilerServices; // For InternalsVisibleTo attribute

// This assembly's internal types can be accessed by any code written
// in the following two assemblies (regardless of version or culture):
[assembly:InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]
[assembly:InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]
```

```
internal sealed class SomeInternalType { ... }
internal sealed class AnotherInternalType { ... }
```

دسترسی به نوع‌های **internal** اسمبلی فوق از یک اسمبلی دوست ساده است. برای نمونه، در کد زیر می‌بینید چگونه اسمبلی دوست به نام "Wintellect" با کلید عمومی "12345678...90abcdef" می‌تواند به نوع درونی **SomeInternalType** در اسمبلی فوق دسترسی پیدا کند:

```
using System;

internal sealed class Foo {
    private static Object SomeMethod() {
        // This "wintellect" assembly accesses the other assembly's
        // internal type as if it were a public type
        SomeInternalType sit = new SomeInternalType();
        return sit;
    }
}
```

چون اعضای **internal** از نوعها در یک اسمبلی، در دسترس اسمبلی دوست قرار می‌گیرند باید با دقت دسترس پذیری اعضای نوع خود را مشخص کرده و تعیین کنید کدام اسمبلی‌ها را دوست خود قرار می‌دهید. توجه کنید که کامپایلر سی‌شارپ نیاز دارد که شما سوئیچ **<file>:out/** را هنگام کامپایل اسمبلی دوست (اسمبلی‌ای که حاوی صفت **InternalsVisibleTo** نیست) تعیین کنید. سوئیچ نیاز است چون کامپایلر باید نام اسمبلی در حال کامپایل را بداند تا تعیین کند که اسمبلی تولیدی باید به عنوان یک اسمبلی دوست در نظر گرفته شود. شاید فکر کنید که کامپایلر سی‌شارپ خودش می‌تواند این را تعیین کند چون معمولا نام فایل خروجی را خودش تعیین می‌کند، اما، کامپایلر تا پایان کامپایل کد، تصمیمی درباره‌ی نام فایل خروجی نمی‌گیرد. پس نیاز به داشتن سوئیچ **<file>:out/**، سرعت کامپایل را به مقدار قابل توجهی افزایش می‌دهد.

همچنین اگر شما یک ماژول (در مقابل یک اسمبلی) را با سوئیچ **/t:module** سی‌شارپ کامپایل کنید و این ماژول بخواهد بخشی از یک اسمبلی دوست باشد شما باید ماژول را هم با سوئیچ کامپایلر سی‌شارپ **<string>:moduleassemblyname/** کامپایل کنید. این به کامپایلر می‌گوید که ماژول بخشی از کدام اسمبلی است تا کامپایلر بتواند به کد درون ماژول اجازه‌ی دسترسی به نوع‌های درونی اسمبلی دیگری را بدهد.

مهم ویژگی اسمبلی دوست تنها باید توسط اسمبلی‌هایی که در یک زمان خاص یا حتی با همدیگر عرضه می‌شوند، استفاده گردد. علت آنست که وابستگی میان اسمبلی‌های دوست به قدری بالاست که عرضه‌ی اسمبلی‌ها دوست در بازه‌های زمانی متفاوت احتمالا منجر به مشکلات عدم سازگاری می‌شود. اگر انتظار دارید که اسمبلی‌ها در زمان‌های مختلف عرضه شوند، سعی کنید کلاس‌های **public** طراحی کنید که هر اسمبلی بتواند از آن‌ها استفاده کند و دسترس پذیری را از طریق یک **LinkDemand** که نیاز به مجوز **StrongNameIdentityPermission** دارد، محدود کنید.

دسترس پذیری عضو

هنگام تعریف عضو یک نوع (که شامل نوع‌های تودرتو است)، شما می‌توانید دسترس پذیری عضو را تعیین کنید. دسترس پذیری یک عضو تعیین می‌کند چه اعضای توسط کد ارجاع کننده قابل دسترسی هستند. CLR مجموعه‌ای از تغییر دهنده^{۴۷}های دسترس پذیری را تعریف می‌کند. اما، هر زبان برنامه‌نویسی نحو و واژه‌ای را انتخاب می‌کند که می‌خواهد برنامه‌نویس‌ها هنگام اعمال دسترس پذیری به عضو، از آن استفاده کنند. برای نمونه CLR از واژه‌ی **Assembly** برای تعیین اینکه یک عضو در دسترس هر کدی درون همان اسمبلی باشد، استفاده می‌کند در حالیکه واژه‌ی سی‌شارپ برای آن، **internal** است.

جدول ۱-۶ شش تغییر دهنده‌ی دسترس پذیری که می‌توانند به یک عضو اعمال شوند را نشان می‌دهد. ردیف‌های جدول به ترتیب از بیشترین محدودیت (**Private**) به کمترین محدودیت (**Public**) است.

جدول ۱-۶ دسترس پذیری عضو

واژه‌ی CLR	واژه‌ی سی‌شارپ	توضیح
Private	private	عضو تنها توسط متدهای درون نوع تعریف کننده یا هر نوع تودرتو دسترس پذیر است.
Family	protected	عضو تنها توسط متدهای درون نوع تعریف کننده، هر نوع تودرتو، یا یکی از نوع‌های مشتق شده‌اش بدون توجه به اسمبلی، دسترس پذیر است.
Family and Assembly	(پشتیبانی نمی‌شود)	عضو تنها توسط متدهای درون نوع تعریف کننده، هر نوع تودرتو، یا توسط هر نوع مشتق شده که در همان اسمبلی تعریف شود، دسترس پذیر است.
Assembly	internal	عضو تنها توسط متدهای درون همان اسمبلی دسترس پذیر است.
Family or Assembly	protected internal	عضو توسط هر نوع تودرتو، هر نوع مشتق شده (بدون توجه به اسمبلی) یا هر متدی در اسمبلی تعریف کننده، دسترس پذیر است.
Public	public	عضو توسط تمام متدها در تمام اسمبلی‌ها دسترس پذیر است.

البته، برای هر عضو که در دسترس باشد، باید درون نوعی که در معرض دید است تعریف شود. برای نمونه، اگر **AssemblyA** یک نوع **internal** با یک متد **public** تعریف کند، کد درون **AssemblyB** نمی‌تواند متد **public** را صدا بزند چون نوع **internal** در معرض دید **AssemblyB** نیست.

⁴⁷ modifier

هنگام کامپایل کد، کامپایلر زبان مسئول بررسی آن است که کد به درستی نوع‌ها و اعضا را ارجاع می‌دهد. اگر کد، نوع یا عضوی را غیر صحیح ارجاع دهد، کامپایلر مسئول تولید پیام خطای مناسب است. به علاوه، کامپایلر JIT وقتی کد IL را در زمان اجرا به دستورات اصلی پردازنده کامپایل می‌کند مطمئن می‌شود که ارجاع‌ها به فیلدها و متدها مجاز باشند. برای نمونه، اگر کامپایلر JIT تشخیص دهد که کد به نادرستی سعی در دسترسی به یک فیلد یا متد خصوصی دارد، کامپایلر JIT به ترتیب یک **FieldAccessException** یا یک **MethodAccessException** تولید می‌کند.

بازبینی کد IL این اطمینان را می‌دهد که دسترس پذیری عضو ارجاعی به درستی در زمان اجرا لحاظ می‌شود حتی اگر یک کامپایلر زبان بررسی دسترس پذیری را نادیده بگیرد. احتمال دیگر این است که کامپایلر کدی را کامپایل کند که به یک عضو **public** از نوع دیگری (در اسمبلی دیگری) دسترسی پیدا کرده، اما در زمان اجرا، نسخه‌ی متفاوتی از اسمبلی بارگذاری شده است و در نسخه جدید، عضو **public** اکنون به **protected** یا **private** تغییر یافته است.

در سی‌شارپ، اگر شما صریحا دسترس پذیری یک عضو را تعیین نکنید، کامپایلر معمولا (اما نه همیشه) پیش فرض را بر انتخاب **private** (محدود ترین آن‌ها) می‌گذارد. CLR نیاز دارد که تمام اعضای یک نوع رابط، عمومی باشند. کامپایلر سی‌شارپ این را می‌داند و مانع از تعیین صریح دسترس پذیری بر اعضای رابط می‌شود، کامپایلر همه‌ی اعضا را برای شما **public** می‌کند.

اطلاعات بیشتر بخش “Declared Accessibility” را در مشخصات زبان سی‌شارپ جهت لیست کامل قوانین سی‌شارپ درباره‌ی آنکه چه دسترس پذیری‌هایی به اعضا و نوع‌ها می‌تواند اعمال شود و بسته به محیط تعریف آن‌ها سی‌شارپ چه پیش فرض‌هایی را دارد، ببینید.

پیش از این، متوجه خواهید شد که CLR یک دسترس پذیری به نام *Family and Assembly* ارائه می‌کند. اما، سی‌شارپ آن را در زبان به کار نمی‌برد. تیم سی‌شارپ احساس کردند که این دسترس پذیری برای اغلب موارد بلااستفاده است و تصمیم گرفتند آن را در زبان سی‌شارپ به کار نگیرند.

وقتی یک نوع مشتق شده، یک عضو تعریف شده در کلاس پایه‌اش را بازنویسی می‌کند، کامپایلر سی‌شارپ نیاز دارد که عضو اصلی و عضو بازنویسی شده، دارای دسترس پذیری یکسانی باشند. یعنی، اگر عضو در کلاس پایه، **protected** است، عضو بازنویسی شده در کلاس مشتق شده نیز **protected** باشد. هر چند، این یک محدودیت سی‌شارپ است و نه محدودیت CLR. هنگام مشتق کردن از یک کلاس پایه، CLR اجازه می‌دهد دسترس پذیری یک عضو محدودیت کمتری داشته باشد و نه محدودیت بیشتر. برای نمونه، یک کلاس می‌تواند یک متد **protected** تعریف شده در کلاس پایه‌اش را بازنویسی کند و متد بازنویسی شده را **public** (در دسترس تر) کند. اما، یک کلاس نمی‌تواند یک متد **protected** تعریف شده در کلاس پایه‌اش را بازنویسی کرده و متد بازنویسی شده را **private** (در دسترسی کمتر) قرار دهد. علت آنکه یک کلاس نمی‌تواند یک متد کلاس پایه‌اش را محدود تر کند آنست که کاربر کلاس مشتق شده همیشه می‌تواند آن را به نوع پایه تبدیل کند و به متد کلاس پایه دسترسی یابد. اگر CLR اجازه می‌داد متد مشتق شده دسترس پذیری کمتری داشته باشد، ادعایی کرده بود که قابل اجرا نبود.

کلاس‌های استاتیک

کلاس‌های خاصی وجود دارند که هرگز نمونه سازی نمی‌شوند مثل **Math**، **Console**، **Environment** و **ThreadPool**. این کلاس‌ها، تنها دارای اعضای استاتیک هستند و در واقع این کلاس‌ها تنها برای گروه بندی مجموعه‌ای از اعضای مشابه وجود دارند. برای نمونه، کلاس **Math** تعدادی متد برای انجام عملیات‌های ریاضی تعریف می‌کند. سی‌شارپ به شما اجازه می‌دهد که کلاس غیر قابل نمونه سازی را با کلمه کلیدی **static** سی‌شارپ تعریف کنید. این کلمه کلیدی تنها می‌تواند بر کلاس‌ها، نه ساختارها (نوع‌های مقداری) اعمال شود. چون CLR همواره اجازه می‌دهد نوع‌های مقداری نمونه سازی شوند و راهی برای توقف یا مانع شدن از آن وجود ندارد.

کامپایلر محدودیت‌های زیادی برای یک کلاس استاتیک تعیین می‌کند:

- کلاس باید مستقیما از **System.Object** مشتق شود، چون اشتقاق از هر کلاس پایه‌ی دیگر معنی ندارد چرا که وراثت تنها بر اشیاء اعمال می‌شود و شما نمی‌توانید یک نمونه از یک کلاس استاتیک بسازید.
- کلاس نباید هیچ رابطی را پیاده‌سازی کند چون متدهای رابط تنها توسط یک نمونه از یک کلاس قابل فراخوانی هستند.
- کلاس فقط باید اعضای استاتیک (فیلدها، متدها، ویژگی‌ها و رویدادها) را تعریف کند. وجود هر عضو نمونه، منجر به تولید خطا توسط کامپایلر می‌شود.
- کلاس نمی‌تواند به عنوان یک فیلد، پارامتر متد یا متغیر محلی استفاده شود چون تمام این‌ها یک متغیر را که به یک نمونه اشاره دارد تعیین می‌کنند و این مجاز نیست. اگر کامپایلر هر یک از این استفاده‌ها را ببیند یک خطا اعلام می‌کند.

نمونه‌ای از یک کلاس استاتیک که تعدادی عضو **static** تعریف می‌کند در این جا آمده است، این کد (با یک هشدار) کامپایل می‌شود اما کلاس کار خاصی انجام نمی‌دهد:

```
using System;

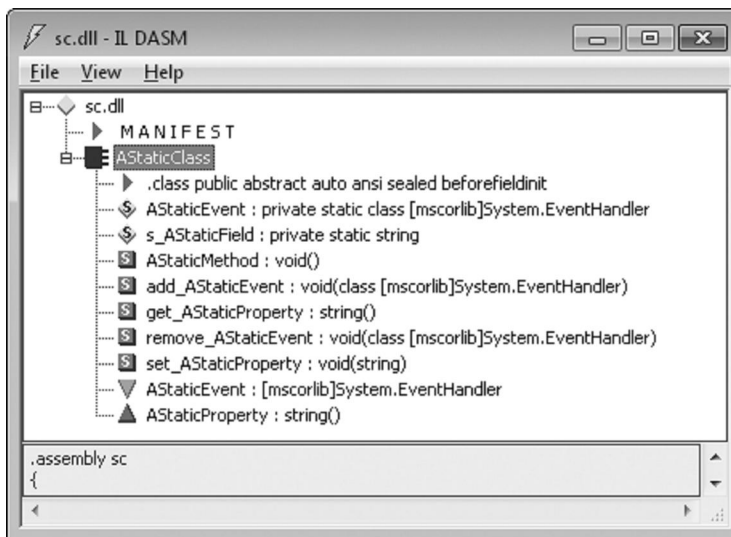
public static class AStaticClass {
public static void AStaticMethod() { }

    public static String AStaticProperty {
        get { return s_AStaticField; }
        set { s_AStaticField = value; }
    }

    private static String s_AStaticField;

    public static event EventHandler AStaticEvent;
}
```

اگر شما کد فوق را به یک اسمبلی کتابخانه‌ای (DLL) کامپایل کنید و با ILDasm.exe به خروجی نگاه کنید، آنچه در شکل ۲-۶ آمده است را می‌بینید. همانگونه که در شکل ۲-۶ می‌بینید، تعریف یک کلاس با کلمه کلیدی **static** باعث می‌شود کامپایلر سی‌شارپ کلاس را هم **abstract** هم **sealed** کند. علاوه بر این، کامپایلر، یک متد سازنده‌ی نمونه را در نوع قرار نمی‌دهد. توجه کنید که هیچ متد سازنده نمونه (**ctor**) در شکل ۲-۶ دیده نمی‌شود.



شکل ۲-۶ ILDasm.exe در حال نمایش کلاس **abstract** و **sealed** در متادیتا

کلاس‌های جزئی، ساختارها و رابط‌ها

در این بخش، من کلاس‌های جزئی، ساختارها و رابط‌ها را بحث می‌کنم. باید گفته شود که این ویژگی کاملاً توسط کامپایلر سی‌شارپ ارائه می‌شود (بعضی از دیگر کامپایلرها هم این ویژگی را ارائه می‌کنند)؛ CLR چیزی درباره‌ی کلاس‌های جزئی، ساختارها و رابط‌ها نمی‌داند.

کلمه کلیدی **partial** به معنای جزئی، به کامپایلر سی‌شارپ می‌گوید سورس کد متعلق به تعریف یک کلاس، ساختار یا رابط می‌تواند میان یک یا بیشتر فایل پخش شود. سه دلیل اصلی برای پخش کردن سورس کدتان بین چند فایل وجود دارد:

- **کنترل سورس** تصور کنید تعریف یک نوع از سورس کد زیادی تشکیل شده است و یک برنامه‌نویس، سورس کد را برای اعمال تغییرات بررسی می‌کند. هیچ برنامه‌نویس دیگری قادر به تغییر نوع در همان زمان بدون ادغام تغییرات در آینده، نیست. استفاده از کلمه کلیدی **partial** به شما

اجازه می‌دهد که کد متعلق به یک نوع را در میان چندین فایل پخش کنید که هر کدام از این فایل‌ها می‌توانند به صورت جداگانه بررسی شوند تا چند برنامه‌نویس بتوانند نوع را در یک زمان تغییر دهند.

▪ **تقسیم یک کلاس یا ساختار به چند واحد منطقی** من گاهی اوقات درون یک فایل یک تک نوع می‌سازم که ویژگی‌های مختلفی را ارائه می‌کند و به عنوان یک راه حل کامل در نظر گرفته می‌شود. برای ساده کردن پیاده‌سازی، من گاهی همان نوع جزئی را چندین بار درون یک فایل سورس کد تعریف می‌کنم، سپس در هر بخش از نوع جزئی، یک ویژگی همراه با تمام فیلدها، متدها، ویژگی‌ها، رویدادهای آن و غیره را پیاده‌سازی می‌کنم - این به من اجازه می‌دهد که تمام اعضای که یک ویژگی را فراهم می‌کنند در قالب یک گروه ببینم که کد زدن را برایم آسانتر می‌کند. همچنین، می‌توانم براحتی یک بخش از نوع جزئی را کامنت کنم تا کل یک ویژگی را از کلاس حذف و آن را با پیاده‌سازی دیگری (از طریق یک بخش جدید از نوع جزئی) جایگزین کنم.

▪ **تولید کد** در ویژوال استودیو، وقتی یک پروژه Windows Forms یا Web Forms جدید می‌سازید، چند فایل سورس کد به عنوان بخشی از پروژه ساخته می‌شوند. این فایل‌های سورس کد قالب‌هایی دارند که به شما در آغاز ساخت این برنامه‌ها کمک می‌کنند. وقتی از طراح-های ویژوال استودیو استفاده می‌کنید و کنترل‌ها را درون فرم ویندوز یا فرم وب می‌اندازید، ویژوال استودیو، سورس کد را برای شما به صورت خودکار می‌نویسد و آن را در فایل‌های سورس کد قرار می‌دهد. این واقعا بازدهی شما را افزایش می‌دهد. در گذشته، کد تولید شده درون همان فایل سورس کدی که شما با آن کار می‌کردید، قرار می‌گرفت. مشکل این کار این بود که ممکن بود شما کد تولیدی توسط طراح را تصادفاً تغییر دهید و طراح دیگر نمی‌توانست به درستی کار کند. از ویژوال استودیو ۲۰۰۵ به بعد، وقتی یک فرم ویندوز، فرم وب، یا کنترل کاربری و غیره می‌سازید، ویژوال استودیو دو فایل سورس کد برای شما درست می‌کند: یکی برای کد شما و دیگری برای کد تولیدی توسط طراح. چون کد طراح در فایل مجزایی است، احتمال تغییر احتمالی آن توسط شما بسیار کمتر است.

کلمه کلیدی **partial** به نوع‌ها در تمام فایل‌ها اعمال می‌شود. وقتی فایل‌ها با هم کامپایل می‌شوند، کامپایلر کد را برای تولید یک نوع که درون فایل اسمبلی `.exe` یا `dll` (یا فایل ماژول `.netmodule`) است، با هم ترکیب می‌کند. همانطور که در ابتدای این بخش گفتیم، ویژگی نوع‌های جزئی کاملاً توسط کامپایلر سی‌شارپ پیاده‌سازی می‌شود و CLR اصلاً چیزی درباره‌ی نوع‌های جزئی نمی‌داند. این علت آن است که چرا فایل‌های سورس کد برای نوع، می‌بایست در یک زبان برنامه‌نویسی نوشته شوند و همگی تحت یک واحد، کامپایل شوند.

کامپوننت‌ها، چندریختی و نسخه بندی

برنامه‌نویسی شی گرا (OOP) برای سال‌ها وجود داشته است. وقتی برای اولین بار در اواخر دهه ۷۰ و اوایل دهه ۸۰ استفاده شد، برنامه‌ها بسیار کوچکتر بوده و تمام کد برای ساختن یک برنامه توسط یک شرکت نوشته می‌شد. مطمئناً سیستم عامل‌هایی بودند که برنامه‌ها از ویژگی‌های آن‌ها استفاده می‌کردند اما این سیستم عامل‌ها در قیاس با سیستم عامل‌های کنونی، ویژگی‌های به مراتب کمتری ارائه می‌کردند.

امروزه، نرم افزار بسیار پیچیده تر شده است و کاربران می‌خواهند برنامه‌ها رابط گرافیکی، منوها، ورودی موس، ورودی تبلت، خروجی پرینتر، امور شبکه و غیره را ارائه کنند. به همین دلایل، سیستم عامل‌ها و پلتفرم‌های برنامه‌نویسی در طی سالیان اخیر بسیار رشد کرده اند. علاوه بر این، برای برنامه‌نویسان، نوشتن تمام کدهای مورد نیاز یک برنامه برای کارکرد مطابق با خواسته‌های کاربر دیگر عملی نیست. امروزه، برنامه‌ها از کدهایی که توسط شرکت‌های مختلف تولید شده‌اند تشکیل می‌شوند. این کدها با شی گرای به همدیگر متصل می‌شوند.

برنامه‌نویسی کامپوننت (CSP)

در اینجا برخی از ویژگی‌های یک کامپوننت را می‌بینیم:

- یک کامپوننت (یک اسمبلی در دات‌نت) دارای احساس "منتشر شدن" است.
- یک کامپوننت یک هویت دارد (یک نام، نسخه، فرهنگ و کلید عمومی).
- یک کامپوننت برای همیشه هویت خود را نگه می‌دارد (کد درون یک اسمبلی هرگز به صورت ثابت به اسمبلی دیگری لینک نمی‌شوند. دات‌نت همیشه از لینک پویا استفاده می‌کند).
- یک کامپوننت به وضوح کامپوننت‌هایی که به آن‌ها وابسته است را تعیین می‌کند (جدول‌های ارجاع متادیتا).
- یک کامپوننت باید کلاس‌ها و اعضای خود را مستند سازی کند. سی‌شارپ این کار را با نوشتن مستندات XML درون سورس کد همراه با سویچ کامپایلر `/doc` فراهم می‌کند.
- یک کامپوننت باید اجازه‌های امنیتی مورد نیازش را تعیین کند. امنیت دسترسی به کد (CAS) در CLR این امکان را فراهم می‌کند.

▪ یک کامپوننت یک رابط (مدل شی) که برای هیچ عملیات خدماتی تغییر نمی‌کند را منتشر می‌کند. یک خدمات **servicing** نسخه‌ای جدید از یک کامپوننت است که قصد دارد با نسخه‌های قبلی کامپوننت سازگار باشد. نوعا، یک نسخه خدمات شامل رفع خطاها، پیچ‌های امنیتی و برخی ویژگی‌های اضافی کوچک است. اما یک خدمات نمی‌تواند نیاز به وابستگی‌های جدید یا اجازه‌های امنیتی اضافی داشته باشد.

همانطور که در لیست قبلی دیدید، بخش بزرگی از **CSP** مرتبط با نسخه‌بندی است. کامپوننت‌ها در طول زمان تغییر می‌کنند و در بازه‌های زمان مختلفی عرضه می‌شوند. نسخه‌بندی یک لایه جدید از پیچیدگی برای **CSP** معرفی می‌کند که با **OOP** که در آن همه‌ی کدها به عنوان یک واحد و توسط یک شرکت نوشته، تست و عرضه می‌شدند، وجود نداشت. در این بخش من می‌خواهم بر روی نسخه‌بندی کامپوننت تمرکز کنم.

در دات‌نت، یک شماره نسخه از چهار بخش تشکیل شده است: یک بخش بزرگ، یک بخش کوچک، یک بخش ساخت و یک بخش بازبینی. برای نمونه، یک اسمبلی با شماره نسخه ۱.۲.۳.۴ دارای یک بخش بزرگ ۱، یک بخش کوچک ۲، یک بخش ساخت ۳ و یک بخش بازبینی ۴ است. بخش‌های بزرگ/کوچک معمولا برای نمایش یک سری ویژگی‌های ثابت و پایدار یک اسمبلی و بخش‌های ساخت/بازبینی برای نمایش یک سری ویژگی‌های خدماتی یک اسمبلی استفاده می‌شوند.

فرض کنید یک شرکت یک اسمبلی با نسخه ۲.۷.۰.۰ را عرضه می‌کند. اگر شرکت، در آینده خواست یک خطا در کامپوننت را برطرف کند، آن‌ها یک اسمبلی جدید که تنها بخش‌های ساخت/بازبینی نسخه تغییر کرده باشد، چیزی شبیه ۲.۷.۱.۳۴ را ارائه می‌کنند. این یعنی اسمبلی یک خدمات است که قصدش سازگاری با نسخه‌ی اصلی کامپوننت (نسخه ۲.۷.۰.۰) است.

در سوی دیگر، اگر شرکت خواست اسمبلی جدیدی بسازد که تغییرات مهمی کرده و دیگر نمی‌خواهد با نسخه اورجینال اسمبلی سازگار باشد، شرکت یک کامپوننت جدید و یک اسمبلی جدید می‌سازد و باید شماره نسخه‌ای به آن بدهد که بخش‌های بزرگ/کوچک متفاوت با کامپوننت کنونی باشد (برای نمونه نسخه ۳.۰.۰.۰).

نکته نحوه تفکر درباره‌ی شماره نسخه‌ها را برایتان گفتم. متاسفانه، **CLR** با شماره نسخه‌ها بدین گونه رفتار نمی‌کند. اکنون، **CLR** با شماره نسخه مثل یک مقدار مبهم رفتار می‌کند و اگر یک اسمبلی بر نسخه‌ی ۱.۲.۳.۴ از اسمبلی دیگری وابسته باشد، **CLR** فقط سعی می‌کند نسخه ۱.۲.۳.۴ را بارگذاری کند (مگر آنکه یک راهنما برای عمل اتصال در محل موجود باشد).

حال که دیدیم چگونه از شماره‌های نسخه برای آپدیت هویت یک کامپوننت به منظور نمایش یک نسخه جدید استفاده کنیم، بگذارید نگاهی به ویژگی‌های ارائه شده توسط **CLR** و زبان‌های برنامه‌نویسی (مثل سی‌شارپ) بیاندازیم که به برنامه‌نویسان اجازه می‌دهند تا کدی بنویسند که نسبت به تغییرات احتمالی در کامپوننت‌های مورد استفاده، انعطاف پذیر باشند.

مشکلات نسخه‌بندی زمانی مطرح می‌شود که یک نوع تعریف شده در یک کامپوننت (اسمبلی) به عنوان کلاس پایه برای یک نوع در کامپوننت (اسمبلی) دیگری استفاده می‌شود. واضح است اگر نسخه‌ی کلاس پایه عوض شود، رفتار نوع مشتق شده نیز تغییر می‌کند و احتمالا باعث می‌شود کلاس غیر صحیح عمل کند. این بخصوص پیرامون سناریوهای چند ریختی که یک نوع مشتق شده، متدهای مجازی تعریف شده توسط کلاس پایه را بازنویسی می‌کند، صحیح است. سی‌شارپ پنج کلمه کلیدی ارائه می‌کند که شما می‌توانید بر نوع‌ها و/یا اعضای نوع که بر نسخه‌بندی کامپوننت موثرند، اعمال کنید. این ویژگی‌ها دقیقا بر ویژگی‌های نسخه‌بندی کامپوننت که **CLR** ارائه می‌کند، منطبق هستند. جدول ۲-۶ کلمات کلیدی سی‌شارپ مرتبط با نسخه‌بندی کامپوننت را داراست و بیان می‌کند هر کلمه کلیدی بر تعریف یک نوع یا عضو نوع، چگونه اثر می‌گذارد:

جدول ۲-۶ کلمات کلیدی سی‌شارپ و اثر آن‌ها بر نسخه‌بندی کامپوننت

کلمه کلیدی سی‌شارپ	نوع	متد/ویژگی/ارویداد	ثابت/فیلد
abstract	بیان می‌کند که هیچ نمونه‌ای از نوع نمی‌توان ساخت	بیان می‌کند که نوع مشتق شده، پیش از آنکه نمونه-ای از نوع مشتق شده ساخته شود باید این عضو را بازنویسی و پیاده‌سازی کند.	(غیر مجاز)
virtual	(غیر مجاز)	بیان می‌کند که این عضو می‌تواند توسط نوع مشتق شده بازنویسی شود.	(غیر مجاز)
override	(غیر مجاز)	بیان می‌کند که نوع مشتق شده، در حال بازنویسی عضو نوع پایه است.	(غیر مجاز)
sealed	بیان می‌کند که نوع نمی‌تواند به عنوان	بیان می‌کند که عضو نمی‌تواند توسط یک نوع مشتق	(غیر مجاز)

یک نوع پایه استفاده شود. شده بازنویسی شود. این کلمه کلیدی فقط بر متدی قابل اعمال است که در حال بازنویسی یک متد مجازی است.

new وقتی به یک نوع تودرتو، متد، ویژگی، رویداد، ثابت یا فیلد اعمال می‌شود بیان می‌کند که عضو، هیچ رابطه‌ای با عضو مشابه که ممکن است در کلاس پایه باشد، ندارد.

من در بخش آتی با عنوان "کار با متدهای مجازی هنگام نسخه‌بندی نوع‌ها" تمام این کلمات کلیدی و استفاده‌ی آن‌ها را توضیح می‌دهم، اما پیش از آنکه به سناریوی نسخه‌بندی بپردازیم بگذارید بر نحوه‌ی فراخوانی متدهای مجازی توسط CLR نگاه کنیم.

چگونه CLR متدها، ویژگی‌ها و رویدادهای مجازی را فراخوانی می‌کند

در این بخش، من بر متدها تمرکز می‌کنم اما این بحث به ویژگی‌های مجازی و رویدادهای مجازی هم ربط دارد. ویژگی‌ها و رویدادها در واقع به عنوان متد پیاده‌سازی می‌شوند؛ این موضوع در فصل‌های مربوط به خودشان گفته خواهد شد.

متدها کدی را نشان می‌دهند که عملیاتی بر روی یک نوع (متدهای استاتیک) یا بر روی نمونه‌ای از یک نوع (متدهای غیر استاتیک) انجام می‌دهد. همه‌ی متدها یک نام، یک امضاء و یک مقدار برگشتی (که ممکن است **void** باشد) دارند. CLR اجازه می‌دهد یک نوع، چندین متد با نام یکسان تا زمانی که هر متد مجموعه‌ای متفاوت از پارامترها یا مقدار برگشتی متفاوت دارند را تعریف کند. پس این ممکن است که دو متد با نام یکسان و پارامترهای یکسان تا زمانی که متدها نوع برگشتی متفاوتی دارند، تعریف کنید. هرچند، به جز زبان اسمبلی IL، من از زبان دیگری که از این "ویژگی" استفاده کند، مطلع نیستم؛ اغلب زبان‌ها (شامل سی‌شارپ) نیاز دارند که متدها از لحاظ پارامترها متفاوت باشند و هنگام تشخیص منحصر بفرود بودن، نوع برگشتی متد را نادیده می‌گیرند. (سی‌شارپ در واقع هنگام تعریف متدهای عملگر تبدیل، این ویژگی را آزاد می‌کند؛ به فصل ۸ مراجعه کنید).

کلاس **Employee** که در زیر آمده است، سه گونه متفاوت از متدها را تعریف می‌کند:

```
internal class Employee {
    // A nonvirtual instance method
    public Int32 GetYearsEmployed() { ... }

    // A virtual method (virtual implies instance)
    public virtual String GetProgressReport() { ... }

    // A static method
    public static Employee Lookup(String name) { ... }
}
```

هنگام کامپایل این کد، کامپایلر، سه ورودی در جدول تعریف متد اسمبلی تولیدی، قرار می‌دهد. هر ورودی دارای پرچم‌های تنظیم شده است که بیان می‌کند متد، نمونه یا مجازی یا استاتیک است.

وقتی کدی برای فراخوانی هر یک از این متدها نوشته می‌شود، کامپایلر، پرچم‌های تعریف متد را بررسی می‌کند تا معلوم شود چگونه کد IL صحیح را تولید کند تا فراخوانی به درستی صورت پذیرد. CLR دو دستور IL برای فراخوانی یک متد ارائه می‌کند:

- دستور **IL call**، می‌تواند برای فراخوانی متدهای استاتیک، نمونه و مجازی استفاده شود. وقتی دستور **call** برای فراخوانی یک متد استاتیک استفاده می‌شود، شما باید نوعی را تعیین کنید که متدی که CLR باید فراخوانی کند را تعریف می‌کند. وقتی **call** برای فراخوانی یک متد نمونه یا مجازی استفاده می‌شود، شما باید متغیری که به یک شی اشاره دارد را تعیین کنید. دستور **call** فرض می‌کند که این متغیر، **null** نیست. به بیان دیگر، خود نوع متغیر، بیان می‌کند متدی که باید CLR فراخوانی کند را کدام نوع تعریف کرده است. اگر نوع متغیر، متد را تعریف نکرده باشد، نوع‌های پایه برای یافتن متد بررسی می‌شوند. دستور **call** اغلب برای فراخوانی غیرمجازی یک متد مجازی استفاده می‌شود.
- دستور **IL callvirt**، می‌تواند برای فراخوانی متدهای نمونه و مجازی استفاده شود ولی برای متدهای استاتیک استفاده نمی‌شود. وقتی دستور **callvirt** برای فراخوانی یک متد نمونه یا مجازی استفاده می‌شود، شما باید یک متغیر که به یک شی اشاره دارد را تعیین کنید. وقتی دستور **callvirt** برای فراخوانی یک متد غیرمجازی نمونه استفاده می‌شود، نوع متغیر تعیین می‌کند کدام نوع، متدی که CLR باید فراخوانی کند را

تعریف کرده است. وقتی دستور **callvirt** برای فراخوانی یک متد مجازی نمونه استفاده می‌شود، CLR نوع واقعی شی که فراخوانی کرده است را کشف می‌کند و سپس متد را به صورت چندریختی فراخوانی می‌کند. برای تعیین نوع، متغیری که فراخوانی را انجام داده است نباید **null** باشد. به بیان دیگر، هنگام کامپایل این فراخوانی، کامپایلر JIT کدی تولید می‌کند که بررسی می‌کند مقدار متغیر **null** نباشد. اگر آن **null** است، دستور **callvirt** باعث می‌شود که CLR یک **NullReferenceException** تولید کند. این بررسی اضافی به این معنی است که دستور **IL**، **callvirt**، کمی آهسته تر از دستور **call** عمل می‌کند. توجه کنید که این بررسی **null** نبودن حتی وقتی دستور **callvirt** برای فراخوانی یک متد غیرمجازی نمونه استفاده می‌شود نیز انجام می‌پذیرد.

حال، همه این‌ها را کنار هم بگذاریم و ببینیم چگونه سی‌شارپ از دستورات مختلف **IL** استفاده می‌کند.

```
using System;
```

```
public sealed class Program {
    public static void Main() {
        Console.WriteLine(); // Call a static method

        Object o = new Object();
        o.GetHashCode(); // Call a virtual instance method
        o.GetType(); // Call a nonvirtual instance method
    }
}
```

اگر شما کد فوق را کامپایل کرده و به **IL** تولید شده نگاه کنید، محتویات زیر را می‌بینید:

```
.method public hidebysig static void Main() cil managed {
    .entrypoint
    // Code size 26 (0x1a)
    .maxstack 1
    .locals init (object v_0)
    IL_0000: call void System.Console::WriteLine()
    IL_0005: newobj instance void System.Object::.ctor()
    IL_000a: stloc.0
    IL_000b: ldloc.0
    IL_000c: callvirt instance int32 System.Object::GetHashCode()
    IL_0011: pop
    IL_0012: ldloc.0
    IL_0013: callvirt instance class System.Type System.Object::GetType()
    IL_0018: pop
    IL_0019: ret
} // end of method Program::Main
```

توجه کنید که کامپایلر سی‌شارپ از دستور **IL**، **call**، برای فراخوانی متد **WriteLine** از **Console** استفاده می‌کند. این، مورد انتظار است چون **WriteLine** یک متد استاتیک است. بعد از آن، دقت کنید که دستور **IL**، **callvirt** برای فراخوانی **GetHashCode** استفاده شده است. این هم طبق انتظار است چون **GetHashCode** یک متد مجازی است. سرانجام، کامپایلر سی‌شارپ باز هم از دستور **IL**، **callvirt** برای فراخوانی متد **GetType** استفاده نموده است. این تعجب‌آور است، چون **GetType** یک متد مجازی نیست. اما این کد کار می‌کند چون وقتی کد، JIT کامپایل می‌شود، CLR می‌داند که **GetType** یک متد مجازی نیست و کد JIT شده، **GetType** را به صورت غیرمجازی فراخوانی می‌کند.

البته، سوال اینست که چرا کامپایلر سی‌شارپ از دستور **call** به جای آن استفاده نکرد؟ جواب اینست که تیم سی‌شارپ تصمیم گرفتند که کامپایلر JIT کدی تولید کند که بررسی نماید متغیر فراخواننده، **null** نباشد. این یعنی فراخوانی متدهای غیرمجازی نمونه آهسته تر از آنچه باید باشند، هستند. این همچنین یعنی، کد سی‌شارپ زیر منجر به تولید **NullReferenceException** می‌شود. در برخی زبان‌های برنامه‌نویسی دیگر، کد زیر به خوبی اجرا می‌شود.

```
using System;
```

```
public sealed class Program {
    public Int32 GetFive() { return 5; }
    public static void Main() {
        Program p = null;
        Int32 x = p.GetFive(); // In C#, NullReferenceException is thrown
    }
}
```

از نظر تئوری، کد فوق صحیح است. مطمئناً متغیر **p**، **null** است، اما هنگام فراخوانی یک متد غیرمجازی (**GetFive**)، CLR نیاز دارد که تنها نوع داده‌ای **p** را بداند، که **Program** است. اگر **GetFive** فراخوانی شود، مقدار آرگومان **this**، **null** خواهد بود. چون آرگومان، درون متد **GetFive** استفاده نمی‌شود، هیچ **NullReferenceException** تولید نخواهد شد. اما چون کامپایلر سی‌شارپ به جای یک دستور **call**، یک دستور **callvirt** تولید می‌کند، کد فوق منجر به تولید **NullReferenceException** خواهد شد.

مهم اگر شما یک متد را غیرمجازی تعریف کنید، هرگز نباید متد را در آینده به مجازی تغییر دهید. علت آن است که برخی کامپایلرها متد غیر مجازی را با دستور **call** به جای دستور **callvirt** فراخوانی می‌کنند. اگر متد از غیرمجازی به مجازی تغییر کند و کد ارجاع کننده مجدداً کامپایل نشود، متد مجازی به صورت غیرمجازی فراخوانی می‌شود که باعث می‌شود رفتاری غیر منتظره داشته باشد. اگر ارجاع کننده در سی‌شارپ نوشته شده باشد، مشکلی نیست، چون سی‌شارپ تمام متدهای نمونه را با دستور **callvirt** فراخوانی می‌کند. اما اگر کد ارجاع کننده با زبان برنامه‌نویسی دیگری نوشته شده باشد، این می‌تواند منجر به بروز مشکل شود.

گاهی، کامپایلر برای فراخوانی یک متد مجازی به جای استفاده از دستور **callvirt** از دستور **call** استفاده می‌کند. در ابتدا تعجب برانگیز است، اما کد زیر نشان می‌دهد چرا گاهی این کار نیاز است:

```
internal class SomeClass {
    // ToString is a virtual method defined in the base class: Object.
    public override String ToString() {

        // Compiler uses the 'call' IL instruction to call
        // Object's ToString method nonvirtually.

        // If the compiler were to use 'callvirt' instead of 'call', this
        // method would call itself recursively until the stack overflowed.
        return base.ToString();
    }
}
```

هنگام فراخوانی **base.ToString** (یک متد مجازی)، کامپایلر سی‌شارپ یک دستور **call** تولید می‌کند تا مطمئن شود که متد **ToString** در نوع پایه به صورت غیرمجازی فراخوانی می‌شود. این نیاز است چون اگر **ToString**، مجازی فراخوانی شود، فراخوانی تا سرریز شدن پشته‌ی ترد به صورت بازگشتی ادامه می‌یابد که مطلوب نیست.

کامپایلرها مایلند که هنگام فراخوانی متدهای تعریف شده توسط یک نوع مقداری، از دستور **call** استفاده کنند، چون نوع‌های مقداری مهر شده‌اند (**sealed**). این بیان می‌کند که حتی برای متدهای مجازی آن‌ها هم چند ریختی وجود ندارد، که باعث می‌شود فراخوانی سریعتر انجام پذیرد. به علاوه، ماهیت یک نمونه‌ی نوع مقداری تضمین می‌کند که هرگز **null** نیست، پس **NullReferenceException** هرگز تولید نمی‌شود. سرانجام، اگر بخواهید متد مجازی از یک نوع مقداری را به صورت مجازی فراخوانی کنید، CLR نیاز به داشتن یک ارجاع به شی نوع از نوع مقداری دارد تا به جدول متد درون آن اشاره کند. این نیازمند بسته‌بندی نوع مقداری است. بسته‌بندی، فشار بیشتری بر هیپ اعمال می‌کند و منجر به عملیات جمع‌آوری بیشتر و کاهش سرعت می‌شود.

بدون توجه به اینکه **call** یا **callvirt** برای فراخوانی یک متد نمونه یا مجازی استفاده شود، این متدها همیشه یک آرگومان مخفی **this** به عنوان اولین پارامتر متد، دریافت می‌کنند. آرگومان **this** به شی‌ای که بر روی آن عمل می‌شود، اشاره دارد.

هنگام طراحی یک نوع، شما باید سعی کنید تعداد متدهای مجازی که تعریف می‌کنید را حداقل کنید. اول اینکه، فراخوانی یک متد مجازی، کندتر از فراخوانی یک متد غیرمجازی است. دوم اینکه، متدهای مجازی نمی‌توانند توسط کامپایلر JIT خطی (inline) شوند که باز هم سرعت را کاهش می‌دهد. سوم اینکه، متدهای مجازی نسخه‌بندی کامپوننت‌ها را همانگونه که در بخش بعدی گفته می‌شود، شکننده تر می‌کنند. چهارم اینکه، هنگام تعریف یک نوع پایه، رایج است که تعدادی متدهای سربارگذاری شده ارائه شود. اگر بخواهید این متدها چندریختی باشند، بهترین کار این است که پیچیده ترین آن‌ها را مجازی کنید و تمام دیگر متدهای سربارگذاری شده را غیرمجازی باقی بگذارید. پیروی از این قاعده، قابلیت نسخه‌بندی یک کامپوننت را بدون اثر بر نوع‌های مشتق شده، افزایش می‌دهد. نمونه‌ای در اینجا آمده است:

```
public class Set {
    private Int32 m_length = 0;

    // This convenience overload is not virtual
    public Int32 Find(Object value) {
        return Find(value, 0, m_length);
    }

    // This convenience overload is not virtual
    public Int32 Find(Object value, Int32 startIndex) {
        return Find(value, startIndex, m_length - startIndex);
    }

    // The most feature-rich method is virtual and can be overridden
    public virtual Int32 Find(Object value, Int32 startIndex, Int32 endIndex) {
        // Actual implementation that can be overridden goes here...
    }

    // Other methods go here
}
```

استفاده هوشمندانه از پدیداری نوع و دسترس پذیری نوع

با دات‌نت فریم‌ورک، برنامه‌ها از نوع‌های تعریف شده در چندین اسمبلی که توسط شرکت‌های مختلف تولید شده اند، تشکیل می‌گردند. این یعنی، برنامه‌نویس کمترین کنترل را بر کامپوننتی دارد که از نوع‌های تعریف شده در آن استفاده می‌کند. برنامه‌نویس، نوعا به سورس کد دسترسی ندارد (و احتمالا نمی‌داند چه زبان برنامه‌نویسی برای ساخت کامپوننت استفاده شده است) و کامپوننت‌ها اغلب در زمان‌های مختلف نسخه‌بندی می‌شوند. علاوه بر این، به دلیل چندریختی و اعضای محافظت شده، برنامه‌نویس یک کلاس پایه به کد نوشته شده توسط برنامه‌نویس کلاس مشتق شده، باید اعتماد کند. و البته برنامه‌نویس یک کلاس مشتق شده باید به کدی که از کلاس پایه به ارث می‌برد اعتماد کند. این‌ها تنها تعدادی از مسائلی بود که شما هنگام طراحی کامپوننت‌ها و نوع‌ها باید به آن‌ها فکر کنید.

در این بخش، می‌خواهم فقط چند کلمه پیرامون چگونگی طراحی یک نوع با لحاظ این مسائل بگویم. به خصوص می‌خواهم بر نحوه‌ی صحیح تنظیم پدیداری نوع و دسترس پذیری عضو تمرکز کنم تا شما در این زمینه موفق باشید.

اول اینکه، هنگام تعریف یک نوع جدید، کامپایلرها باید کلاس را به صورت پیش فرض مهر کنند تا کلاس بتواند به عنوان یک کلاس پایه استفاده شود. در عوض، بسیاری از کامپایلرها، شامل سی‌شارپ، کلاس‌ها را مهر نمی‌کنند و به برنامه‌نویس اجازه می‌دهند که صریحا با کلمه کلیدی **sealed** یک کلاس را مهر کند. می‌دانم که الان خیلی دیر است، اما فکر می‌کنم کامپایلرهای امروزی، پیش فرض اشتباهی انتخاب کرده‌اند و خیلی خوب است اگر در کامپایلرهای آینده این امر اعمال شود. سه دلیل برای آنکه چرا یک کلاس مهر شده بهتر از یک کلاس مهر نشده است وجود دارد:

- **نسخه‌بندی** وقتی یک کلاس از اساس مهر شده باشد، آن می‌تواند در آینده بدون از بین بردن سازگاری، بدون مهر شود. اما، وقتی یک کلاس بدون مهر است، شما هرگز در آینده نمی‌توانید بدون از کار انداختن نوع‌های مشتق شده، آن را مهر کنید. به علاوه، اگر کلاس بدون مهر، هر متد مجازی بدون مهری تعریف کند، ترتیب فراخوانی متدهای مجازی باید حفظ شود وگرنه احتمال از کار افتادن نوع‌های مشتق شده در آینده وجود دارد.

- **عملکرد** همانگونه که در بخش قبلی گفته شد، فراخوانی یک متد مجازی به خوبی فراخوانی یک متد غیرمجازی عمل نمی‌کند چون CLR باید نوع شی را در زمان اجرا بیابد تا تعیین کند کدام نوع، متد فراخوانی شده را تعریف کرده است. اما، اگر کامپایلر JIT یک فراخوانی به یک متد مجازی توسط یک نوع مهر شده را ببیند، کامپایلر JIT می‌تواند کدی کارا تر با فراخوانی متد به صورت غیرمجازی تولید کند. این کار را به این دلیل می‌تواند انجام دهد که اگر کلاس مهر شده باشد، یک کلاس مشتق شده وجود ندارد. برای نمونه، در کد زیر، کامپایلر JIT می‌تواند متد ToString را به صورت غیرمجازی فراخوانی کند:

```
using System;
public sealed class Point {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }

    public static void Main() {
        Point p = new Point(3, 4);

        // The C# compiler emits the callvirt instruction here but the
        // JIT compiler will optimize this call and produce code that
        // calls ToString nonvirtually because p's type is Point,
        // which is a sealed class
        Console.WriteLine(p.ToString());
    }
}
```

- **امنیت و قابلیت پیش‌بینی** یک کلاس باید از وضعیتش محافظت کرده و اجازه ندهد که خودش حتی خراب شود. وقتی یک کلاس مهر نشده باشد، در صورتیکه فیلدها یا متدهایی که در درون، فیلدها را دستکاری می‌کنند در دسترس بوده و خصوصی نباشند، یک کلاس مشتق شده می‌تواند به وضعیت کلاس پایه دسترسی داشته و آن را دستکاری کند. به علاوه، یک متد مجازی می‌تواند توسط یک کلاس مشتق شده، بازنویسی شود و کلاس مشتق شده می‌تواند تصمیم بگیرد پیاده‌سازی کلاس پایه از متد را فراخوانی کند یا خیر. با مجازی کردن یک متد، ویژگی یا رویداد، کلاس پایه بخشی از کنترل بر رفتار و وضعیتش را در اختیار دیگران قرار می‌دهد. اگر با دقت بدان توجه نشود، این می‌تواند باعث شود که شی غیر منتظره رفتار کرده و حفره‌های امنیتی احتمالی را ایجاد کند.

مشکل یک کلاس مهر شده این است که برای کاربران نوع، اسباب مزاحمت است. گهگاه، برنامه‌نویسان می‌خواهند کلاسی مشتق شده از یک نوع موجود بسازند تا فیلدها یا اطلاعاتی برای استفاده برنامه خودشان را به آن اضافه کنند. در واقع، شاید بخواهند متدهای کمکی برای تغییر و دستکاری این فیلدها نیز تعریف کنند. چون کلاس مهر شده جلوی این قابلیت را می‌گیرد، من یک پیشنهاد به تیم CLR دادم که آن‌ها یک تغییر دهنده‌ی جدید برای کلاس، به نام **closed** تعریف کنند.

یک کلاس بسته می‌تواند به عنوان کلاس پایه استفاده شود اما رفتارش بسته بوده و توسط کلاس مشتق شده قابل تغییر نیست. اساساً، یک کلاس پایه بسته، مانع از دسترسی یک کلاس مشتق شده به هر عضو غیر عمومی از کلاس پایه می‌شود. این اجازه می‌دهد که کلاس پایه تغییر کند با دانستن این نکته که بر کلاس مشتق شده اثری ندارد. به صورت ایده‌آل، کامپایلرها تغییر دهنده دسترسی پیش فرض را برای نوع‌ها به **closed** تغییر می‌دهند چون این امن‌ترین انتخاب است بدون آنکه محدودیت زیادی ایجاد کند. اکنون خیلی زود است که بدانیم آیا این ایده راهش را به CLR و زبان‌های برنامه‌نویسی پیدا می‌کند یا نه. به هر حال، من خیلی امیدوارم که روزی این اتفاق بیافتد.

ضمناً، شما همین الان می‌توانید آنچه **closed** برای آن طراحی شده است را پیاده‌سازی کنید، فقط خیلی ناچور است. اساساً، وقتی شما کلاستان را پیاده‌سازی می‌کنید، مطمئن شوید تمام متدهای مجازی که به ارث می‌برید را مهر کنید (شامل متدهای تعریف شده توسط **System.Object**). همچنین، هیچ متدی که در آینده مشکل نسخه‌بندی ایجاد کند، مثل متدهای مجازی یا محافظت شده را تعریف نکنید. نمونه‌ای را در زیر می‌بینید:


```

public class SimulatedClosedClass : Object {
    public sealed override Boolean Equals(Object obj) {
        return base.Equals(obj);
    }
    public sealed override Int32 GetHashCode() {
        return base.GetHashCode();
    }
    public sealed override String ToString() {
        return base.ToString();
    }
    // Unfortunately, C# won't let you seal the Finalize method

    // Define additional public or private members here...
    // Do not define any protected or virtual members
}

```

متأسفانه، کامپایلرها و CLR، هم اکنون از نوع‌های بسته پشتیبانی نمی‌کنند. در اینجا راهنمایی‌هایی که هنگام تعریف کلاس‌های خودم پیروی می‌کنم را می‌گویم.

- هنگام تعریف یک کلاس، من همیشه صریحا آن را **sealed** می‌کنم مگر آنکه قصد داشته باشم که کلاس، یک کلاس پایه باشد که اجازه خصوصی سازی را به کلاس‌های مشتق شده بدهد. همانگونه که قبلا گفتم، این برخلاف چیزی است که سی‌شارپ و بسیاری دیگر کامپایلرها اکنون به عنوان پیش فرض انتخاب می‌کنند. من همچنین پیش فرض را بر **internal** کردن کلاس می‌گذارم مگر آنکه بخواهم کلاس خارج از اسمبلی من در معرض دید عمومی باشد. خوشبختانه، اگر شما صریحا پدیداری یک نوع را تعیین نکنید، کامپایلر سی‌شارپ پیش فرض را بر **internal** می‌گذارد. من اگر واقعا حس کنم مهم است که کلاسی تعریف کنم که دیگران بتوانند از آن مشتق شوند اما اجازه خصوصی سازی را به آن‌ها ندهم، من ساخت یک کلاس بسته را با تکنیک مهر کردن متدهای مجازی که کلاسم به ارث می‌برد را همانند مثال قبل، شبیه سازی می‌کنم.
- درون کلاس، همواره فیلدهای داده را **private** تعریف می‌کنم و هرگز تردید نخواهم کرد. خوشبختانه، سی‌شارپ پیش فرض را بر **private** کردن فیلدها می‌گذارد. من حقیقتا ترجیح می‌دهم که سی‌شارپ حکم می‌کرد تمام فیلدها باید خصوصی باشند و شما نمی‌توانستید فیلدها را **public**، **internal**، **protected** و غیره بکنید. در معرض قرار دادن وضعیت نوع، ساده ترین راه افتادن در مشکلات است، شی شما به صورت غیر قابل پیش‌بینی رفتار کرده و حفره‌های امنیتی احتمالی باز می‌شوند. حتی اگر بعضی فیلدها را هم **internal** تعریف کنم، باز این مطلب درست است. حتی درون یک تک اسمبلی، خیلی سخت است که تمام کدهایی که به یک فیلد ارجاع دارند را ردیابی کنید، به ویژه اگر چندین برنامه‌نویس کد می‌نویسند و آن را در یک اسمبلی کامپایل می‌کنند.
- درون کلاس، من همیشه متدها، ویژگی‌ها و رویدادها را **private** و غیرمجازی تعریف می‌کنم. خوشبختانه، سی‌شارپ پیش فرض را بر همین می‌گذارد. مطمئنا، من یک متد، ویژگی یا رویداد را **public** می‌کنم تا کاربردی از نوع را در معرض عموم قرار دهم. من سعی می‌کنم از **protected** یا **internal** کردن اعضا خودداری کنم چون این کار، نوع من را نفوذ پذیر خواهد کرد. اما به حال، به جای **virtual** کردن یک عضو آن را **protected** یا **internal** می‌کنم، چون یک عضو مجازی کنترل بسیاری را در اختیار دیگران قرار می‌دهد و واقعا به رفتار درست کلاس مشتق شده بستگی دارد.
- یک مثل قدیمی OOP وجود دارد که این گونه است: وقتی چیزها خیلی پیچیده می‌شوند، نوع‌های بیشتری بساز. وقتی پیاده‌سازی یک الگوریتم می‌خواهد پیچیده شود، من نوع‌های کمکی تعریف می‌کنم که بخش‌های مجزای کاربردی را در خود داشته باشند. اگر من این کلاس‌های کمکی را برای استفاده توسط یک کلاس اصلی تعریف می‌کنم، کلاس‌های کمکی را درون کلاس اصلی به صورت تودرتو تعریف خواهم کرد. این امر به کد درون نوع کمکی اجازه می‌دهد که اعضای خصوصی تعریف شده در نوع اصلی را ارجاع کند. اما، یک قاعده راهنمای طراحی وجود دارد که توسط ابزار تحلیل کد Code Analysis (FxCopCmd.exe) در ویژوال استودیو، بر آن تاکید می‌شود. این قاعده بیان می‌کند که نوع‌های تودرتوی عمومی باید در میدان فایل یا اسمبلی و نه درون نوع دیگر تعریف شوند. این قاعده وجود دارد چون بعضی برنامه‌نویسان، نحو ارجاع به نوع‌های تودرتو را مایه‌ی زحمت می‌دانند. من این قاعده را ارزش می‌نهم و هرگر نوع‌های تودرتوی عمومی تعریف نمی‌کنم.

کار با متدهای مجازی هنگام نسخه بندی نوع ها

همانطور که قبلا گفتیم، در یک محیط برنامه نویسی کامپوننت، نسخه بندی یک مسئله بسیار مهم است. من پیرامون برخی از این مسائل در فصل ۳ "اسمبلی های اشتراکی و اسمبلی های قوی نام" صحبت کردم جاییکه درباره ی اسمبلی های قوی نام توضیح دادم و بحث کردم چگونه یک مدیر می تواند مطمئن شود یک برنامه به همان اسمبلی هایی متصل می شود که با آن ها تست و ساخته شده است. اما، مسائل نسخه بندی دیگری باعث می شوند مسائل عدم سازگاری در کد پیش آید. برای نمونه، شما باید هنگام اضافه کردن یا تغییر اعضای یک نوع که به عنوان یک نوع پایه استفاده شده است، بسیار دقت کنید. بگذارید به چند مثال نگاه کنیم:

CompanyA نوع Phone را طراحی کرده است:

```
namespace CompanyA {
    public class Phone {
        public void Dial() {
            Console.WriteLine("Phone.Dial");
            // Do work to dial the phone here.
        }
    }
}
```

حال تصور کنید که CompanyB نوع دیگری، **BetterPhone**، را طراحی می کند که از نوع **Phone** از CompanyA به عنوان نوع پایه استفاده می کند:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {
        public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Do work to establish the connection.
        }
    }
}
```

وقتی CompanyB سعی کند این کد را کامپایل کند، کامپایلر سی شارپ پیام زیر را نشان می دهد:

“warning CS0108: ‘CompanyB.BetterPhone.Dial()’ hides inherited member ‘CompanyA.Phone.Dial()’. Use the new keyword if hiding was intended.”

این هشدار، برنامه نویسی را مطلع می کند که **BetterPhone** یک متد **Dial** تعریف می کند که متد **Dial** تعریف شده توسط **Phone** را مخفی می کند. این متد جدید می تواند معنای **Dial** (آنگونه که CompanyA در اصل، متد **Dial** را ساخته است) را تغییر دهد.

این ویژگی بسیار جالبی از کامپایلر است که به شما درباره ی این عدم تطبیق معنایی احتمالی، هشدار می دهد. کامپایلر همچنین به شما می گوید چگونه این هشدار را با افزودن کلمه کلیدی **new** قبل از تعریف **Dial** در کلاس **BetterPhone** از بین ببرید. کلاس **BetterPhone** تصحیح شده را در اینجا می بینید:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // This Dial method has nothing to do with Phone's Dial method.
        public new void Dial() {
```

```

        Console.WriteLine("BetterPhone.Dial");
        EstablishConnection();
        base.Dial();
    }

    protected virtual void EstablishConnection() {
        Console.WriteLine("BetterPhone.EstablishConnection");
        // Do work to establish the connection.
    }
}
}

```

در این لحظه، **CompanyB** می‌تواند از **BetterPhone.Dial** در برنامه‌اش استفاده کند. نمونه کدی که **CompanyB** شاید بنویسد را در زیر می‌بینید:

```

public sealed class Program {
    public static void Main() {
        CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
        phone.Dial();
    }
}

```

وقتی این کد اجرا شود، خروجی زیر نمایش داده می‌شود:

```

BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial

```

این خروجی نشان می‌دهد که **CompanyB** رفتار مطلوبش را بدست آورده است. فراخوانی **Dial**، فراخوانی متد **Dial** ای است که توسط **BetterPhone** تعریف شده است که آن هم متد مجازی **EstablishConnection** و سپس متد **Dial** از کلاس پایه **Phone** را فراخوانی می‌کند. حال تصور کنید که چندین شرکت تصمیم به استفاده از نوع **Phone** از **CompanyA** گرفته‌اند. بعد از این، تصور کنید که این شرکت‌ها به این نتیجه رسیده‌اند که قابلیت ایجاد یک ارتباط در متد **Dial**، واقعا یک ویژگی مفید است. این فیدبک به **CompanyA**، منجر به تغییر کلاس **Phone** بدین گونه شده است:

```

namespace CompanyA {
    public class Phone {
        public void Dial() {
            Console.WriteLine("Phone.Dial");
            EstablishConnection();
            // Do work to dial the phone here.
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("Phone.EstablishConnection");
            // Do work to establish the connection.
        }
    }
}

```

حال وقتی **CompanyB** نوع **BetterPhone** خودش را کامپایل می‌کند (که از نسخه‌ی جدید **Phone** از **CompanyA** مشتق شده است)، کامپایلر این پیام را نشان می‌دهد:

“warning CS0114: ‘CompanyB.BetterPhone.EstablishConnection()’ hides inherited member ‘CompanyA.Phone.EstablishConnection()’. To make the current member override that implementation, add the override keyword. Otherwise, add the new keyword.”

کامپایلر شما را از اتفاقی مطلع می‌کند که هر دوی **Phone** و **BetterPhone** یک متد **EstablishConnection** ارائه می‌کنند که معنای این دو احتمالاً یکی نیست، فقط کامپایلر مجدد **BetterPhone** نمی‌تواند دیگر همان رفتاری را داشته باشد که هنگام استفاده از اولین نسخه از نوع **Phone**، داشته است.

اگر **CompanyB** به این نتیجه برسد که متدهای **EstablishConnection** از لحاظ معنایی در هر دو نوع یکسان نیستند، **CompanyB** می‌تواند به کامپایلر بگوید که متدهای **Dial** و **EstablishConnection** تعریف شده در **BetterPhone**، متدهای صحیح برای استفاده هستند و رابطه‌ای با متد **EstablishConnection** تعریف شده توسط نوع پایه **Phone** ندارد. **CompanyB** با افزودن کلمه کلیدی **new** به متد **EstablishConnection**، کامپایلر را این تصمیم خود آگاه می‌کند:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // Keep 'new' to mark this method as having no
        // relationship to the base type's Dial method.
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        // Add 'new' to mark this method as having no
        // relationship to the base type's EstablishConnection method.
        protected new virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Do work to establish the connection.
        }
    }
}
```

در این کد، کلمه کلیدی **new** به کامپایلر می‌گوید متادیتایی تولید کند که برای CLR معلوم کند با متد **EstablishConnection** از **BetterPhone** به عنوان متد جدیدی رفتار شود که توسط نوع **BetterPhone** معرفی شده است. اکنون CLR می‌داند که رابطه‌ای بین متدهای **Phone** و **BetterPhone** وجود ندارد.

وقتی همان کد برنامه قبلی (در متد **Main**) اجرا می‌شود، خروجی این گونه است:

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

این خروجی نشان می‌دهد که فراخوانی **Dial** در متد **Main**، متد جدید **Dial** تعریف شده توسط **BetterPhone.Dial** را فراخوانی می‌کند که آن هم در عوض، متد مجازی **EstablishConnection** که باز هم توسط **BetterPhone** تعریف شده است را فراخوانی می‌کند. وقتی متد **EstablishConnection** از **BetterPhone** برمی‌گردد، متد **Dial** از **Phone** فراخوانی می‌شود. متد **Dial** از **Phone**، متد **EstablishConnection** را فراخوانی می‌کند اما چون متد **EstablishConnection** از **BetterPhone** با **new** علامت زده شده است، متد **EstablishConnection** از **BetterPhone** به عنوان یک بازنویسی متد مجازی **EstablishConnection** از **Phone** در نظر گرفته نمی‌شود. در نتیجه، متد **Dial** از **Phone**، متد **EstablishConnection** از **Phone** را فراخوانی می‌کند — این رفتار مورد انتظار است.

نکته اگر کامپایلرها به صورت پیش فرض با متدها به عنوان باز نویسی شده رفتار می‌کردند (همانگونه که کامپایلر C++ اصلی انجام می‌دهد)، برنامه‌نویس نوع **BetterPhone** نمی‌توانست از نام‌های متد **Dial** و **EstablishConnection** استفاده کند. این احتمالاً منجر به موجی از تغییرات در سرتاسر کد پایه و از بین رفتن سازگاری سورس و باینری می‌گشت. این گونه تغییرات فراگیر نامطلوب است، به خصوص در هر پروژه‌ی نسبتاً بزرگ. اما اگر تغییر نام متدها فقط به آپدیت‌های کوچکی در سورس کد منجر می‌شود، شما باید نام متدها را تغییر دهید تا معانی متفاوت **Dial** و **EstablishConnection** دیگر برنامه‌نویسان را گیج نکند.

متنابا، CompanyB می‌توانست با نسخه‌ی جدید نوع **Phone** از CompanyA به این نتیجه برسد که معنای **Dial** و **EstablishConnection** از **Phone** دقیقاً همان چیزی است که به دنبال آن است. در این مورد، CompanyB، نوع **BetterPhone** خود را با حذف کامل متد **Dial** تغییر می‌دهد. افزون بر این، چون CompanyB اکنون می‌خواهد به کامپایلر بگوید که متد **EstablishConnection** از **BetterPhone** به متد **EstablishConnection** از **Phone** ربط دارد، کلمه کلیدی **new** باید حذف گردد. تنها حذف کلمه کلیدی **new** کافی نیست، چون اکنون کامپایلر نمی‌تواند تعیین کند منظور واقعی متد **EstablishConnectin** از **BetterPhone** چیست. برای بیان دقیق منظور خود، برنامه‌نویس CompanyB باید متد **EstablishConnection** از **BetterPhone** را از **virtual** به **override** تغییر دهد. کد زیر نسخه‌ی جدید از **BetterPhone** را نشان می‌دهد:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // Delete the Dial method (inherit Dial from base).

        // Remove 'new' and change 'virtual' to 'override' to
        // mark this method as having a relationship to the base
        // type's EstablishConnection method.
        protected override void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Do work to establish the connection.
        }
    }
}
```

حال وقتی همان کد برنامه (در متد **Main**) اجرا می‌شود، خروجی زیر تولید می‌گردد:

```
Phone.Dial
BetterPhone.EstablishConnection
```

این خروجی نشان می‌دهد که فراخوانی **Dial** درون متد **Main**، متد **Dial** تعریف شده توسط **Phone** و ارث برده شده توسط **BetterPhone** را فراخوانی می‌کند. سپس وقتی متد **Dial** از **Phone**، متد مجازی **EstablishConnection** را فراخوانی می‌کند، متد **EstablishConnection** از **BetterPhone** فراخوانی می‌شود، چون آن، متد مجازی **EstablishConnection** تعریف شده توسط **Phone** را بازنویسی می‌کند.

فصل ۷: ثابت‌ها و فیلدها

در این فصل، من نشان می‌دهم چگونه اعضای داده‌ای را به یک نوع اضافه کنید. به خصوص، به فیلدها و ثابت‌ها نگاه می‌اندازیم.

ثابت‌ها

یک ثابت نمادی است که یک مقدار بدون تغییر دارد. هنگام تعریف یک نماد ثابت، مقدار آن باید در زمان کامپایل قابل تعیین باشد. کامپایلر، سپس مقدار ثابت را در متادیتای اسمبلی ذخیره می‌کند. این یعنی، شما می‌توانید تنها برای نوع‌هایی که کامپایلر به عنوان نوع‌های اصلی می‌شناسد، ثابت تعریف کنید. در سی‌شارپ، این نوع‌ها، اصلی هستند و می‌توانند برای تعریف ثابت‌ها استفاده شوند: **Boolean**، **Char**، **Byte**، **SByte**، **Int16**، **UInt16**، **Int32**، **UInt32**، **Single**، **Double**، **Decimal** و **String**. هر چند، سی‌شارپ اجازه می‌دهد که شما یک متغیر ثابت از یک نوع غیر اصلی تعریف کنید اگر مقدار آن را برابر با **null** قرار دهید:

```
using System;
```

```
public sealed class SomeType {
    // SomeType is not a primitive type but C# does allow
    // a constant variable of this type to be set to 'null'.
    public const SomeType Empty = null;
}
```

چون یک مقدار ثابت هرگز تغییر نمی‌کند، ثابت‌ها همیشه به عنوان بخشی از نوع تعریف کنند، در نظر گرفته می‌شوند. به بیان دیگر، ثابت‌ها همیشه عضو استاتیک در نظر گرفته می‌شوند و نه عضو نمونه. تعریف یک ثابت منجر به ایجاد متادیتا می‌شود.

وقتی کدی به یک نماد ثابت اشاره می‌کند، کامپایلر، نماد را در متادیتای اسمبلی‌ای که ثابت را تعریف می‌کند جست و جو کرده و مقدار ثابت را به دست می‌آورد و مقدار را در کد خروجی زبان میانی (IL) تعبیه می‌کند. چون مقدار یک ثابت مستقیماً در کد تعبیه می‌شود، ثابت‌ها نیاز به تخصیص حافظه در زمان اجرا ندارند. این محدودیت‌ها، همچنین به این معنیست که ثابت‌ها برای نسخه‌بندی بین اسمبلی‌ها، داستان خوبی به دنبال ندارند، پس باید از آن‌ها تنها وقتی که مطمئن هستید مقدار یک نماد هرگز تغییر نمی‌کند، استفاده کنید. (تعریف **MaxInt16** به عنوان **32767** یک مثال خوب است). بگذارید نشان دهم دقیقاً منظورم چیست. اول، کد زیر را بردارید و آن را به یک اسمبلی DLL کامپایل کنید:

```
using System;
```

```
public sealed class SomeLibraryType {
    // NOTE: C# doesn't allow you to specify static for constants
    // because constants are always implicitly static.
    public const Int32 MaxEntriesInList = 50;
}
```

سپس از کد زیر برای ساخت یک برنامه اسمبلی استفاده کنید

```
using System;
```

```
public sealed class Program {
    public static void Main() {
        Console.WriteLine("Max entries supported in list: "
            + SomeLibraryType.MaxEntriesInList);
    }
}
```

شما متوجه می‌شوید که کد این برنامه به ثابت **MaxEntriesInList** که در کلاس **SomelibraryType** تعریف شده است اشاره می‌کند. وقتی کامپایلر، کد برنامه را می‌سازد، می‌بینید که **MaxEntriesInList** یک لیترال ثابت با مقدار ۵۰ است پس **Int32** با مقدار ۵۰ را درست درون کد IL برنامه تعبیه می‌کند. همانگونه که در کد IL زیر می‌توانید ببینید. در واقع، پس از ساخت اسمبلی برنامه، اسمبلی DLL اصلا در زمان اجرا بارگذاری نمی‌شود و می‌تواند از دیسک حذف گردد.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 25 (0x19)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Max entries supported in list: "
    IL_0006: ldc.i4.s 50
    IL_0008: box [mscorlib]System.Int32
    IL_000d: call string [mscorlib]System.String::Concat(object, object)
    IL_0012: call void [mscorlib]System.Console::WriteLine(string)
    IL_0017: nop
    IL_0018: ret
} // end of method Program::Main
```

این مثال باید مسئله نسخه‌بندی را برای شما روشن کند. اگر برنامه‌نویس، ثابت **MaxEntriesInList** را به ۱۰۰۰ تغییر دهد و تنها اسمبلی DLL را دوباره بسازد، اسمبلی برنامه تغییر نمی‌کند. برای آنکه برنامه مقدار جدید را استفاده کند، باید مجدداً از نو کامپایل شود. شما نمی‌توانید از ثابت‌ها استفاده کنید اگر نیاز دارید یک مقدار در یک اسمبلی توسط اسمبلی دیگری در زمان اجرا (به جای زمان کامپایل) استفاده شود. به جای آن، شما می‌توانید از فیلدهای **readonly** استفاده کنید که در ادامه آن‌ها را می‌گوییم.

فیلدها

یک فیلد یک عضو داده‌ای است که یک نمونه از یک نوع مقداری یا ارجاع به یک نوع ارجاعی را نگه می‌دارد. جدول ۷-۱ تغییر دهنده‌هایی که می‌توان به فیلدها اعمال کرد را نشان می‌دهد.

جدول ۷-۱ تغییر دهنده‌های فیلد

واژه CLR	واژه سی‌شارپ	توضیح
Static	static	فیلد بخشی از وضعیت نوع است، درست در مقابل اینکه بخشی از وضعیت یک شی باشد.
Instance	(پیش فرض)	فیلد همراه با نمونه‌ای از نوع است نه با خود نوع.
InitOnly	readonly	فیلد تنها توسط کد درون یک متد سازنده قابل نوشتن است.
Volatile	volatile	کدی که به فیلد دسترسی پیدا می‌کند در معرض بهینه‌سازی غیرامن ترد که ممکن است توسط کامپایلر، CLR یا سخت افزار انجام شود نیست. تنها نوع‌های زیر را می‌توان با volatile علامت زد: تمام نوع‌های ارجاعی، Single ، Boolean ، Byte ، SByte ، Int16 ، UInt16 ، Int32 ، Char ، UInt32 و تمام نوع‌های شمارشی که از نوع Byte ، SByte ، Int16 ، UInt16 ، Int32 یا UInt32 باشند. فیلدهای فرار در فصل ۲۸ "ساخت‌های اصلی همزمانی ترد" بحث می‌شوند.

همانگونه که جدول ۷-۱ نشان می‌دهد، زبان مشترک اجرایی (CLR)، هر دوی فیلدهای نوع (استاتیک) و نمونه (غیر استاتیک) را پشتیبانی می‌کند. برای فیلدهای نوع، حافظه‌ی پویای مورد نیاز برای نگهداری داده‌ی فیلد، درون شی نوع تخصیص داده می‌شود، که هنگام بارگذاری نوع درون یک **AppDomain** ساخته می‌شود (فصل ۲۲ "میزبانی CLR و **AppDomain** ها" را ببینید) که نوعاً، اولین بار که هر متدی نوع را ارجاع کرده، JIT کامپایل شود، رخ می‌دهد. برای فیلدهای نمونه، حافظه‌ی پویا برای نگهداری فیلد، هنگامی که یک نمونه از نوع ساخته می‌شود ایجاد می‌گردد.

چون فیلدها در حافظه‌ی پویا ذخیره می‌شوند، مقدار آنها فقط در زمان اجرا قابل دستیابی است. فیلدها همچنین مسئله نسخه‌بندی که با ثابت‌ها وجود داشت را برطرف می‌کنند. به اضافه، یک فیلد می‌تواند از هر نوعی باشد، پس شما مجبور نیستید خود را به نوع‌های اصلی کامپایلر (آنگونه که برای ثابت‌ها بودید)، محدود کنید.

CLR از فیلدهای **readonly** و فیلدهای **read/write** پشتیبانی می‌کند. اکثر فیلدها، **read/write** هستند به این معنی که مقدار فیلد هنگام اجرای کد ممکن است تغییر کند. اما، فیلدهای **readonly** فقط درون متد سازنده قابل نوشتن هستند (که فقط یکبار فراخوانی می‌شود آن هم هنگام ساخت یک شی). کامپایلرها و مرحله بازبینی مطمئن می‌شوند که فیلدهای **readonly** توسط هیچ متدی جز سازنده نوشته نشوند. توجه کنید که برای تغییر یک فیلد **readonly** می‌توان از رفلکشن استفاده نمود.

بگذارید مثال "ثابت‌ها" را برداشته و مسئله نسخه‌بندی را با استفاده از یک فیلد استاتیک **readonly** حل کنیم. در اینجا نسخه‌ی جدید کد اسمبلی DLL بدین گونه است:

```
using System;
```

```
public sealed class SomeLibraryType {
    // The static is required to associate the field with the type.
    public static readonly Int32 MaxEntriesInList = 50;
}
```

این تنها تغییری است که می‌دهیم، کد برنامه اصلاً نیاز به تغییر ندارد، هر چند شما باید برای دیدن رفتار جدید آن را مجدداً کامپایل کنید. حال، وقتی متد **Main** از برنامه اجرا می‌شود، CLR، اسمبلی DLL را بارگذاری می‌کند (سپس این اسمبلی اکنون در زمان اجرا مورد نیاز است) و مقدار فیلد **MaxEntriesInList** را از حافظه‌ی پویای تخصیص یافته به آن، بر می‌دارد. البته که مقدار، ۵۰ خواهد بود.

فرض کنید که برنامه‌نویس اسمبلی DLL، ۵۰ را به ۱۰۰۰ تغییر دهد و اسمبلی را از نو بسازد. وقتی کد برنامه مجدداً اجرا می‌شود، آن به صورت خودکار مقدار جدید یعنی ۱۰۰۰ را برخواهد داشت. در این مورد، کد برنامه نیاز به ساخت مجدد ندارد و به خوبی کار می‌کند (اگرچه سرعتش کمتر شده است). یک اخطار: این سناریو فرض می‌کند که نسخه جدید اسمبلی DLL، قوی‌نام نیست و سیاست نسخه‌بندی برنامه به گونه‌ای است که CLR این نسخه جدید را بارگذاری می‌کند.

مثال زیر نشان می‌دهد چگونه یک فیلد استاتیک **readonly** که با خود نوع همراه است، به همراه فیلدهای استاتیک **read/write** و فیلدهای نمونه‌ی **readonly** و **read/write** تعریف کنید:

```
public sealed class SomeType {
    // This is a static read-only field; its value is calculated and
    // stored in memory when this class is initialized at run time.
    public static readonly Random s_random = new Random();

    // This is a static read/write field.
    private static Int32 s_numberOfWrites = 0;

    // This is an instance read-only field.
    public readonly String Pathname = "Untitled";

    // This is an instance read/write field.
    private System.IO.FileStream m_fs;
    public SomeType(String pathname) {
        // This line changes a read-only field.
        // This is OK because the code is in a constructor.
        this.Pathname = pathname;
    }

    public string DoSomething() {
```

```

// This line reads and writes to the static read/write field.
s_numberOfWrites = s_numberOfWrites + 1;
// This line reads the read-only instance field.
return Pathname;
}
}

```

در این کد، بسیاری از فیلدها به صورت خطی (**inline**) مقداردهی اولیه شده اند. سی شارپ اجازه می‌دهد از این نحو مقداردهی اولیه خطی برای مقداردهی اولیه فیلدهای **read/write** و **readonly** و ثابت‌های یک کلاس استفاده کنید.

همانگونه که در فصل ۸ "متدها" خواهید دید، سی شارپ با مقداردهی خطی یک فیلد تحت عنوان نحو خلاصه برای مقداردهی فیلد در یک سازنده، رفتار می‌کند. همچنین در سی شارپ، چند مسئله‌ی کارایی وجود دارد که هنگام مقداردهی اولیه فیلدها با نحو خطی در مقابل نحو انتساب در یک سازنده پیش می‌آید، باید مورد توجه قرار گیرد. این مسائل کارایی در فصل ۸ بحث می‌شوند.

مهم وقتی یک فیلد از یک نوع ارجاعی است و فیلد با **readonly** علامت زده شده است، چیزی که تغییر ناپذیر است ارجاع می‌باشد نه شی‌ای که فیلد بدان اشاره دارد. کد زیر مسئله را روشن می‌کند:

```

public sealed class AType {
    // InvalidChars must always refer to the same array object
    public static readonly Char[] InvalidChars = new Char[] { 'A', 'B', 'C' };
}

public sealed class AnotherType {
    public static void M() {
        // The lines below are legal, compile, and successfully
        // change the characters in the InvalidChars array
        AType.InvalidChars[0] = 'X';
        AType.InvalidChars[1] = 'Y';
        AType.InvalidChars[2] = 'Z';

        // The line below is illegal and will not compile because
        // what InvalidChars refers to cannot be changed
        AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };
    }
}
}

```

فصل ۸: متدها

این فصل بر گونه‌های مختلف متدها که شما با آن‌ها سروکار دارید، شامل سازنده‌های نمونه و سازنده‌های نوع، همچنین چگونگی تعریف متدهایی برای سربرگذاری عملگرها و تبدیل‌های نوع (تبدیل ضمنی و صریح) تمرکز دارد. ما همچنین پیرامون متدهای گسترشی که به شما اجازه می‌دهند به صورت منطقی متدهای نمونه خود را به نوع‌های موجود بیافزایید و متدهای جزئی که اجازه تقسیم کردن پیاده‌سازی یک نوع به چند بخش را می‌دهند، بحث می‌کنیم.

سازنده‌های نمونه و کلاس‌ها (نوع‌های ارجاعی)

سازنده‌ها متدهای ویژه‌ای هستند که اجازه می‌دهند یک نمونه از یک نوع با یک مقدار مناسب مقداردهی اولیه شود. متدهای سازنده همیشه با عنوان **ctor** (برای **constructor**) در جدول متادیتای یک متد می‌آیند. هنگام ساخت یک نمونه از یک نوع ارجاعی، حافظه برای فیلدهای داده‌ای نمونه تخصیص داده می‌شود، فیلدهای اضافی شی (اشاره‌گر شی نوع و اندیس بلوک همزمانی) مقداردهی اولیه می‌شوند، و سپس سازنده‌ی نمونه برای تنظیم مقدار اولیه شی فراخوانی می‌شود.

هنگام ساخت یک شی از نوع ارجاعی، حافظه‌ی تخصیص یافته برای شی همیشه قبل از فراخوانی سازنده‌ی نمونه، صفر می‌شود. هر فیلدی را که سازنده صریحا بر روی آن ننویسد، تضمین می‌شود که دارای مقدار **0** یا **null** باشد.

برخلاف دیگر متدها، متدهای سازنده هرگز به ارث برده نمی‌شوند. یعنی اینکه، یک کلاس، تنها سازنده‌های نمونه‌ای را دارد که خودش تعریف کرده است. چون سازنده‌های نمونه هرگز به ارث برده نمی‌شوند، شما نمی‌توانید تغییردهنده‌های زیر را به یک سازنده‌ی نمونه اعمال کنید: **new**، **virtual**، **sealed** یا **abstract**. اگر شما کلاسی تعریف کنید که صریحا هیچ سازنده‌ای را تعریف نکرده باشد، کامپایلر سی‌شارپ یک سازنده (بدون پارامتر) پیش فرض برای شما تعریف می‌کند که پیاده‌سازی آن فقط سازنده‌ی بدون پارامتر از کلاس پایه را فراخوانی می‌کند. برای نمونه اگر شما کلاس زیر را تعریف کنید:

```
public class SomeType {
}
```

همانند آن است که کد را شبیه به این بنویسید:

```
public class SomeType {
    public SomeType() : base() { }
}
```

اگر کلاس **abstract** باشد، سازنده‌ی پیش فرض تولید شده توسط کامپایلر دارای پدیداری **protected** است و در غیر این صورت سازنده دارای پدیداری **public** خواهد بود. اگر کلاس پایه یک سازنده‌ی بدون پارامتر تعریف نکرده باشد، کلاس مشتق شده باید صریحا یک سازنده از کلاس پایه را فراخوانی کند وگرنه کامپایلر اعلام خطا می‌کند. اگر کلاس **static** باشد (**sealed** و **abstract**)، کامپایلر، یک سازنده‌ی پیش فرض را در تعریف کلاس تولید نمی‌کند.

یک نوع می‌تواند چندین سازنده‌ی نمونه تعریف کند. هر سازنده باید یک امضای متفاوت داشته باشد و هر کدام می‌توانند پدیداری متفاوتی داشته باشند. برای آنکه کد قابل بررسی باشد، سازنده‌ی نمونه یک کلاس باید قبل از دسترسی به هر یک از فیلدهای به ارث برده از کلاس پایه، سازنده کلاس پایه‌اش را فراخوانی کند. کامپایلر سی‌شارپ اگر سازنده‌ی کلاس مشتق شده صریحا یکی از سازنده‌های کلاس پایه را صدا نکند، به صورت خودکار یک فراخوانی به سازنده پیش فرض کلاس پایه، تولید می‌کند. سرانجام، سازنده‌ی عمومی و بدون پارامتر **System.Object**، فراخوانی می‌شود. این سازنده کاری انجام نمی‌دهد و فقط برمی‌گردد. این بدین علت است که **System.Object** هیچ فیلد داده‌ای تعریف نمی‌کند. پس سازنده‌ی آن کاری برای انجام دادن ندارد.

در موارد کمی، یک نمونه از یک نوع را می‌تواند بدون فراخوانی یک سازنده نمونه، ایجاد کرد. به خصوص، فراخوانی متد **MemberwiseClone** از **Object**، حافظه را تخصیص می‌دهد، فیلدهای اضافی شی را مقداردهی اولیه می‌کند و بایت‌های شی مبدا را به شی جدید کپی می‌کند. همچنین، هنگام غیرسریالی کردن یک شی با سریالی کننده‌ی زمان اجرا، یک سازنده معمولا فراخوانی نمی‌شود. کد غیرسریالی کننده، بدون آنکه یک سازنده را فراخوانی کند حافظه را با متدهای **GetUninitializedObject** و **GetSafeUninitializedObject** از

System.Runtime.Serialization.FormatterServices (همانگونه که در فصل ۲۴ "سریالی کردن زمان اجرا" بحث می‌شود) تخصیص

می‌دهد.

مهم شما نباید هیچ متد مجازی را درون یک سازنده که می‌تواند بر شی در حال ساخت اثر گذارد، فراخوانی کنید. علت آن است که اگر متد مجازی در نوعی که در حال نمونه سازی است، بازنویسی شده باشد، متد بازنویسی شده در نوع مشتق شده اجرا خواهد شد اما هنوز همه‌ی فیلدها کاملاً مقداردهی اولیه نشده اند. فراخوانی یک متد مجازی منجر به رفتار غیر قابل پیش‌بینی خواهد شد.

سی‌شارپ یک نحو ساده ارائه می‌کند که اجازه می‌دهد فیلدهای تعریف شده در یک نوع ارجاعی وقتی یک نمونه از نوع ساخته می‌شود، مقداردهی اولیه شوند:

```
internal sealed class SomeType {
    private Int32 m_x = 5;
}
```

وقتی یک شی **SomeType** ساخته می‌شود، فیلد **m_x** آن با **5** مقداردهی اولیه می‌گردد. این چگونه اتفاق می‌افتد؟ خوب، اگر شما **IL** را برای متد سازنده‌ی **SomeType** (**.ctor**) بررسی کنید، کد زیر را خواهید دید:

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size 14 (0xe)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldc.i4.5
    IL_0002: stfld int32 SomeType::m_x
    IL_0007: ldarg.0
    IL_0008: call instance void [mscorlib]system.Object::.ctor()
    IL_000d: ret
} // end of method SomeType::.ctor
```

در این کد، شما می‌بینید که سازنده‌ی **SomeType** شامل کدی برای ذخیره‌ی یک **5** در **m_x** و سپس فراخوانی سازنده‌ی کلاس پایه است. به بیان دیگر، کامپایلر سی‌شارپ اجازه می‌دهد با نحو آسان فیلدهای نمونه را به صورت خطی مقداردهی کنید و کامپایلر این نحو را به کد درون متد سازنده برای انجام مقداردهی ترجمه می‌کند. این یعنی شما باید از انفجار کد که در تعریف کلاس زیر نمایش داده شده است، آگاه باشید:

```
internal sealed class SomeType {
    private Int32 m_x = 5;
    private String m_s = "Hi there";
    private Double m_d = 3.14159;
    private Byte m_b;

    // Here are some constructors.
    public SomeType() { ... }
    public SomeType(Int32 x) { ... }
    public SomeType(String s) { ...; m_d = 10; }
}
```

وقتی کامپایلر کد را برای سه متد سازنده تولید می‌کند، ابتدای هر متد شامل کدی برای مقداردهی اولیه **m_x**، **m_s** و **m_d** است. پس از این کد مقداردهی اولیه، کامپایلر یک فراخوانی به سازنده‌ی کلاس پایه قرار می‌دهد و سپس کامپایلر کدی که در متدهای سازنده نمایش داده شده است را به متد اضافه می‌کند. برای نمونه، کد تولید شده برای سازنده‌ای که یک پارامتر **String** می‌گیرد شامل کد مقداردهی اولیه **m_x**، **m_s** و **m_d**، فراخوانی

سازنده‌ی کلاس پایه (Object) و سپس بازنویسی `m_d` با مقدار 10 است. توجه کنید که `m_b` به صورت تضمینی 0 است حتی اگر کدی برای مقداردهی صریح آن وجود نداشته باشد.

نکته کامپایلر، هر فیلدی با نحو آسان را قبل از فراخوانی یک سازنده‌ی کلاس پایه، مقداردهی اولیه می‌کند تا نشان دهد که این فیلدها همواره مقداری که سورس کد تعیین می‌کند را دارند. مشکل احتمالی وقتی رخ می‌دهد که سازنده‌ی یک کلاس پایه، متد مجازی‌ای را صدا می‌زند که منجر به فراخوانی متدی در کلاس مشتق شده می‌شود. اگر این اتفاق بیافتد، فیلدهای مقداردهی شده با نحو آسان، قبل از آنکه متد مجازی فراخوانی شود، مقدار دهی شده بوده اند.

چون در کلاس قبلی سه سازنده وجود دارد، کامپایلر کد برای مقداردهی اولیه `m_x`، `m_s` و `m_d` را سه بار تولید می‌کند – یکبار برای هر سازنده. اگر شما چندین فیلد مقداردهی اولیه شده و تعداد زیادی متد سازنده‌ی سربرگذاری شده دارید، شما باید فیلدها را بدون مقداردهی اولیه تعریف کنید، یک سازنده بسازید که این مقداردهی‌های مشترک را انجام دهد و هر سازنده صریحاً این سازنده‌ی مشترک را برای مقداردهی فراخوانی کند. با این روش، اندازه کد تولیدی کاهش می‌یابد. در اینجا مثالی می‌بینید که از قابلیت سی‌شارپ در فراخوانی صریح یک سازنده توسط سازنده‌ی دیگر به کمک کلمه کلیدی `this` استفاده می‌کند:

```
internal sealed class SomeType {
    // Do not explicitly initialize the fields here
    private Int32 m_x;
    private String m_s;
    private Double m_d;
    private Byte m_b;

    // This constructor sets all fields to their default.
    // All of the other constructors explicitly invoke this constructor.
    public SomeType() {
        m_x = 5;
        m_s = "Hi there";
        m_d = 3.14159;
        m_b = 0xff;
    }

    // This constructor sets all fields to their default, then changes m_x.
    public SomeType(Int32 x) : this() {
        m_x = x;
    }

    // This constructor sets all fields to their default, then changes m_s.
    public SomeType(String s) : this() {
        m_s = s;
    }

    // This constructor sets all fields to their default, then changes m_x & m_s.
    public SomeType(Int32 x, String s) : this() {
        m_x = x;
        m_s = s;
    }
}
```

سازنده های نمونه و ساختارها (نوع های مقداری)

سازنده های نوع مقداری (**struct**) کاملا متفاوت از سازنده های نوع ارجاعی (**class**) عمل می کنند. CLR همواره اجازه ی ساخت نمونه های نوع مقداری را می دهد و راهی برای جلوگیری از نمونه سازی از یک نوع مقداری وجود ندارد. به همین دلیل، نوع های مقداری واقعا نیاز به تعریف یک سازنده درون خود ندارند و کامپایلر سی شارپ سازنده های بدون پارامتر پیش فرض را برای نوع های مقداری تولید نمی کند. کد زیر را بررسی کنید:

```
internal struct Point {
    public Int32 m_x, m_y;
}
internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;
}
```

برای ساخت یک **Rectangle** عملگر **new** باید استفاده شود و یک سازنده باید تعیین گردد. در این مورد، سازنده ی پیش فرض که به صورت خودکار توسط کامپایلر سی شارپ تولید شده است فراخوانی می شود. وقتی حافظه برای **Rectangle** تخصیص داده شد، حافظه شامل دو نمونه از نوع مقداری **Point** است. به دلایل مربوط به کارایی، CLR سعی نمی کند برای هر نوع مقداری درون نوع ارجاعی، یک سازنده را فراخوانی کند. اما همانطور که قبلا گفتیم، فیلدهای متعلق به نوع مقداری به **0** یا **null** مقداردهی اولیه می شوند.

CLR اجازه می دهد سازنده هایی برای نوع مقداری تعریف کنید. تنها راهی که این سازنده ها می توانند اجرا شوند آن است که شما کدی بنویسید که صریحا یکی از آن ها را فراخوانی کند، همانند آنچه در سازنده ی **Rectangle** در زیر آمده است:

```
internal struct Point {
    public Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
        // In C#, new on a value type calls the constructor to
        // initialize the value type's fields.
        m_topLeft = new Point(1, 2);
        m_bottomRight = new Point(100, 200);
    }
}
```

سازنده ی یک نوع مقداری، تنها زمانی که صریحا فراخوانی شود، اجرا می گردد. پس اگر سازنده ی **Rectangle**، فیلدهای **m_topLeft** و **m_bottomRight** خود را با عملگر **new** و در پی آن فراخوانی سازنده ی **Point**، مقداردهی اولیه نکند، فیلدهای **m_x** و **m_y** در دو فیلد **Point**، **0** خواهند بود. در نوع مقداری **Point** که قبلا تعریف شده، هیچ سازنده ی بدون پارامتر پیش فرضی تعریف نشده است. اما اجازه دهید کد را به صورت زیر بنویسیم:

```
internal struct Point {
    public Int32 m_x, m_y;

    public Point() {
        m_x = m_y = 5;
    }
}
```

```

    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
    }
}

```

حال وقتی یک **Rectangle** جدید ساخته می‌شود، فکر می‌کنید فیلدهای **m_x** و **m_y** در دو فیلد **Point**، **m_topLeft** و **m_bottomRight** به چه چیزی مقداردهی اولیه می‌شوند؟ **0** یا **5** (راهنمایی: این یک سوال حقه آمیز است)

بسیاری از برنامه‌نویسان (به خصوص آن‌هایی که پیش زمینه‌ی ++C دارند) فکر می‌کنند کامپایلر سی‌شارپ، کدی در سازنده‌ی **Rectangle** تولید می‌کند که به صورت خودکار سازنده‌ی بدون پارامتر پیش فرض **Point** را برای دو فیلد متعلق به **Rectangle** فراخوانی کند. در واقع، بسیاری از کامپایلرها هرگز کدی که سازنده پیش فرض یک نوع مقداری را فراخوانی کند تولید نمی‌کنند حتی اگر نوع، یک سازنده بدون پارامتر را تعریف کند. برای آنکه سازنده بدون پارامتر یک نوع مقداری اجرا شود، برنامه‌نویس باید کدی اضافه کند که صریحاً سازنده‌ی یک نوع مقداری را فراخوانی کند.

بر طبق اطلاعات پاراگراف قبل، شما باید انتظار داشته باشید که فیلدهای **m_x** و **m_y** در دو فیلد **Point** از **Rectangle**، با مقدار **0** مقداردهی اولیه شوند چون هیچ فراخوانی صریحی به سازنده‌ی **Point** در هیچ جای کد وجود ندارد.

هرچند، من گفتم که سؤال یک سوال حقه آمیز است. بخش حقه‌ی آن اینست که سی‌شارپ اجازه نمی‌دهد که یک نوع مقداری، یک سازنده بدون پارامتر تعریف کند. پس کد قبلی اصلاً کامپایل نمی‌شود. کامپایلر سی‌شارپ هنگام تلاش برای کامپایل کد، پیام زیر را نشان می‌دهد:

“error CS0568: Structs cannot contain explicit parameterless constructors”

سی‌شارپ مفرضانه اجازه‌ی تعریف سازنده بدون پارامتر توسط نوع مقداری را سلب می‌کند تا جلوی هرگونه سردرگمی که برنامه‌نویس درباره‌ی اینکه چه وقت سازنده فراخوانی می‌شود را بگیرد. اگر سازنده نتواند تعریف شود، کامپایلر هرگز کدی برای فراخوانی خودکار آن تولید نمی‌کند. بدون یک سازنده بدون پارامتر، فیلدهای یک نوع مقداری همواره با **0** یا **null** مقداردهی اولیه می‌شوند.

نکته صریحاً بگوییم که فیلدهای نوع مقداری همیشه تضمینی **0/null** هستند وقتی که نوع مقداری، فیلدی تودرتو درون یک نوع ارجاعی باشد. اما، فیلدهای نوع مقداری درون پشته، به صورت تضمینی **0** یا **null** نیستند. برای قابل بررسی بودن، هر فیلد نوع مقداری درون پشته باید قبل از آنکه خوانده شود، نوشته شود. اگر کدی سعی در خواندن یک فیلد نوع مقداری قبل از نوشتن در فیلد کند، احتمال مشکلات امنیتی وجود دارد. سی‌شارپ و دیگر کامپایلرهایی که کد قابل بازبینی تولید می‌کنند، مطمئن می‌شوند که همه‌ی نوع‌های مقداری درون پشته، قبل از خوانده شدن، تمام فیلدهایشان صفر شده یا حداقل نوشته شده باشند تا در زمان اجرای اکسپشن، بازبینی رخ ندهد. برای اغلب موارد، این یعنی شما می‌توانید فرض کنید که فیلدهای نوع مقداری شما با **0** مقداردهی اولیه شده‌اند و شما می‌توانید هر آنچه در این نکته گفته شد را کاملاً نادیده بگیرید.

به خاطر داشته باشد اگرچه سی‌شارپ اجازه نمی‌دهد نوع‌های مقداری، سازنده‌های بدون پارامتر داشته باشند اما CLR این اجازه را می‌دهد. پس اگر رفتار غیر واضحی که توضیح داده شد شما را اذیت نمی‌کند، می‌توانید از زبان برنامه‌نویسی دیگری (مثل زبان اسمبلی IL) برای تعریف نوع مقداری خود با یک سازنده بدون پارامتر استفاده کنید.

چون سی‌شارپ اجازه نمی‌دهد نوع‌های مقداری، سازنده‌های بدون پارامتر داشته باشند، کامپایلر نوع زیر منجر به تولید این خطا می‌شود:

“errr CS0573: ‘SomeValType.m_x’: cannot have instance field initializers in structs.”

```

internal struct SomeValType {
    // You cannot do inline instance field initialization in a value type
    private Int32 m_x = 5;
}

```

به علاوه، چون کد قابل بازبینی نیاز دارد که هر فیلد از یک نوع مقداری قبل از خواندن، نوشته شود، هر سازنده‌ای که برای یک نوع مقداری دارید باید تمام فیلدهای نوع را مقداردهی اولیه کند:

```
internal struct SomeValType {
    private Int32 m_x, m_y;

    // C# allows value types to have constructors that take parameters.
    public SomeValType(Int32 x) {
        m_x = x;
        // Notice that m_y is not initialized here.
    }
}
```

وقتی این نوع کامپایل می‌شود، کامپایلر سی‌شارپ این پیام را اعلام می‌کند:

“error CS0171: Field ‘SomeValType.m_y’ must be fully assigned before control leaves the constructor.”

برای حل این مشکل، در سازنده، یک مقدار (معمولا 0) را به **m_y** نسبت دهید. راه جایگزین برای مقداردهی تمام فیلدهای یک نوع مقداری، انجام کار زیر است:

```
// C# allows value types to have constructors that take parameters.
public SomeValType(Int32 x) {
    // Looks strange but compiles fine and initializes all fields to 0/null
    this = new SomeValType();

    m_x = x; // Overwrite m_x's 0 with x
    // Notice that m_y was initialized to 0.
}
```

در سازنده‌ی یک نوع مقداری، **this** یک نمونه از خود نوع مقداری را نشان می‌دهد و شما واقعا می‌توانید حاصل **new** کردن از یک نوع مقداری را به آن نسبت دهید که در حقیقت تمام فیلدها را صفر می‌کند. در سازنده‌ی یک نوع ارجاعی، **this** فقط-خواندنی است و نمی‌توانید چیزی را به آن نسبت دهید.

سازنده‌های نوع

علاوه بر سازنده‌های نمونه، CLR از سازنده‌های نوع (که با عناوین سازنده‌های استاتیک، سازنده‌های کلاس یا مقداردهی‌کننده‌های نوع نیز شناخته می‌شوند) نیز پشتیبانی می‌کند. یک سازنده‌ی نوع می‌تواند بر رابطه‌ها (اگرچه سی‌شارپ اجازه نمی‌دهد)، نوع‌های ارجاعی و نوع‌های مقداری اعمال شود. درست همانگونه که سازنده‌های نمونه برای تنظیم وضعیت اولیه یک نمونه از نوع استفاده می‌شوند، سازنده‌های نوع هم برای تنظیم وضعیت اولیه یک نوع استفاده می‌شوند. به صورت پیش فرض، نوع‌ها دارای یک سازنده‌ی نوع تعریف شده درونشان نیستند. اگر یک نوع یک سازنده‌ی نوع دارد، نمی‌تواند بیش از یکی داشته باشد. به علاوه، سازنده‌های نوع هرگز پارامتر ندارند. در سی‌شارپ، روش تعریف یک نوع ارجاعی و یک نوع مقداری همراه با سازنده‌های نوع را می‌بینید:

```
internal sealed class SomeRefType {
    static SomeRefType() {
        // This executes the first time a SomeRefType is accessed.
    }
}

internal struct SomeValType {
    // C# does allow value types to define parameterless type constructors.
    static somevaltype() {
        // This executes the first time a SomeValType is accessed.
    }
}
```



```

}
}

```

می بینید که سازنده‌های نوع را درست همانند سازنده‌های نمونه‌ی بدون پارامتر تعریف می‌کنید به جز آنکه باید آن‌ها را **static** کنید. همچنین، سازنده‌های نوع همیشه باید خصوصی باشند. سی‌شارپ آن‌ها را به صورت خودکار برای شما **private** می‌کند. در واقع، اگر شما صریحاً یک سازنده‌ی نوع را **private** (یا هر چیز دیگر) کنید، کامپایلر سی‌شارپ خطای زیر را می‌گیرد:

“error CS0515: ‘SomeValType.SomeValType()’: access modifiers are not allowed on static constructors.”

سازنده‌های نوع باید خصوصی باشند تا هیچ کد نوشته شده توسط برنامه‌نویس، نتواند آن‌ها را فراخوانی کند؛ CLR همیشه قادر به فراخوانی یک سازنده‌ی نوع است.

مهم اگرچه می‌توانید یک سازنده‌ی نوع را درون یک نوع مقداری تعریف کنید، شما هرگز نباید این کار را بکنید چون مواقعی وجود دارد که CLR، سازنده‌ی نوع استاتیک یک نوع مقداری را فراخوانی نمی‌کند. مثالی ببینید:

```

internal struct SomeValType {
    static SomeValType() {
        Console.WriteLine("This never gets displayed");
    }
    public Int32 m_x;
}

public sealed class Program {
    public static void Main() {
        SomeValType[] a = new SomeValType[10];
        a[0].m_x = 123;
        Console.WriteLine(a[0].m_x); // Displays 123
    }
}

```

فراخوانی یک سازنده‌ی نوع، یک حقه بازی است. وقتی کامپایلر فقط در لحظه (JIT) یک متد را کامپایل می‌کند، نوع‌هایی را که به آن‌ها ارجاع داده را می‌بیند. اگر هر یک از نوع‌ها یک سازنده‌ی نوع تعریف کنند، کامپایلر JIT بررسی می‌کند آیا سازنده‌ی نوع از آن نوع قبلاً درون این **AppDomain** اجرا شده است یا خیر. اگر سازنده هرگز اجرا نشده باشد، کامپایلر JIT کدی برای فراخوانی سازنده نوع در کد اصلی که کامپایلر JIT تولید می‌کند، قرار می‌دهد. اگر سازنده‌ی نوع برای آن نوع قبلاً اجرا شده باشد، کامپایلر JIT کدی برای فراخوانی تولید نمی‌کند، چون می‌داند که نوع قبلاً مقداردهی اولیه شده است. (برای یک نمونه، بخش "کارایی سازنده‌های نوع" در ادامه فصل را ببینید).

اکنون، پس از آنکه متد، JIT کامپایل شد، ترد شروع به اجرای آن می‌کند و در پی آن به کدی می‌رسد که سازنده‌ی نوع را صدا می‌زند. در واقع، این ممکن است که چندین ترد، یک متد را به صورت هم‌رند اجرا کنند. CLR می‌خواهد مطمئن شود که یک سازنده‌ی نوع فقط یکبار به ازای هر **AppDomain** اجرا می‌شود. برای تامین این، وقتی یک سازنده‌ی نوع فراخوانی شد، ترد فراخوانی‌کننده یک قفل انحصار متقابل همزمانی ترد بدست می‌آورد. پس اگر چندین ترد سعی در فراخوانی همزمان یک سازنده‌ی استاتیک نوع داشته باشند، تنها یک ترد قفل را دارد و تردهای دیگر مسدود می‌شوند. اولین ترد، کد درون سازنده‌ی استاتیک را اجرا می‌کند. پس از آنکه اولین ترد، سازنده را ترک کرد، تردهای منتظر بیدار می‌شوند و می‌بینند که کد سازنده قبلاً اجرا شده است. این تردها، کد را مجدداً اجرا نمی‌کنند و فقط از متد سازنده برمی‌گردند. به علاوه، اگر هر یک از این متدها مجدداً فراخوانی شوند، CLR می‌داند که سازنده‌ی نوع قبلاً اجرا شده است و مطمئن خواهد شد که سازنده مجدداً فراخوانی نگردد.

نکته چون CLR تضمین می‌کند که یک سازنده نوع فقط یکبار به ازای هر **AppDomain** اجرا می‌شود و این کار ترد-امن است، یک سازنده‌ی نوع مکان بسیار مناسبی برای مقداردهی اولیه هر شی یگانه (singleton) که نوع نیاز دارد، است.

درون یک ترد، یک مشکل احتمالی وجود دارد و وقتی اتفاق می‌افتد که دو سازنده‌ی نوع دارای کدی باشند که به همدیگر ارجاع کنند. برای نمونه، `ClassA` یک سازنده‌ی نوع حاوی کدی که `ClassB` را ارجاع می‌کند، دارد و `ClassB` یک سازنده‌ی نوع حاوی کدی که `ClassA` را ارجاع کند، دارد. در این وضعیت، هنوز هم `CLR` تضمین می‌کند که هر سازنده‌ی نوع فقط یکبار اجرا می‌گردد. اما نمی‌تواند تضمین کند که سازنده‌ی نوع `ClassA` قبل از سازنده‌ی نوع `ClassB` اجرا شود. شما مطمئن باید از نوشتن کدی که این سناریو را پیاده‌سازی کند، خودداری کنید. در واقع، `CLR` مسئول فراخوانی سازنده‌های نوع است، شما همیشه باید از نوشتن کدی که نیاز به اجرای سازنده‌های نوع با یک ترتیب خاص دارد، خودداری کنید.

سرانجام، اگر یک سازنده‌ی نوع، یک اکسپشن مدیریت نشده تولید کند، `CLR` نوع را غیر قابل استفاده در نظر می‌گیرد. سعی در دسترسی به هر یک از فیلدها یا متدهای نوع منجر به تولید `System.TypeInitializationException` می‌گردد.

کد درون یک سازنده‌ی نوع تنها به فیلدهای استاتیک یک نوع دسترسی دارد و هدف معمولش مقداردهی اولیه این فیلدهاست. همانند فیلدهای نمونه، سی-شارپ یک نحو ساده ارائه می‌کند که به شما اجازه‌ی مقداردهی اولیه فیلدهای استاتیک یک نوع را می‌دهد:

```
internal sealed class SomeType {
    private static Int32 s_x = 5;
}
```

نکته در حالیکه سی‌شارپ اجازه نمی‌دهد که یک نوع مقداری از نحو مقداردهی اولیه خطی برای فیلدهای نمونه استفاده کند، به شما اجازه می‌دهد که این کار را برای فیلدهای استاتیک انجام دهید. به بیان دیگر، اگر نوع `SomeType` فوق را از یک `class` به یک `struct` تغییر دهید، کد طبق انتظار کامپایل و اجرا می‌شود.

وقتی این کد ساخته می‌شود، کامپایلر به صورت خودکار یک سازنده‌ی نوع برای `SomeType` تولید می‌کند. این یعنی سورس‌کد را از اساس شبیه به زیر بنویسید:

```
internal sealed class SomeType {
    private static Int32 s_x;
    static SomeType() { s_x = 5; }
}
```

با استفاده از `ILDasm.exe`، آنچه کامپایلر واقعا تولید کرده است را می‌توان با بررسی `IL` برای سازنده‌ی نوع فهمید. متدهای سازنده‌ی نوع همیشه با عنوان `.cctor` (برای `class constructor`) در جدول متادیتای تعریف متد می‌آیند.

در کد زیر، شما می‌بینید که متد `.cctor`، `private` و `static` است. به علاوه دقت کنید که کد درون متد در واقع یک `5` درون فیلد استاتیک `s_x` بارگذاری می‌کند:

```
.method private hidebysig specialname rtspecialname static
    void .cctor() cil managed
{
    // Code size 7 (0x7)
    .maxstack 8
    IL_0000: ldc.i4.5
    IL_0001: stsfld int32 SomeType::s_x
    IL_0006: ret
} // end of method SomeType::.cctor
```

سازنده‌های نوع نباید سازنده‌ی نوع از نوع پایه را فراخوانی کنند. فراخوانی این چنینی، نیاز نیست چون هیچ یک از فیلدهای استاتیک یک نوع، از نوع پایه‌اش به ارث برده نمی‌شوند.

نکته برخی زبان‌ها، مثل جاوا، انتظار دارند که دسترسی به یک نوع، باعث شود که سازنده‌ی نوع آن و تمام سازنده‌های نوع از نوع پایه‌ی آن فراخوانی شوند. به علاوه، رابط‌هایی که توسط نوع پیاده‌سازی شده‌اند نیز باید سازنده‌های نوعشان، فراخوانی شوند. CLR این رفتار را ارائه نمی‌کند. هر چند CLR برای کامپایلرها و برنامه‌نویس‌ها قابلیت فراهم کردن این رفتار را از طریق متد **RunClassConstructor** که توسط نوع **System.Runtime.CompilerServices.RuntimeHelpers** تعریف می‌شود را ارائه می‌کند. هر زبانی که نیاز به این رفتار داشته باشد، باید کامپایلرش کدی در سازنده‌ی نوع تولید کند که این متد را برای تمام نوع‌های پایه فراخوانی کند. هنگام استفاده از متد **RunClassConstructor** برای فراخوانی سازنده یک نوع، CLR می‌داند آیا سازنده‌ی نوع قبلاً فراخوانی شده است یا نه و اگر شده باشد CLR مجدداً آن را فراخوانی نمی‌کند.

سرانجام، فرض کنید این کد را دارید:

```
internal sealed class SomeType {
    private static Int32 s_x = 5;

    static SomeType() {
        s_x = 10;
    }
}
```

در این حالت، کامپایلر سی‌شارپ یک تک متد سازنده‌ی نوع تولید می‌کند. این سازنده ابتدا **s_x** را به 5 مقداردهی اولیه می‌کند و سپس **s_x** را به 10 مقداردهی اولیه می‌کند. به بیان دیگر، وقتی کامپایلر سی‌شارپ کد IL را برای سازنده‌ی نوع تولید می‌کند، ابتدا کد مورد نیاز برای مقداردهی اولیه فیلدهای استاتیک را تولید کرده و قرار می‌دهد و سپس کد درون متد سازنده‌ی نوع شما را قرار می‌دهد.

مهم گاهی برنامه‌نویسان از من می‌پرسند آیا راهی وجود دارد که هنگام تخلیه (unload) یک نوع، کدی اجرا شود. ابتدا شما باید بدانید که نوع‌ها فقط وقتی **AppDomain** تخلیه می‌شود، تخلیه می‌گردند. وقتی **AppDomain** تخلیه می‌شود، و شی‌ای که نوع را شناسایی می‌کند از دسترس خارج می‌شود و جمع‌آوری‌کننده زباله، حافظه‌ی شی نوع را پس می‌گیرد. این رفتار منجر به آن می‌شود که بسیاری از برنامه‌نویسان یک متد استاتیک **Finalize** به نوع اضافه کنند تا هنگام تخلیه به صورت خودکار اجرا شود. متأسفانه، CLR از متدهای استاتیک **Finalize** پشتیبانی نمی‌کند. هر چند، همه چیز از دست نرفته است. اگر شما می‌خواهید کدی داشته باشید که هنگام تخلیه **AppDomain**، اجرا شود، شما می‌توانید یک کالباک (callback) را با رویداد **DomainUnload** از نوع **System.AppDomain**، ثبت کنید.

کارایی سازنده‌ی نوع

در بخش قبلی، اشاره کردم که فراخوانی یک سازنده‌ی نوع یک حقه بازی است. و من تعدادی از نکات آن را توضیح دادم: کامپایلر JIT تصمیم می‌گیرد که کدی برای فراخوانی تولید کند و CLR مطمئن می‌شود که فراخوانی به آن ترد-امن باشد. همانطور که معلوم است، این تنها آغاز این مسائل است. چیزهای بیشتری درباره‌ی کارایی آن وجود دارد.

همانطور که قبلاً گفته شد، هنگام کامپایل یک متد، کامپایلر JIT تعیین می‌کند که باید کدی برای فراخوانی یک سازنده نوع در متد تولید کند یا نه. اگر کامپایلر JIT تصمیم به تولید فراخوانی بگیرد، باید تصمیم بگیرد این فراخوانی را کجا قرار دهد. دو احتمال وجود دارد:

- کامپایلر JIT می‌تواند فراخوانی را دقیقاً بلافاصله قبل از کدی که اولین نمونه از نوع را می‌سازد یا بلافاصله قبل از کدی که به یک فیلد یا عضو ارث برده نشده از کلاس، دسترسی پیدا می‌کند، قرار دهد. این سمانتیک دقیق **precise** نامیده می‌شود چون CLR فراخوانی به سازنده نوع را دقیقاً در زمان صحیح انجام می‌دهد.
- کامپایلر JIT می‌تواند فراخوانی را جایی قبل از آنکه کد برای اولین بار به یک فیلد استاتیک یا یک متد نمونه یا استاتیک دسترسی پیدا می‌کند یا یک سازنده‌ی نمونه را صدا می‌زند، قرار دهد. این سمانتیک قبل از مقداردهی فیلد **before-field-init** نامیده می‌شود چون CLR تضمین می‌کند که سازنده‌ی استاتیک زمانی قبل از دسترسی به عضو اجرا می‌شود؛ آن می‌تواند خیلی قبل‌تر اجرا شود.

مفهوم قبل از مقداردهی فیلد ترجیح داده می‌شود چون به CLR آزادی بسیار بیشتری می‌دهد که چه زمان می‌تواند سازنده نوع را فراخوانی کند و CLR از این، هر جا ممکن باشد بهره‌بردار تا کدی تولید کند که سریعتر اجرا گردد. برای نمونه، CLR ممکن است زمان‌های متفاوتی برای فراخوانی سازنده نوع انتخاب کند بسته به اینکه نوع در یک AppDomain بارگذاری شده است یا مستقل از دامنه بارگذاری شده یا اینکه کد JIT کامپایل شده یا NGen شده است.

به صورت پیش فرض، کامپایلرهای زبان‌ها تعیین می‌کنند کدام یک از این مفاهیم برای نوعی که شما تعریف می‌کنید بهتر است و CLR را از انتخاب خود با تنظیم پرچم `beforefieldinit` در ردیف مربوطه در جدول متادیتای تعریف نوع، آگاه می‌کنند. در این بخش، من بر آنچه کامپایلر سی‌شارپ انجام می‌دهد و اثر آن بر کارایی برنامه تمرکز می‌کنم. بگذارید با بررسی کد زیر شروع کنیم:

```
using System;
using System.Diagnostics;

////////////////////////////////////

// Since this class doesn't explicitly define a type constructor,
// C# marks the type definition with BeforeFieldInit in the metadata.
internal sealed class BeforeFieldInit {
    public static Int32 s_x = 123;
}

// Since this class does explicitly define a type constructor,
// C# doesn't mark the type definition with BeforeFieldInit in the metadata.
internal sealed class Precise {
    public static Int32 s_x;
    static Precise() { s_x = 123; }
}

////////////////////////////////////

public sealed class Program {
    public static void Main() {
        const Int32 iterations = 1000 * 1000 * 1000;
        PerfTest1(iterations);
        PerfTest2(iterations);
    }

    // when this method is JIT compiled, the type constructors for
    // the BeforeFieldInit and Precise classes HAVE NOT executed yet
    // and therefore, calls to these constructors are embedded in
    // this method's code, making it run slower
    private static void PerfTest1(Int32 iterations) {
        Stopwatch sw = Stopwatch.StartNew();
        for (Int32 x = 0; x < iterations; x++) {
            // The JIT compiler hoists the code to call BeforeFieldInit's
            // type constructor so that it executes before the loop starts
            BeforeFieldInit.s_x = 1;
        }
        Console.WriteLine("PerfTest1: {0} BeforeFieldInit", sw.Elapsed);
    }
}
```

```

sw = Stopwatch.StartNew();
for (Int32 x = 0; x < iterations; x++) {
    // The JIT compiler emits the code to call Precise's
    // type constructor here so that it checks whether it
    // has to call the constructor with each loop iteration
    Precise.s_x = 1;
}
Console.WriteLine("PerfTest1: {0} Precise", sw.Elapsed);
}

// When this method is JIT compiled, the type constructors for
// the BeforeFieldInit and Precise classes HAVE executed
// and therefore, calls to these constructors are NOT embedded
// in this method's code, making it run faster
private static void PerfTest2(Int32 iterations) {
    Stopwatch sw = Stopwatch.StartNew();
    for (Int32 x = 0; x < iterations; x++) {
        BeforeFieldInit.s_x = 1;
    }
    Console.WriteLine("PerfTest2: {0} BeforeFieldInit", sw.Elapsed);

    sw = Stopwatch.StartNew();
    for (Int32 x = 0; x < iterations; x++) {
        Precise.s_x = 1;
    }
    Console.WriteLine("PerfTest2: {0} Precise", sw.Elapsed);
}
}

```

//////////////////////////////////// End of File //////////////////////////////////////

وقتی من کد فوق را ساخته و اجرا می‌کنم، خروجی زیر را می‌گیرم:

```

PerfTest1: 00:00:01.9619358 BeforeFieldInit
PerfTest1: 00:00:06.2374912 Precise
PerfTest2: 00:00:03.1576608 BeforeFieldInit
PerfTest2: 00:00:03.1557822 Precise

```

وقتی کامپایلر سی‌شارپ یک کلاس با فیلدهای استاتیک که از مقداردهی اولیه خطی استفاده کرده است (کلاس **BeforeFieldInit**) را می‌بیند، کامپایلر ورودی جدول تعریف نوع کلاس را با پرچم متادیتای **BeforeFieldInit** تولید می‌کند. وقتی کامپایلر سی‌شارپ یک کلاس با یک سازنده نوع صریح (کلاس **Precise**) را می‌بیند، کامپایلر، ورودی جدول تعریف نوع کلاس را بدون پرچم متادیتای **BeforeFieldInit** تولید می‌کند. منطق پشت این کار اینست: مقداردهی اولیه فیلدهای استاتیک باید قبل از دسترسی به فیلد صورت پذیرد، در حالیکه یک سازنده‌ی نوع صریح می‌تواند شامل کدی باشد که اثرات جانبی داشته باشد، پس این کد باید در زمان دقیق خود اجرا گردد.

همانگونه که در خروجی می‌بینید، این تصمیم با اثر بزرگی بر کارایی برنامه همراه است. وقتی **PerfTest1** اجرا می‌شود، حلقه‌ی بالایی در حدود ۱.۹۶ ثانیه اجرا می‌شود و در مقابل حلقه پایینی در حدود ۶.۲۴ ثانیه اجرا می‌گردد – حلقه پایینی ۳ برابر زمان بیشتر برای اجرا نیاز دارد. وقتی **PerfTest2** اجرا می‌شود، زمان‌ها بسیار به هم نزدیکند چون کامپایلر JIT می‌داند که سازنده‌های نوع قبلاً اجرا شده‌اند، پس کد اصلی تولیدی شامل هیچ گونه فراخوانی به متدهای سازنده‌ی نوع نیست.

خیلی خوب می‌شد اگر سی‌شارپ قابلیت تعیین پرچم **BeforeFieldInit** به صورت صریح در سورس‌کد را ایجاد می‌کرد، به جای آنکه کامپایلر این تصمیم را بر اساس آنکه یک سازنده نوع صریحا یا ضمنی ساخته می‌شود بگیرد. به این روش، برنامه‌نویس کنترل مستقیم بیشتری بر کارایی و مفهوم کدش می‌توانست داشته باشد.

متدهای سربارگذاری عملگرها

برخی زبان‌های برنامه‌نویسی اجازه می‌دهند که یک نوع، تعریف کند چگونه عملگرها باید نمونه‌های نوع را دستکاری کنند. برای نمونه، تعدادی از نوع‌ها (مثل **System.String**، **System.Decimal**، **System.DateTime**) عملگرهای برابری (**==**) و نابرابری (**!=**) را سربارگذاری می‌کنند. CLR چیزی درباره‌ی سربارگذاری عملگرها نمی‌داد چون او حتی نمی‌داند یک عملگر چیست. زبان برنامه‌نویسی شما تعریف می‌کند نماد هر عملگر چه معنی‌ای دارد و چه کدی باید هنگام ظاهر شدن این نمادهای خاص، تولید شود.

برای نمونه، در سی‌شارپ، اعمال نماد **+** به اعداد اصلی باعث می‌شود کامپایلر کدی تولید کند که دو عدد را با هم جمع کند. وقتی نماد **+** به اشیاء **String** اعمال می‌شود، کامپایلر سی‌شارپ کدی تولید می‌کند که دو رشته را به هم متصل می‌کند. برای نابرابری، سی‌شارپ از نماد **!=** استفاده می‌کند درحالی‌که ویژوال بیسیک از نماد **<>** استفاده می‌کند. سرانجام، نماد **^** در سی‌شارپ معنی **OR** انحصاری (**XOR**) می‌دهد ولی در ویژوال بیسیک معنی توان را می‌دهد.

اگرچه CLR چیزی درباره‌ی عملگرها نمی‌داند اما تعیین می‌کند چگونه زبان‌ها باید سربارگذاری شده‌ی عملگرها را به کار برند تا توسط کد نوشته شده در یک زبان برنامه‌نویسی متفاوت به راحتی قابل استفاده باشند. هر زبان برنامه‌نویسی برای خودش تعیین می‌کند که سربارگذاری عملگرها را پشتیبانی کند یا خیر و اگر این کار را انجام دهد، نحو بیان و استفاده از آن‌ها را معلوم می‌کند. تا زمانی که CLR مراقب است، سربارگذاری عملگرها، به سادگی فقط متدهایی هستند.

زبان برنامه‌نویسی شما تعیین می‌کند که آیا شما پشتیبانی از سربارگذاری عملگرها را دارید و نحو آن شبیه چیست. وقتی شما کدتان را کامپایل می‌کنید، کامپایلر متدی تولید می‌کند که رفتار عملگر را تشخیص می‌دهد. مشخصات CLR اجبار می‌کنند که متدهای سربارگذاری عملگرها باید متدهای **public** و **static** باشند. به علاوه، سی‌شارپ (و بسیاری زبان‌های دیگر) نیاز دارند که حداقل یکی از پارامترهای متد عملگر از همان نوعی باشد که متد عملگر با آن تعریف شده است.

علت این محدودیت اینست که کامپایلر سی‌شارپ را قادر می‌سازد که در مدت زمان معقولانه‌ای به دنبال متد عملگر بگردد.

نمونه‌ای از یک متد سربارگذاری عملگر تعریف شده در یک کلاس سی‌شارپ را می‌بینید:

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
}
```

کامپایلر یک ورودی متادیتای تعریف متد برای یک متد به نام **op_Addition** تولید می‌کند، ورودی تعریف متد دارای پرچم **specialname** که فعال شده است می‌باشد به این معنی که این یک متد "خاص" است. وقتی کامپایلرهای زبان‌ها (شامل کامپایلر سی‌شارپ)، یک عملگر **+** در سورس‌کد می‌بینید، نگاه می‌کنند آیا نوع یکی از عملوندها یک متد **specialname** به نام **op_Addition** که پارامترهایش با نوع عملوندها سازگار باشد تعریف کرده است یا خیر. اگر این متد وجود داشته باشد، کامپایلر کدی برای فراخوانی این متد تولید می‌کند. اگر چنین متدی وجود نداشته باشد، یک خطای کامپایلر رخ می‌دهد. جدول ۸-۱ و ۸-۲ مجموعه عملگرهای یکانی و باینری که سی‌شارپ، سربارگذاری آن‌ها را پشتیبانی می‌کند، نشان می‌دهد. نماد آن‌ها و نام متد منطبق بر مشخصات مشترک زبان (CLS) که کامپایلر تولید می‌کند نیز نشان داده شده‌اند. من سه ستون جدول را در بخش بعدی توضیح می‌دهم.

جدول ۸-۱ عملگرهای یکانی سی‌شارپ و نام‌ها متد متناظر منطبق بر CLS

نماد عملگر سی‌شارپ	نام خاص متد	نام پیشنهادی متد منطبق بر CLS
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
!	op_LogicalNot	Not
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement

IsTrue { get; }	op_True	(وجود ندارد)
IsFalse { get; }	op_False	(وجود ندارد)

جدول ۲-۸ عملگرهای باینری سی شارپ و نام متد های متناظر منطبق بر CLS

نام عملگر سی شارپ	نام خاص متد	نام پیشنهادی متد منطبق بر CLS
+	op_Addition	Add
-	op_Subtract	Subtract
*	op_Multiply	Multiply
/	op_Division	Division
%	op_Modulus	Mod
&	op_BitwiseAnd	BitwiseAnd
	op_ExclusiveOr	Xor
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
==	op_Equality	Equals
!=	op_Inequality	Compare
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare

مشخصات CLR تعداد زیادی عملگرهای اضافی دیگر که قابل سربرگذاری هستند را تعریف می کند اما سی شارپ این عملگرهای اضافی را پشتیبانی نمی کند. بنابراین، آن ها زیاد استفاده نمی شوند و من آن ها را اینجا نمی آورم. اگر شما می خواهید لیست کامل را ببینید، لطفاً به مشخصات ECMA Common Language Infrastructure (CLI)، قسمت اول، "مفاهیم و معماری" Concepts And Architecture، بخش های ۱۰.۳.۱ (عملگرهای یکانی) و ۱۰.۳.۲ (عملگرهای باینری) مراجعه کنید.

بله اگر شما نوع های عددی اصلی (`UInt32`، `Int64`، `Int32` و غیره) را در کتابخانه کلاس فریمورک (FCL) بررسی کنید، خواهید دید که آن ها هیچ متد سربرگذاری عملگری تعریف نکرده اند. علت آنکه این کار را نکرده اند اینست که کامپایلرها به عملیات بر روی نوع های اصلی به صورت ویژه نگاه می کنند و دستورات IL ای تولید می کنند که مستقیماً نمونه های این نوع ها را دستکاری می کند. اگر نوع ها می خواستند متدهایی ارائه کنند و اگر کامپایلرها کدی برای فرخوانی این متدها تولید می کردند، هزینه ی زمان اجرایی با فراخوانی متد همراه می شد. افزون بر آن، متد سرانجام می بایست تعدادی دستورات IL برای اجرای عملیات مورد انتظار اجرا کند. این علت آنست که چرا نوع های اصلی FCL هیچ متد سربرگذاری عملگری تعریف نمی کنند. معنی این برای شما این است: اگر زبان برنامه نویسی مورد استفاده شما یکی از نوع های اصلی FCL را پشتیبانی نمی کند، شما قادر نخواهید بود هیچ گونه عملی بر روی نمونه های آن نوع انجام دهید.

عملگرها و تقابل زبان برنامه نویسی

سربرگذاری عملگرها می تواند یک ابزار بسیار مفید باشد که به برنامه نویسان اجازه بیان افکارشان با کد خلاصه ای را می دهند. اما، همه ی زبان های برنامه نویسی از سربرگذاری عملگرها پشتیبانی نمی کنند. وقتی از زبانی استفاده می کنید که از سربرگذاری عملگرها پشتیبانی نمی کند، زبان نمی داند چگونه عملگر + را تفسیر کند (مگر آنکه نوع در آن زبان، یک نوع اصلی باشد) و کامپایلر یک خطا تولید می کند. وقتی از زبان های بدون پشتیبانی سربرگذاری عملگرها، استفاده می کنید، زبان باید به شما اجازه ی فراخوانی متد `op_*` (مثل `op_Addition`) را مستقیماً بدهد.

اگر از زبانی استفاده می‌کنید که تعریف سربرگذاری عملگر + در نوع را پشتیبانی نمی‌کند، واضح است که این نوع هنوز می‌تواند یک متد **op_Addition** ارائه کند. شاید از سی‌شارپ انتظار داشته باشید که شما بتوانید متد **op_Addition** را برای عملگر + فراخوانی کنید، اما نمی‌توانید. وقتی کامپایلر سی-شارپ عملگر + را پیدا می‌کند، به دنبال یک متد **op_Addition** که پرچم متادیتای **specialname** آن فعال باشد، می‌گردد تا کامپایلر مطمئن باشد که متد **op_Addition** به عنوان یک متد سربرگذاری عملگر وجود دارد. چون متد **op_Addition** توسط زبانی که سربرگذاری عملگرها را پشتیبانی نمی‌کند، تولید شده، متد، پرچم **specialname** را همراه خود نخواهد داشت و کامپایلر سی‌شارپ یک خطای کامپایلر تولید می‌کند. البته، هر کدی در هر زبانی می‌تواند صریحاً متدی که فقط نامش **op_Addition** است را فراخوانی کند، اما کامپایلر، استفاده از عملگر + را به فراخوانی این متد تفسیر نخواهد کرد.

نظر جف پیرامون قوانین نام گذاری متد عملگر توسط مایکروسافت

من مطمئنم که این قوانین درباره‌ی اینکه چه زمان شما می‌توانید و چه زمان نمی‌توانید یک متد سربرگذاری عملگر را فراخوانی کنید، بسیار گیج‌کننده و پیچیده است. اگر کامپایلرهایی که سربرگذاری عملگر را پشتیبانی می‌کردند، پرچم متادیتای **specialname** را تولید نمی‌کردند، قوانین بسیار ساده‌تر می‌شد و برنامه‌نویسان آسانتر با نوع‌هایی که متدهای سربرگذاری عملگر را ارائه می‌کنند، کار می‌کردند. زبان‌هایی که از سربرگذاری عملگر پشتیبانی می‌کنند از نحو نماد عملگر پشتیبانی می‌کنند و تمام زبان‌ها می‌توانستند فراخوانی صریح متدهای **op_** را پشتیبانی کنند. من دلیلی پیدا نمی‌کنم که چرا مایکروسافت این را بسیار سخت کرده است و امیدوارم در نسخه‌های آتی کامپایلرهایشان این قوانین را آسانتر کنند.

برای یک نوع که متدهای سربرگذاری عملگر را تعریف می‌کند، مایکروسافت توصیه می‌کند که نوع، متدهای استاتیک عمومی مورد پسند تری که در درون، متدهای سربرگذاری عملگر را فراخوانی می‌کنند، نیز تعریف نمایند. برای نمونه، یک متد با نام مورد پسند **Add** باید برای یک نوع که متد **op_Addition** را سربرگذاری می‌کند، تعریف شود. ستون در جدول ۱-۸ و ۲-۸، این نام‌های مورد پسند را برای هر عملگر لیست می‌کند. پس نوع **Complex** که قبلاً نشان داده شد باید بدین گونه تعریف شود:

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
    public static Complex Add(Complex c1, Complex c2) { return(c1 + c2); }
}
```

مطمئناً، کد نوشته شده در هر زبان می‌تواند هر یک از این متدهای عملگر مثل **Add** را فراخوانی کند. راهنمایی مایکروسافت بر تعریف این نام‌های مورد پسند قضیه را بیش از پیش پیچیده می‌کند. من فکر می‌کنم که این پیچیدگی نیاز نیست و فراخوانی این متدها با نام مورد پسند منجر به ضربه به کارایی می‌شود مگر آنکه کاپایلر JIT قادر به خطی کردن کد درون این متدهای با نام مورد پسند، باشد. خطی کردن کد، باعث می‌شود کامپایلر JIT کد را بهینه کرده، فراخوانی‌های اضافی متدها را حذف کرده و سرعت اجرا را افزایش دهد.

نکته برای یک نمونه از نوع که عملگرها را سربرگذاری کرده و از متدهای با نام مورد پسند طبق توصیه مایکروسافت استفاده می‌کند، کلاس **System.Decimal** در FCL را ببینید.

متدهای عملگر تبدیل

گاهی شما نیاز به تبدیل یک شی از یک نوع به یک شی از نوع دیگری دارید. برای نمونه من مطمئنم جایی در زندگی خود، نیاز به تبدیل یک **Byte** به یک **Int32** دارید. وقتی نوع مبدا و نوع مقصد، نوع‌های اصلی کامپایلر باشند، کامپایلر می‌داند چگونه کد لازم برای تبدیل این اشیاء را تولید کند.

اگر نوع مبدا و مقصد نوع‌های اصلی نباشند، کامپایلر کدی تولید می‌کند که CLR عمل تبدیل را انجام دهد (**cast**). در این حالت، CLR فقط بررسی می‌کند آیا نوع شی مبدا از همان نوع شی مقصد (یا مشتق شده از نوع مقصد) باشد. هرچند، گاهی طبیعی است که شما می‌خواهید یک شی از یک نوع را به نوعی کاملاً متفاوت تبدیل کنید. برای نمونه کلاس **System.Xml.Linq.XElement** به شما اجازه می‌دهد یک عنصر XML را به یک **Boolean**، **Int32**، **Int64**، **String**، **Decimal**، **Double**، **Single**، **DateTime**، **DateTimeOffset**، **TimeSpan**، **Guid** و یا معادل تهی پذیر هر یک از این نوع‌ها (به جز **String**) تبدیل کنید. شما همچنین می‌توانید تصور کنید که FCL یک نوع داده‌ای **Rational** دارد و ممکن است تبدیل یک شی **Int32** یا یک شی **Single** به یک شی **Rational** ساده باشد. علاوه بر آن، خیلی خوب است اگر بتوان یک شی **Rational** را به یک شی **Int32** یا یک شی **Single** تبدیل نمود.

برای انجام این تبدیل‌ها، نوع **Rational** باید سازنده‌های عمومی تعریف کند که یک تک پارامتر می‌پذیرند: یک نمونه از نوعی که شما از آن می‌خواهید تبدیل کنید. شما همچنین باید متدهای نمونه عمومی **ToXxx** که پارامتری نمی‌گیرند (درست شبیه به متد بسیار معروف **ToString**) تعریف کنید. هر متد یک نمونه از نوع تعریف کننده را به نوع **Xxx** تبدیل می‌کند. نحوه‌ی صحیح متدها و سازنده‌های تبدیل برای نوع **Rational** در زیر آمده است:

```
public sealed class Rational {
    // Constructs a Rational from an Int32
    public Rational(Int32 num) { ... }

    // Constructs a Rational from a Single
    public Rational(Single num) { ... }

    // Convert a Rational to an Int32
    public Int32 ToInt32() { ... }

    // Convert a Rational to a Single
    public Single ToSingle() { ... }
}
```

با فراخوانی این متدها و سازنده‌ها، برنامه‌نویس در هر زبان برنامه‌نویسی می‌تواند یک **Int32** یا یک شی **Single** را به یک شی **Rational** تبدیل و یک شی **Rational** را به یک شی **Int32** یا یک شی **Single** تبدیل نماید. این قابلیت بسیار پرکاربرد است و هنگام طراحی یک نوع، باید جدا فکر کنید کدام متدها و سازنده‌های تبدیل برای نوع شما مفید هستند.

در بخش قبلی، بحث کردم چگونه بعضی از زبان‌های برنامه‌نویسی، سربرگذاری عملگرها را ارائه می‌کنند. عملگرهای تبدیل، متدهایی هستند که یک شی را از یک نوع به نوعی دیگر تبدیل می‌کنند. شما یک متد عملگر تبدیل را با نحو خاصی تعریف می‌کنید. مشخصات CLR دیکته می‌کند که متدهای سربرگذاری شده‌ی تبدیل باید متدهای **public** و **static** باشند. به علاوه، سی‌شارپ (و بسیاری زبان‌های برنامه‌نویسی دیگر) نیاز دارند که پارامتر یا نوع برگشتی نوع، باید از همان نوعی باشد که متد تبدیل، درون آن تعریف شده است. علت این محدودیت این است که کامپایلر سی‌شارپ را قادر می‌سازد در یک مدت زمان معقولانه جهت اتصال متد عملگر به دنبال آن بگردد. کد زیر چهار متد عملگر تبدیل را به نوع **Rational** می‌افزاید.

```
public sealed class Rational {
    // Constructs a Rational from an Int32
    public Rational(Int32 num) { ... }

    // Constructs a Rational from a Single
    public Rational(Single num) { ... }

    // Convert a Rational to an Int32
    public Int32 ToInt32() { ... }

    // Convert a Rational to a Single
    public Single ToSingle() { ... }
}
```

```
// Implicitly constructs and returns a Rational from an Int32
public static implicit operator Rational(Int32 num) {
    return new Rational(num);
}

// Implicitly constructs and returns a Rational from a Single
public static implicit operator Rational(Single num) {
    return new Rational(num);
}

// Explicitly returns an Int32 from a Rational
public static explicit operator Int32(Rational r) {
    return r.ToInt32();
} // Explicitly returns a Single from a Rational

public static explicit operator Single(Rational r) {
    return r.ToSingle();
}
}

برای متدهای عملگر تبدیل، باید تعیین کنید که کامپایلر می‌تواند به صورت ضمنی کدی برای تبدیل فراخوانی یک متد عملگر تبدیل، تولید کند یا سورس‌کد باید صریحا تعیین کند چه وقت کامپایلر کدی برای فراخوانی یک متد عملگر تبدیل، تولید کند. در سی‌شارپ شما از کلمه کلیدی implicit استفاده می‌کنید تا تعیین کنید که برای تولید کد برای فراخوانی متد، نیاز به یک تبدیل صریح در سورس‌کد نیست. کلمه کلیدی explicit به کامپایلر اجازه می‌دهد که متد را فقط وقتی یک تبدیل صریح در سورس‌کد هست، فراخوانی کند.

پس از کلمه کلیدی implicit یا explicit به کامپایلر می‌گویید که متد یک عملگر تبدیل است یا تعیین کلمه کلیدی operator. پس از کلمه کلیدی operator، نوعی که شی به آن تبدیل می‌شود را تعیین می‌کنید و درون پرانتزها، نوعی که شی از آن تبدیل می‌شود را تعیین می‌کنید.

تعریف عملگرهای تبدیل در نوع Rational قبلی اجازه می‌دهد کدی شبیه به زیر (در سی‌شارپ) بنویسید:
```

```
public sealed class Program {
    public static void Main() {
        Rational r1 = 5;           // Implicit cast from Int32 to Rational
        Rational r2 = 2.5F;        // Implicit cast from Single to Rational

        Int32 x = (Int32) r1;      // Explicit cast from Rational to Int32
        Single s = (Single) r2;    // Explicit cast from Rational to Single
    }
}
}
```

در پشت پرده، کامپایلر سی‌شارپ، **Cast** ها (تبدیل نوع‌ها) را در کد یافته و کد **IL** ای که متدهای عملگر تبدیل تعریف شده توسط نوع **Rational** فراخوانی کند تولید می‌نماید. اما نام این متدها چیست؟ خوب، کامپایلر نوع **Rational** و بررسی متادیتای آن نشان می‌دهد که کامپایلر، یک متد برای هر عملگر تبدیل تعریف شده، تولید می‌کند. برای نوع **Rational**، متادیتا برای چهار متد عملگر تبدیل شبیه به این است:

```
public static Rational op_Implicit(Int32 num)
public static Rational op_Implicit(Single num)
public static Int32 op_Explicit(Rational r)
public static Single op_Explicit(Rational r)
```

همانگونه که می‌بینید، متدهایی که می‌توانند یک شی را از نوعی به نوع دیگر تبدیل کنند، همواره نام **op_Implicit** یا **op_Explicit** دارند. شما باید فقط وقتی یک عملگر تبدیل ضمنی تعریف کنید که دقت یا بزرگی هنگام تبدیل کم نشود مثل تبدیل یک **Int32** به یک **Rational**. هر چند، شما باید یک عملگر تبدیل صریح را در صورتی که دقت یا بزرگی هنگام تبدیل کم می‌شود، تعریف کنید مثل تبدیل یک شی **Rational** به یک **Int32**. اگر یک

تبدیل صریح شکست بخورد، شما باید این را بدین گونه نشان دهید که عملگر تبدیل صریح شما یک **OverflowException** یا یک **InvalidOperationException** تولید کند.

نکته دو متد **op_Explicit** در مثال قبل، یک نوع پارامتر می‌گیرند، یک **Rational**. اما متدها فقط از لحاظ نوع برگشتی متفاوتند، یک **Int32** و یک **Single**. این مثالی از دو متد است که تنها از لحاظ نوع برگشتی متفاوتند. CLR کاملاً از این پشتیبانی می‌کند که یک نوع، چندین متد که فقط از لحاظ نوع برگشتی متفاوتند را تعریف کند. همانطور که احتمالاً آگاهی، **C++**، **C#**، **Visual Basic** و **Java** همه نمونه‌هایی از زبان‌هایی هستند که تعریف چندین متد که تنها از لحاظ نوع برگشتی فرق دارند را پشتیبانی نمی‌کنند. تعداد کمی از زبان‌ها (مثل زبان اسمبلی IL) به برنامه‌نویس اجازه می‌دهند که صریحاً انتخاب کند کدامیک از متدها را فراخوانی نماید. البته، برنامه‌نویسان زبان اسمبلی IL نباید از این قابلیت استفاده کنند چون متدهایی که تعریف می‌کنند از زبان‌های برنامه‌نویسی دیگر قابل فراخوانی نیستند. اگرچه سی‌شارپ این قابلیت را در اختیار برنامه‌نویسان سی‌شارپ قرار نمی‌دهد، هنگامی که یک نوع متدهای عملگر تبدیل را تعریف می‌کند، کامپایلر در درون خود از این قابلیت بهره می‌برد.

سی‌شارپ از عملگرهای تبدیل پشتیبانی کامل می‌کند. وقتی کدی را می‌بیند که در آن شما از یک شی از یک نوع استفاده می‌کنید در حالیکه یک شی از یک نوع دیگر مورد انتظار است، کامپایلر به دنبال یک متد عملگر تبدیل ضمنی که قادر به این تبدیل باشد می‌گردد و کدی برای فراخوانی آن متد تولید می‌کند. اگر یک متد عملگر تبدیل ضمنی موجود باشد، کامپایلر در کد IL حاصل، یک فراخوانی به آن متد را قرار می‌دهد. اگر کامپایلر ببیند سورس کد صریحاً یک شی از یک نوع را به نوعی دیگر تبدیل می‌کند، کامپایلر به دنبال یک متد عملگر تبدیل ضمنی یا صریح می‌گردد. اگر یکی از آنها موجود باشند، کامپایلر کدی برای فراخوانی آن تولید می‌کند. اگر کامپایلر، متد عملگر تبدیل مناسب را پیدا نکند، خطایی تولید کرده و کد را کامپایل نمی‌کند.

نکته سی‌شارپ هنگام استفاده از یک عبارت تبدیل (**cast**) کدی برای فراخوانی عملگرهای تبدیل تولید می‌کند، این متدها هرگز هنگام استفاده از عملگرهای **is** یا **as** سی‌شارپ فراخوانی نمی‌شوند.

برای آنکه کاملاً متدهای سربارگذاری عملگر و متدهای عملگر تبدیل را درک کنید، قویاً شما را به بررسی نوع **System.Decimal** توصیه می‌کنم. **Decimal** چندین سازنده تعریف می‌کند که اجازه می‌دهند اشیاء را از نوع‌های مختلف به یک **Decimal** تبدیل کنید. آن همچنین، چندین متد **ToXXX** ارائه می‌کند که به شما اجازه می‌دهد یک شی **Decimal** را به نوعی دیگر تبدیل نمایید. سرانجام، این نوع چندین متد عملگر تبدیل و سربارگذاری عملگر نیز تعریف می‌کند.

متدهای گسترشی

بهترین راه درک متدهای گسترشی **extension methods** سی‌شارپ با یک مثال است. در بخش "اعضای **StringBuilder**" در فصل ۱۴ "کاراکترها، رشته‌ها و کار با متن" اشاره می‌کنم چگونه کلاس **StringBuilder** متدهای کمتری از کلاس **String** برای کار با رشته‌ها ارائه می‌کند و عجیب اینست که کلاس **StringBuilder**، روش ترجیحی برای دستکاری یک رشته است، چون یک رشته، تغییر ناپذیر است.

پس، بگوییم که شما دوست دارید برخی از این متدهای از دست داده را برای کار روی یک **StringBuilder** تعریف کنید. برای نمونه شاید شما بخواهید متد **IndexOf** را به طریق زیر تعریف کنید:

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

حال که این متد را تعریف کردید، می‌توانید آن را به طریق زیر استفاده کنید:

```
StringBuilder sb = new StringBuilder("Hello. My name is Jeff."); // The initial string
```

```
// Change period to exclamation and get # characters in 1st sentence (5).
```

```
Int32 index = StringBuilderExtensions.IndexOf(sb.Replace('.', '!'), '!');
```

این کد به خوبی کار می‌کند اما از دید یک برنامه‌نویس ایده‌آل نیست. اولین مشکل اینست که برنامه‌نویس که بخواهد اندیس یک کاراکتر را درون یک **StringBuilder** بدست آورد، باید بداند که کلاس **StringBuilderExtensions** اصلاً وجود دارد، مشکل دوم اینست که ترتیب عملیاتی که بر شی **StringBuilder** اعمال می‌شود را نشان نمی‌دهد و کد را برای نوشتن، خواندن و نگهداری مشکل می‌کند. برنامه‌نویس می‌خواهد اول **Replace** و بعد **IndexOf** را فراخوانی نماید. اما وقتی شما آخرین خط کد را از چپ به راست می‌خوانید، **IndexOf** اول ظاهر شده و **Replace** دوم آمده است. البته، شما می‌توانید این مشکل را کمتر کرده و کد را شبیه زیر به گونه‌ای قابل درک تر بنویسید:

```
// First, change period to exclamation mark
```

```
sb.Replace('.', '!');
```

```
// Now, get # characters in 1st sentence (5)
```

```
Int32 index = StringBuilderExtensions.IndexOf(sb, '!');
```

اما، یک مشکل سوم با هر دو نسخه‌ی کد وجود دارد که بر درک رفتار کد اثرگذار است. استفاده از **StringBuilderExtensions** غلبه دارد و ذهن برنامه‌نویس را از عملی که در حال انجام است (**IndexOf**) پرت می‌کند. اگر کلاس **StringBuilder** خودش متد **IndexOf** را تعریف کرده بود، آنگاه ما می‌توانستیم کد فوق را اینگونه بنویسیم:

```
// Change period to exclamation and get # characters in 1st sentence (5).
```

```
Int32 index = sb.Replace('.', '!').IndexOf('!');
```

نگاه کنید که این کد چقدر برای نگهداری عالی تر است. در شی **StringBuilder**، می‌خواهیم نقطه را با علامت تعجب جایگزین کنیم و سپس اندیس علامت تعجب را بیابیم.

حال من می‌توانم توضیح دهم که قابلیت متدهای گسترشی سی‌شارپ چه می‌کند. آن، به شما اجازه می‌دهد یک متد استاتیک تعریف کنید که بتوانید با نحو متد نمونه آن را فراخوانی کنید. یا به بیان دیگر، اکنون ما می‌توانیم متد **IndexOf** خودمان را تعریف کنیم و سه مشکل بالا کلاً از بین برود. برای آنکه متد **IndexOf** را به یک متد گسترشی تبدیل کنیم، به سادگی کلمه کلیدی **this** را قبل از اولین آرگومان آن اضافه می‌کنیم:

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(this StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

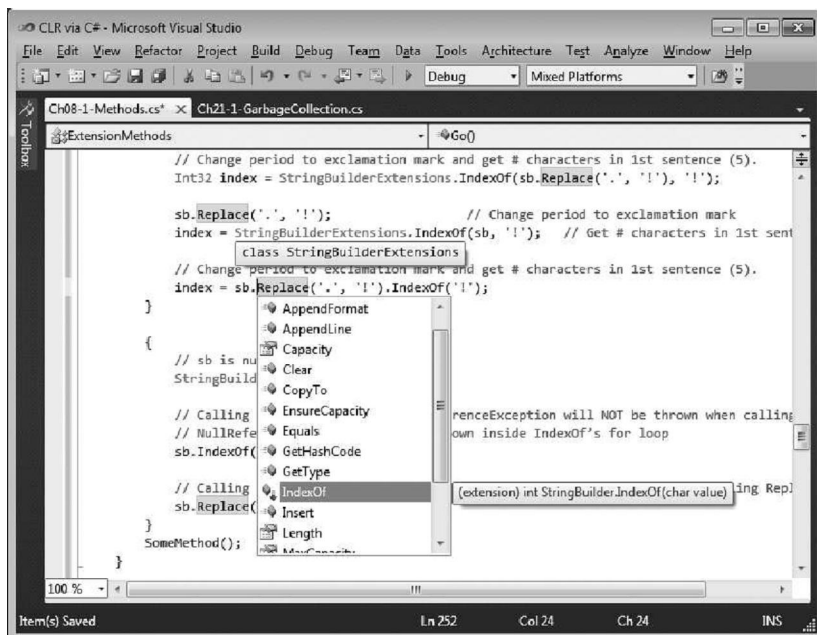
حال، وقتی کامپایلر کدی شبیه به این ببیند:

```
Int32 index = sb.IndexOf('x');
```

کامپایلر اول بررسی می‌کند آیا کلاس **StringBuilder** یا یکی از کلاس‌های پایه‌اش متد نمونه‌ای به نام **IndexOf** که یک پارامتر **Char** بگیرد، دارند. اگر یک متد نمونه موجود باشد، کامپایلر کد **IL** ای برای فراخوانی آن تولید می‌کند. اگر هیچ متد نمونه‌ای یافت نشد، کامپایلر به دنبال هر کلاس استاتیکی که متد استاتیکی به نام **IndexOf** که اولین پارامترش با نوع عبارتی که متد را فراخوانی کرده، یکی باشد، می‌گردد. همچنین این نوع باید با کلمه کلیدی **this** علامت زده شده باشد. در این مثال، عبارت **sb** است، که از نوع **StringBuilder** می‌باشد. در این مورد، کامپایلر خصوصاً دنبال یک متد **IndexOf** که دو پارامتر: یک **StringBuilder** (علامت زده شده با کلمه کلیدی **this**) و یک **Char** دریافت کند، می‌گردد. کامپایلر، متد **IndexOf** ما را یافته و کدی برای فراخوانی متد استاتیک ما تولید می‌کند.

خوب – حال این نشان می‌دهد چگونه کامپایلر، دو مسئله‌ی آخر درباره‌ی قابل درک بودن کد که قبلاً اشاره کردم را حل می‌کند. هرچند، هنوز اولین مشکل را مطرح نکرده‌ام: چگونه یک برنامه‌نویس بداند که اصلاً یک متد **IndexOf** وجود دارد که می‌تواند بر یک شی **StringBuilder** عمل کند؟ جواب این سوال در قابلیت **IntelliSense** ویژوال استودیو است. در ویرایشگر، وقتی شما یک نقطه تایپ می‌کنید، پنجره‌ی **IntelliSense** ویژوال استودیو باز شده و لیست متدهای نمونه‌ی موجود را نمایش می‌دهد. خوب، این پنجره **IntelliSense** به شما متدهای گسترشی برای نوع عبارتی که نقطه را پس از آن قرار داده اید، را نیز نمایش می‌دهد. شکل ۸-۱ پنجره‌ی **IntelliSense** ویژوال استودیو را نشان می‌دهد. آی‌کون یک متد گسترشی همراه با یک فلش رو به

پایین است، و پنجره‌ی **tooltip** کنار متد بیان می‌کند که متد واقعا یک متد گسترشی است. این واقعا عالی است چون اکنون شما متدهای خودتان را برای کار روی نوع‌های مختلف از اشیاء تعریف می‌کنید و برنامه‌های دیگر هنگام استفاده از این نوع‌ها، براحتی متدهای شما را پیدا می‌کنند.



شکل ۸-۱ پنجره‌ی **IntelliSense** ویزوال استودیو در حال نمایش یک متد گسترشی

قوانین و راهنمایی‌ها

چندین قانون و راهنمایی اضافی وجود دارد که پیرامون متدهای گسترشی باید بدانید:

- سی‌شارپ تنها از متدهای گسترشی پشتیبانی می‌کند، ویژگی‌های گسترشی، رویدادهای گسترشی، عملگرهای گسترشی و غیره را ارائه نمی‌کند.
- متدهای گسترشی (متدهایی با **this** قبل از اولین آرگومان‌شان) باید در کلاس‌های غیرجنریک و استاتیک تعریف شوند. اما هیچ محدودیتی بر روی نام کلاس وجود ندارد، شما می‌توانید هر اسمی روی آن بگذارید. البته، یک متد گسترشی باید حداقل یک پارامتر داشته باشد و فقط اولین پارامتر می‌تواند با کلمه کلیدی **this** علامت زده شود.
- کامپایلر سی‌شارپ تنها به دنبال متدهای گسترشی تعریف شده در کلاس‌های استاتیکی می‌گردد که این کلاس‌ها در میدان فایل تعریف شده باشند. به بیان دیگر، اگر شما کلاس استاتیک را درون کلاس دیگری تعریف کنید، کامپایلر سی‌شارپ خطای زیر را نشان می‌دهد:

“error CS1109: Extension method must be defined in a top-level static class; StringBuilderExtensions is a nested class.”

- چون کلاس‌های استاتیک، هر نامی که بخواهید می‌توانند داشته باشند، این، وقت کامپایلر سی‌شارپ را می‌گیرد که برای یافتن متدهای گسترشی باید تمام کلاس‌های استاتیک در دامنه فایل را نگاه کرده و متدهای استاتیک آن‌ها را بررسی کند. جهت بهبود سرعت و خودداری از توجه به متدهای گسترشی‌ای که شما نمی‌خواهید، کامپایلر سی‌شارپ نیاز دارد که شما متدهای گسترشی را "وارد کنید" (**import**). برای نمونه، اگر کسی یک کلاس **StringBuilderExtensions** در یک فضای نام **Wintellect** تعریف کرده است، آنگاه یک برنامه‌نویس که می‌خواهد به متدهای گسترشی کلاس دسترسی داشته باشد باید **using Wintellect** را بالای فایل سورس کدش قرار دهد.
- ممکن است چندین کلاس استاتیک، متد گسترشی یکسانی را تعریف کنند. اگر کامپایلر ببیند دو یا بیشتر متد گسترشی یکسان وجود دارد، خطای زیر را اعلام می‌کند:

“error CS0121: The call is ambiguous between the following methods or properties: ‘StringBuilderExtensions.IndexOf(String, Char)’ and ‘AnotherStringBuilderExtensions.IndexOf(String, Char)’.”

برای رفع این خطا، شما باید سورس کدتان را تغییر دهید. خصوصاً، شما دیگر نمی‌توانید از نحو متد نمونه برای فراخوانی این متد استاتیک استفاده کنید، به جای آن شما باید از نحو متد استاتیک استفاده کرده جاییکه صریحاً نام کلاس استاتیک را تعیین کرده تا به کامپایلر بگویید کدام متد را می‌خواهید فراخوانی کنید.

- شما باید از این ویژگی کم استفاده کنید، چون همه‌ی برنامه‌نویسان با این ویژگی آشنا نیستند. برای نمونه، وقتی شما یک نوع را با یک متد گسترشی، گسترش می‌دهید، شما در واقع نوع‌های مشتق شده را نیز با این متد گسترش می‌دهید. پس، شما نباید یک متد گسترشی که اولین پارامترش **System.Object** است تعریف کنید، چرا که این متد برای تمام انواع عبارت‌ها قابل فراخوانی است و واقعا پنجره IntelliSense ویژوال استودیو را شلوغ می‌کند.
- یک مسئله احتمالی نسخه‌بندی پیرامون متدهای گسترشی وجود دارد. اگر در آینده، مایکروسافت، یک متد نمونه **IndexOf** به کلاس **StringBuilder** به همان گونه‌ای که متد شما فراخوانی می‌شود اضافه کند، آنگاه وقتی من کدم را کامپایل مجدد کنم، کامپایلر به جای متد استاتیک **IndexOf** من، به متد نمونه **IndexOf** مایکروسافت متصل خواهد شد. به همین خاطر، برنامه من با رفتاری متفاوت مواجه خواهد شد. این مسئله نسخه‌بندی دلیل دیگری است که چرا از این ویژگی باید کم استفاده شود.

گسترش نوع های مختلف با متدهای گسترشی

در این فصل، نشان دادم چگونه یک متد گسترشی برای یک کلاس **StringBuilder** تعریف کنید. می‌خواهم اشاره کنم که چون یک متد گسترشی در واقع فراخوانی یک متد استاتیک است، CLR کدی تولید نمی‌کند که مطمئن شود مقدار عبارت مورد استفاده برای فراخوانی، **null** نباشد:

```
// sb is null
StringBuilder sb = null;

// Calling extension method: NullReferenceException will NOT be thrown when calling IndexOf
// NullReferenceException will be thrown inside IndexOf's for loop
sb.IndexOf('X');
```

// Calling instance method: NullReferenceException WILL be thrown when calling Replace
sb.Replace('.', '!');

من همچنین دوست دارم اشاره کنم که شما می‌توانید متدهای گسترشی برای نوع‌های رابط را همانند کد زیر تعریف کنید:

```
public static void ShowItems<T>(this IEnumerable<T> collection) {
    foreach (var item in collection)
        Console.WriteLine(item);
}

متد گسترشی فوق می‌تواند توسط هر عبارتی که منجر به نوعی شود که رابط IEnumerable<T> را پیاده‌سازی می‌کند، استفاده شود:
```

```
public static void Main() {
    // Shows each Char on a separate line in the console
    "Grant".ShowItems();

    // Shows each String on a separate line in the console
    new[] { "Jeff", "Kristin" }.ShowItems();

    // Shows each Int32 value on a separate line in the console
    new List<Int32>() { 1, 2, 3 }.ShowItems();
}
```

مهم متدهای گسترشی اساس تکنولوژی زبان یکپارچه پرس و جوی مایکروسافت Language Integrated Query (LINQ) است. برای یک نمونه عالی از کلاسی که متدهای گسترشی فراوانی ارائه می‌کند، کلاس استاتیک **System.Linq.Enumerable** و تمام متدهای گسترشی استاتیک آن را در SDK دات‌نت ببینید. هر متد گسترشی در این کلاس، رابط **IEnumerable** یا **IEnumerable<T>** را گسترش می‌دهد.

شما می‌توانید متدهای گسترشی را برای نوع‌های نماینده (delegate) نیز تعریف کنید. برای یک نمونه از این نوع، به صفحه ۲۷۸ در فصل ۱۱ "رویدادها" بروید. شما همچنین می‌توانید متدهای گسترشی را به نوع‌های شمارشی اضافه کنید. من یک نمونه از این را در بخش "افزودن متدها به نوع‌های شمارشی" در فصل ۱۵ "نوع‌های شمارشی و پرچم‌های بیتی" نشان می‌دهم.

و آخرین نکته اینکه می‌خواهم اشاره کنم کامپایلر سی‌شارپ به شما اجازه می‌دهد یک نماینده (فصل ۱۷ "نماینده‌ها" را ببینید) تعریف کنید که به یک متد گسترشی روی یک شی اشاره کند:

```
public static void Main () {
    // Create an Action delegate that refers to the static ShowItems extension method
    // and has the first argument initialized to reference the "Jeff" string.
    Action a = "Jeff".ShowItems;
    .
    .
    .
    // Invoke the delegate which calls ShowItems passing it a reference
    // to the "Jeff" string.
    a();
}
```

در کد فوق، کامپایلر سی‌شارپ کد IL ای تولید می‌کند که یک نماینده **Action** درست می‌کند. هنگام ساخت یک نماینده، به سازنده، متدی که باید فراخوانی شود و همچنین یک اشاره‌گر به یک شی که باید به پارامتر مخفی متد، **this**، ارسال گردد، ارسال می‌شوند. در حالت عادی، وقتی شما یک نماینده که به یک متد استاتیک اشاره دارد را می‌سازید، ارجاع شی **null** است چون متدهای استاتیک پارامتر **this** ندارند. هر چند، در این مثال، کامپایلر سی‌شارپ کد خاصی تولید می‌کند تا نماینده‌ای بسازد که به یک متد استاتیک (**ShowItems**) اشاره می‌کند و شی هدف از متد استاتیک، ارجاعی به رشته‌ی "Jeff" است. بعداً، وقتی نماینده درخواست می‌شود، CLR متد استاتیک را فراخوانی کرده و ارجاعی به رشته‌ی "Jeff" را بدان ارسال می‌کند. این کمی عجیب به نظر می‌رسد، اما این به خوبی کار می‌کند و تا زمانی که به اتفاقات درون آن فکر نکنید عادی به نظر می‌رسد.

خاصیت گسترشی

خیلی خوب می‌شد اگر این مفهوم متدهای گسترشی ویژه‌ی سی‌شارپ نبود. به خصوص، ما می‌خواهیم برنامه‌نویسان، مجموعه‌ای از متدهای گسترشی را در بعضی از زبان‌های برنامه‌نویسی تعریف کنند تا افراد در زبان‌های برنامه‌نویسی دیگر از آن‌ها بهره ببرند. برای عملی شدن این، کامپایلر مورد انتخاب باید جستجوی نوع‌ها و متدهای استاتیک برای یافتن متدهای گسترشی را پشتیبانی کند. و کامپایلرها باید این کار را به سرعت انجام دهند تا زمان کامپایلر مینیمم باقی بماند.

در سی‌شارپ، وقتی شما اولین پارامتر یک متد استاتیک را با کلمه کلیدی **this** علامت می‌زنید، در درون، کامپایلر یک خاصیت را بر متد اعمال می‌کند و این خاصیت در متادیتای فایل تولیدی قرار می‌گیرد. این خاصیت در اسمبلی **System.Core.dll** جای دارد و شبیه به این است:

```
// Defined in the System.Runtime.CompilerServices namespace
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class | AttributeTargets.Assembly)]
public sealed class ExtensionAttribute : Attribute {
}
```

به علاوه، این خاصیت بر متادیتای هر کلاس که حداقل یک متد گسترشی داشته باشد، اعمال می‌گردد. و این خاصیت همچنین بر هر اسمبلی که حداقل یک کلاس استاتیک که حداقل یک متد گسترشی داشته باشد، اعمال می‌گردد. پس اکنون، هنگام کامپایلر کدی که یک متد نمونه که وجود ندارد را

فراخوانی می‌کند، کامپایلر می‌تواند به سرعت تمام اسمبلی‌های ارجاعی را جست و جو کند تا بداند کدامیک از آن‌ها متد گسترشی دارند. سپس آن فقط این اسمبلی‌ها را برای کلاس‌های استاتیک حاوی متدهای گسترشی جستجو می‌کند و فقط می‌تواند متدهای گسترشی را برای تطابق احتمالی کامپایل کند تا کد در سریعترین زمان کامپایل شود.

نکته کلاس **ExtensionAttribute** در اسمبلی **System.Core.dll** تعریف شده است. این یعنی اسمبلی تولیدی توسط کامپایلر، یک ارجاع به **System.Core.dll** دارد حتی اگر کد هیچ ارجاعی به نوع‌های **System.Core.dll** نداشته و حتی هنگام کامپایل کد، ارجاعی به **System.Core.dll** ندهیم. اما این مسئله بدی نیست، چون **ExtensionAttribute** تنها در زمان کامپایل استفاده می‌شود؛ در زمان اجرا **System.Core.dll** نیاز به بارگذاری ندارد مگر آنکه برنامه از نوع‌های آن استفاده کند.

متدهای جزئی

تصور کنید ابزاری دارید که یک فایل سورس کد سی‌شارپ شامل تعریف یک نوع را تولید می‌کند. ابزار می‌داند که مکان‌های احتمالی در کد وجود دارد که شما بخواهید در آنجا رفتار نوع را سفارشی کنید. به صورت عادی، سفارشی‌سازی از طریق فراخوانی متدهای مجازی در کد تولید شده توسط ابزار، صورت می‌گیرد. کد تولیدی توسط ابزار، همچنین حاوی تعریف سه متد مجازی است و این متدها کاری انجام نمی‌دهند و فقط برمی‌گردند. حال، اگر شما بخواهید رفتار کلاس را سفارشی کنید، شما کلاس خودتان را تعریف می‌کنید، آن را از کلاس پایه مشتق می‌کنید و هر متد مجازی را بازنویسی کرده تا رفتار دلخواه شما را داشته باشد.

نمونه‌ای در اینجا آمده است:

```
// Tool-produced code in some source code file:
internal class Base {
    private String m_name;

    // Called before changing the m_name field
    protected virtual void OnNameChanging(String value) {
    }

    public String Name {
        get { return m_name; }
        set {
            OnNameChanging(value.ToUpper()); // Inform class of potential change
            m_name = value; // Change the field
        }
    }
}

// Developer-produced code in some other source code file:
internal class Derived : Base {
    protected override void OnNameChanging(string value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

متأسفانه، دو مشکل با این کد وجود دارد:

- نوع باید کلاسی مهر شده (sealed) نباشد. شما نمی‌توانید از این تکنیک برای کلاس‌های مهر شده یا نوع‌های مقداری (چون نوع‌های مقداری به صورت ضمنی مهر شده‌اند) استفاده کنید. به علاوه، شما نمی‌توانید برای متدهای استاتیک از این تکنیک استفاده کنید چون آن‌ها قابل بازبینی نیستند.

- مشکلات کارایی نیز وجود دارد. یک نوع فقط برای بازنویسی یک متد تعریف شده است، این کار مقدار کمی از منابع سیستم را هدر می‌دهد. و حتی اگر شما نخواهید رفتار **OnNameChanging** را تغییر دهید، کد کلاس پایه هنوز متد مجازی‌ای را فراخوانی می‌کند که کاری جز برگشتن انجام نمی‌دهد. همچنین، **ToUpper** بدون توجه به اینکه **OnNameChanging** به پارامترش دسترسی پیدا می‌کند یا خیر، فراخوانی می‌شود.

ویژگی متدهای جزئی سی‌شارپ به شما اجازه‌ی بازنویسی رفتار یک نوع را می‌دهد در عین حال که مشکلات مطرح شده را برطرف می‌کند. کد زیر از متدهای جزئی برای پیاده‌سازی مفهوم کد قبلی استفاده می‌کند:

```
// Tool-produced code in some source code file:
internal sealed partial class Base {
    private String m_name;

    // This defining-partial-method-declaration is called before changing the m_name field
    partial void OnNameChanging(String value);

    public String Name {
        get { return m_name; }
        set {
            OnNameChanging(value.ToUpper()); // Inform class of potential change
            m_name = value; // Change the field
        }
    }
}
```

```
// Developer-produced code in some other source code file:
internal sealed partial class Base {

    // This implementing-partial-method-declaration is called before m_name is changed
    partial void OnNameChanging(String value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

به چندین نکته پیرامون نسخه‌ی جدید کد باید توجه کنید:

- اکنون کلاس مهر شده است (اگرچه مجبور نیست که باشد). در واقع، می‌توانست یک کلاس استاتیک یا حتی یک نوع مقداری باشد.
 - کد تولید شده توسط ابزار و کد تولید شده توسط برنامه‌نویس، دو تعریف جزئی هستند که در نهایت یک تعریف نوع را می‌سازند. برای اطلاعات بیشتر پیرامون نوع‌های جزئی به بخش "کلاس، ساختارها و رابط‌های جزئی" در فصل ۶ "مبانی نوع و عضو" مراجعه کنید.
 - کد تولید شده توسط ابزار، یک متد جزئی را تعریف می‌کند. این متد با نشانه **partial** علامت زده شده است و بدنه‌ای ندارد.
 - کد تولیدی توسط برنامه‌نویس، یک متد جزئی را پیاده‌سازی می‌کند. این متد نیز با نشانه **partial** علامت زده شده است و یک بدنه دارد.
- حال، وقتی شما این کد را کامپایل می‌کنید، اثری یکسان با کد قبلی می‌بینید. مجدداً مزیت اصلی اینجا اینست که شما می‌توانید ابزار را مجدداً اجرا کرده و کد جدیدی در فایل دیگری تولید کنید، اما کد شما در فایل مجزا و دست نخورده باقی می‌ماند. و این تکنیک برای کلاس‌های مهر شده، استاتیک و نوع‌های مقداری نیز کار می‌کند.

نکته در ویرایشگر ویژوال استودیو، اگر **partial** را تایپ کرده و **space** را بزیند، پنجره **IntelliSense**، تمام متدهای جزئی تعریف شده در نوع‌ها که هنوز پیاده‌سازی‌ای برایشان وجود ندارد را لیست می‌کند. سپس شما می‌توانید براحتی یک متد جزئی را از پنجره‌ی **IntelliSense** انتخاب کنید و ویژوال استودیو شکل کلی متد را برای شما تولید می‌کند. این ویژگی بسیار عالی است که منجر به افزایش بهره‌وری شما می‌شود.

اما، مزیت مهم دیگری وجود دارد که از متدهای جزئی حاصل می‌شود. فرض کنید شما نمی‌خواهید رفتار کد تولیدی توسط ابزار را تغییر دهید. اگر تنها خود کد تولیدی توسط ابزار را کامپایل کنید، کامپایلر، کد **IL** و متادیتا را به گونه‌ای تولید می‌کند که گویا کد تولیدی توسط ابزار، شبیه به زیر بوده است:

```
// Logical equivalent of tool-produced code if there is no
// implementing partial method declaration:
internal sealed class Base {
    private string m_name;

    public string Name {
        get { return m_name; }
        set {
            m_name = value;        // Change the field
        }
    }
}
```

یعنی اگر هیچ پیاده‌سازی متد جزئی یافت نشود، کامپایلر هیچ متادیتایی برای نمایش متد جزئی تولید نمی‌کند. به علاوه، کامپایلر هیچ دستور **IL** ای برای فراخوانی متد جزئی نیز تولید نمی‌کند و کامپایلر کدی برای ارزیابی آرگومان ارسالی به متد جزئی نیز تولید نمی‌کند. در این مثال، کامپایلر کدی برای فراخوانی متد **ToUpper** تولید نخواهد کرد. نتیجه آن است که متادیتا و **IL** کمتری تولید می‌شود و سرعت زمان اجرایی، بهتر است.

نکته متدهای جزئی شبیه به خاصیت **System.Diagnostics.ConditionalAttribute** عمل می‌کنند. هر چند، متدهای جزئی فقط درون یک نوع کار می‌کنند در حالیکه **ConditionalAttribute** می‌تواند برای فراخوانی متدهای تعریف شده در هر نوعی به کار رود.

قوانین و راهنمایی‌ها

چند قانون و راهنمایی اضافی وجود دارد که باید پیرامون متدهای جزئی بدانید:

- تنها در یک کلاس یا ساختار جزئی قابل تعریف هستند.
- متدهای جزئی همیشه باید دارای نوع برگشتی **void** باشند و نمی‌توانند دارای پارامتری علامت زده شده با **out** باشند. این محدودیت‌ها برای اینست که در زمان اجرا شاید متد وجود نداشته باشد و شما نمی‌توانید یک متغیر را به مقداری که شاید متد برگرداند مقداردهی اولیه کنید چون ممکن است متد وجود نداشته باشد. به طریق مشابه شما نمی‌توانید یک پارامتر **out** داشته باشید چون متد مجبور است که آن را مقداردهی اولیه کند و ممکن است متد اصلاً وجود نداشته باشد. یک متد جزئی می‌تواند پارامتر **ref** بگیرد. جنریک باشد، نمونه یا استاتیک باشد و شاید با **unsafe** نیز علامت زده شود.
- البته، تعریف متد جزئی و پیاده‌سازی متد جزئی باید دارای امضاهای یکسانی باشند. اگر هر دو دارای صفت‌های سفارشی اعمال شده باشند، کامپایلر صفت‌های آن‌ها را با همدیگر ترکیب می‌کند. هر خاصیت اعمال شده به یک پارامتر، نیز ترکیب می‌شود.
- اگر یک پیاده‌سازی متد جزئی موجود نباشد، آنگاه شما نمی‌توانید کدی داشته باشید که سعی در ساخت یک نماینده داشته باشد که به متد جزئی اشاره می‌کند. مجدداً علت اینست که در زمان اجرا شاید متد وجود نداشته باشد. کامپایلر این پیام را تولید می‌کند:
“error CS0762: Cannot create delegate from method ‘Base.OnChanging(String)’ because it is a partial method without an implementation declaration”
- متدهای جزئی همیشه به عنوان متدهای خصوصی در نظر گرفته می‌شوند. هرچند، کامپایلر سی‌شارپ به شما اجازه نمی‌دهد قبل از اعلان متد جزئی، کلمه کلیدی **private** را قرار دهید.

فصل ۹: پارامترها

این فصل بر راه‌های مختلف ارسال پارامتر به یک متد شامل اینکه چگونه به صورت انتخابی پارامترها را تعیین کنید، پارامترها را با نام تعیین کنید، پارامترها را با ارجاع ارسال کنید و چگونه متدهایی تعریف کنید که یک تعداد متغیری از پارامترها را دریافت می‌کنند تمرکز دارد.

پارامترهای نامی و انتخابی

هنگام طراحی پارامترهای یک متد، شما می‌توانید مقادیر پیش فرض را برای همه یا برخی از آن‌ها اختصاص دهید، آنگاه کدی که این متد را فراخوانی می‌کند، می‌تواند به صورت انتخابی، برخی از آرگومان‌ها را تعیین نکند، و مقادیر پیش فرض را بپذیرد. به علاوه، وقتی شما یک متد را فراخوانی می‌کنید می‌توانید آرگومان‌ها را با نام پارامترهایشان تعیین کنید. کدی را در زیر می‌بینید که هردوی پارامترهای نامی و انتخابی را نشان می‌دهد:

```
public static class Program {
    private static Int32 s_n = 0;

    private static void M(Int32 x = 9, String s = "A",
        DateTime dt = default(DateTime), Guid guid = new Guid()) {

        Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);
    }

    public static void Main() {
        // 1. Same as: M(9, "A", default(DateTime), new Guid());
        M();

        // 2. Same as: M(8, "X", default(DateTime), new Guid());
        M(8, "X");

        // 3. Same as: M(5, "A", DateTime.Now, Guid.NewGuid());
        M(5, guid: Guid.NewGuid(), dt: DateTime.Now);

        // 4. Same as: M(0, "1", default(DateTime), new Guid());
        M(s_n++, s_n++.ToString());

        // 5. Same as: String t1 = "2"; Int32 t2 = 3;
        // M(t2, t1, default(DateTime), new Guid());
        M(s: (s_n++).ToString(), x: s_n++);
    }
}
```

وقتی این برنامه را اجرا می‌کنم، خروجی زیر را می‌گیرم:

```
x=9, s=A, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=8, s=X, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=5, s=A, dt=7/2/2009 10:14:25 PM, guid=d24a59da-6009-4aae-9295-839155811309
x=0, s=1, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=3, s=2, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
```

همانگونه که می‌بینید، وقتی آرگومان‌های هنگام فراخوانی حذف می‌شود، کامپایلر سی‌شارپ مقدار پیش فرض پارامتر را قرار می‌دهد. سومین و پنجمین فراخوانی به **M** از ویژگی پارامترهای نامی سی‌شارپ بهره می‌برد. در هر دو آن‌ها، من صریحاً یک مقدار برای **x** ارسال کردم و گفتم که می‌خواهم یک آرگومان برای پارامترهای **guid** و **dt** ارسال کنم.

هنگام ارسال آرگومان به یک متد، کامپایلر، آرگومان‌ها را از چپ به راست ارزیابی می‌کند. در چهارمین فراخوانی به **M**، مقدار درون **s_n** که **0** است برای **x** ارسال می‌شود، سپس **s_n** افزوده می‌شود و **s_n (1)** به عنوان رشته برای **s** ارسال می‌گردد و سپس **s_n** مجدداً به **2** افزایش می‌یابد. وقتی شما آرگومان‌ها را با پارامترهای نامی ارسال می‌کنید، کامپایلر هنوز هم آرگومان‌ها را از چپ به راست ارزیابی می‌کند. در پنجمین فراخوانی **M**، مقدار درون **s_n (2)** به رشته تبدیل شده و در یک متغیر موقتی (**t1**) که کامپایلر ایجاد می‌کند ذخیره می‌شود. سپس **s_n** به **3** افزایش می‌یابد و این مقدار در متغیر موقتی دیگری (**t2**) که توسط کامپایلر ساخته می‌شود ذخیره می‌گردد و سپس **s_n** مجدداً به **4** افزایش می‌یابد. سرانجام **M** فراخوانی می‌شود و **t1**، **t2**، یک **DateTime** پیش فرض و یک **Guid** جدید بدان ارسال می‌شود.

قوانین و راهنمایی‌ها

چندین قانون و راهنمایی اضافی وجود دارد که شما هنگام تعریف یک متد که مقادیر پیش فرض برای پارامترهایش تعیین می‌کند، باید لحاظ کنید:

- شما می‌توانید مقادیر پیش فرض را برای پارامتر متدها، متدهای سازنده و ویژگی‌های پارامتردار (ایندکس‌های سی‌شارپ) تعیین کنید. شما همچنین می‌توانید مقادیر پیش فرض را برای پارامترهایی که بخشی از تعریف یک نماینده هستند، تعیین کنید. آنگاه هنگام به کار بردن یک متغیر از این نوع نماینده، می‌توانید آرگومان‌ها را حذف کرده و مقادیر پیش فرض را بپذیرید.
- پارامترهایی که مقادیر پیش فرض دارند باید بعد از هر پارامتری بیایند که مقدار پیش فرض ندارند. این یعنی، وقتی یک پارامتر تعریف می‌کنید که دارای مقدار پیش فرض است، تمام پارامترهای سمت راست آن هم باید مقدار پیش فرض داشته باشند. برای نمونه، در تعریف متد **M** من، من در صورتی که مقدار پیش فرض برای **s**، ("**A**") را حذف کنم، یک خطای کامپایلر دریافت می‌کنم. یک استثنا برای این قانون وجود دارد: یک پارامتر آرایه‌ای **param** (که در ادامه فصل توضیح می‌دهم) باید پس از تمام پارامترها (شامل آن‌هایی که مقدار پیش فرض دارند) بیاید و آرایه، خودش نمی‌تواند مقدار پیش فرض داشته باشد.
- مقادیر پیش فرض باید در زمان کامپایل مقدار ثابت شناخته شده باشند. این یعنی شما می‌توانید مقادیر پیش فرض را برای نوع‌هایی از پارامترها که سی‌شارپ آن‌ها را نوع اصلی تلقی می‌کند، قرار دهید، همانطور که در جدول ۱-۵ در فصل ۵ "نوع‌های اصلی، ارجاعی و مقداری" نشان داده شد. این همچنین شامل نوع‌های شمارشی و هر نوع ارجاعی قابل تنظیم به **null** نیز می‌شود. برای یک پارامتر از یک نوع مقداری دلخواه، شما می‌توانید مقدار پیش فرض را برابر با نمونه‌ای از یک نوع مقداری که تمام فیلدهای نمونه صفر باشند، قرار دهید. شما می‌توانید از کلمه کلیدی **default** یا کلمه کلیدی **new** برای این کار استفاده کنید، هر دو نحو، کد **IL** یکسانی تولید می‌کنند. نمونه‌هایی از هر دو نحو توسط متد **M** به ترتیب برای تنظیم مقدار پیش فرض پارامتر **dt** و پارامتر **guid** استفاده شده است.
- مراقب باشید که متغیرهای پارامتر را تغییر نام ندهید چون هر فراخوانی کننده‌ای که آرگومان‌ها را با نام پارامتر ارسال کند مجبور است کدش را تغییر دهد. برای نمونه، در تعریف متد **M** من، اگر من متغیر **dt** را به **datetime** تغییر دهم، آنگاه سومین فراخوانی من به **M** در کد قبلی باعث می‌شود کامپایلر این پیام را تولید کند:

"error CS1739: The best overload for 'M' does not have a parameter named 'dt'."

- توجه کنید تغییر مقدار پیش فرض یک پارامتر احتمالاً خطرناک است اگر متد خارج از مازول فراخوانی شده باشد. یک فراخوانی کننده مقدار پیش فرض را در فراخوانی‌اش تعبیه می‌کند. اگر شما بعداً مقدار پیش فرض پارامتر را تغییر دهید، و کد حاوی فراخوانی کننده را کامپایل مجدد نکنید، آنگاه باعث می‌شود که متد شما با ارسال مقدار پیش فرض قبلی فراخوانی شود. شما شاید بخواهید از مقدار پیش فرض **0/null** به عنوان یک نگرهبان برای بیان رفتار پیش فرض استفاده کنید و این به شما اجازه می‌دهد که پیش فرض را بدون نیاز به کامپایل مجدد تمام فراخوانی کننده‌ها، تغییر دهید. نمونه‌ای ببینیم:

// Don't do this:

```
private static String MakePath(String filename = "Untitled") {
    return String.Format(@"C:\{0}.txt", filename);
}
```

// Do this instead:

```
private static String MakePath(String filename = null) {
    // I am using the null-coalescing operator (??) here; see Chapter 19
    return String.Format(@"C:\{0}.txt", filename ?? "Untitled");
}
```

- شما نمی‌توانید برای پارامترهای علامت زده شده با کلمه کلیدی **ref** یا **out** مقدار پیش فرض تعیین کنید چون راهی برای ارسال یک مقدار پیش فرض معنی دار برای این پارامترها وجود ندارد.
- چندین قانون و راهنمایی دیگر وجود دارد که هنگام فراخوانی یک متد که از پارامترهای نامی یا انتخابی استفاده می‌کند باید بدانید:
- آرگومان‌ها به هر ترتیبی می‌توانند ارسال شوند، اما آرگومان‌های نامی همیشه باید در انتهای لیست آرگومان‌ها بیایند.
- شما می‌توانید آرگومان‌ها را با نام به پارامترهایی که مقدار پیش فرض ندارند، ارسال کنید، اما تمام آرگومان‌های مورد نیاز باید ارسال شوند (با موقعیت یا با نام) تا کامپایلر بتواند کد را کامپایل کند.
- سی‌شارپ اجازه نمی‌دهد که آرگومان‌ها را بین کاما حذف کنید مثل این: **M(1, , DateTime.Now)**، چون این باعث می‌شود که شمارنده کاما نتواند درست عمل کند. آرگومان‌ها را با نام پارامترهایشان ارسال کنید اگر می‌خواهید چند آرگومان را برای پارامترهایی با مقدار پیش فرض، حذف کنید.
- جهت ارسال یک آرگومان با نام پارامتر که نیاز به **ref/out** دارد، از این نحو استفاده کنید:

```
// Method declaration:
private static void M(ref Int32 x) { ... }
```

```
// Method invocation:
Int32 a = 5;
M(x: ref a);
```

نکته ویژگی پارامترهای نامی و انتخابی سی‌شارپ واقعا هنگام نوشتن کد سی‌شارپی که با مدل شی COM در Microsoft Office کار می‌کند، بسیار رایج و مفید است. سی‌شارپ اجازه‌ی حذف **ref/out** هنگام ارسال یک آرگومان با ارجاع را می‌دهد تا کدنویسی را بیشتر ساده کند. وقتی در حال فراخوانی یک کامپوننت COM نیستید، سی‌شارپ نیاز دارد که کلمه کلیدی **ref/out** بر آرگومان اعمال شود.

صفت‌های Optional و DefaultValue

خیلی خوب می‌شد اگر این ویژگی آرگومان‌های انتخابی و پیش فرض مخصوص سی‌شارپ نبود. مخصوصاً، ما می‌خواهیم برنامه‌نویسان در یک زبان برنامه‌نویسی، متدی تعریف کنند که بیان کند کدام پارامترها انتخابی هستند و کدامیک مقدار پیش فرض دارند. و آنگاه برنامه‌نویسان در زبان برنامه‌نویسی دیگر قابلیت فراخوانی این متد را داشته باشند. برای عملی شدن این، کامپایلر مورد انتخاب باید به فراخوانی کننده اجازه‌ی حذف برخی آرگومان‌ها را بدهد و راهی برای تعیین مقادیر پیش فرض آن‌ها داشته باشد.

در سی‌شارپ وقتی شما به یک پارامتر یک مقدار پیش فرض می‌دهید، کامپایلر، صفت **System.Runtime.InteropServices.OptionalAttribute** را بر پارامتر اعمال می‌کند و این صفت در متادیتای فایل حاصل جای می‌گیرد. به علاوه، کامپایلر، صفت **System.Runtime.InteropServices.DefaultParameterValueAttribute** را بر پارامتر اعمال کرده و این صفت در متادیتای فایل تولیدی قرار می‌گیرد. آنگاه، به سازه‌ی **DefaultParameterValueAttribute** مقدار ثابتی که شما در سورس کد تعیین کرده اید را ارسال می‌کند.

اکنون، وقتی یک کامپایلر می‌بیند که شما کدی دارید که یک متد که چند آرگومانش وجود ندارد را فراخوانی می‌کند، کامپایلر می‌تواند مطمئن شود که شما آرگومان‌های انتخابی را حذف کرده اید، مقادیر پیش فرض آن‌ها را از متادیتا بدست می‌آورد و مقادیر را در فراخوانی به صورت خودکار برای شما قرار می‌دهد.

متغیرهای محلی با نوع ضمنی

سی‌شارپ از قابلیت استنتاج نوع یک متغیر محلی در یک متد، از روی نوع عبارتی که برای مقداردهی آن استفاده شده است، پشتیبانی می‌کند. نمونه کدی که از این ویژگی بهره می‌برد:

```
private static void ImplicitlyTypedLocalVariables() {
    var name = "Jeff";
    ShowVariableType(name);           // Displays: System.String

    // var n = null;                 // Error
    var x = (Exception)null;         // OK, but not much value
    ShowVariableType(x);             // Displays: System.Exception

    var numbers = new Int32[] { 1, 2, 3, 4 };
    ShowVariableType(numbers);       // Displays: System.Int32[]

    // Less typing for complex types
    var collection = new Dictionary<String, Single>( ) { { ".NET", 4.0f } };

    // Displays: System.Collections.Generic.Dictionary`2[System.String, System.Single]
    ShowVariableType(collection);
    foreach (var item in collection) {
        // Displays: System.Collections.Generic.KeyValuePair`2[System.String, System.Single]
        ShowVariableType(item);
    }
}

private static void ShowVariableType<T>(T t) {
    Console.WriteLine(typeof(T));
}
```

اولین خط کد درون متد **ImplicitlyTypedLocalVariables** یک متغیر محلی جدید با نشانه **var** سی‌شارپ را معرفی می‌کند. برای تعیین نوع متغیر **name**، کامپایلر به نوع عبارت سمت راست عملگر انتساب (=) نگاه می‌کند. چون **"Jeff"** یک رشته است، کامپایلر می‌فهمد که نوع **name** باید **String** باشد. برای اثبات اینکه کامپایلر نوع را به درستی استنتاج کرده است، من متد **ShowVariableType** را نوشته‌ام. این متد جنریک نوع آرگومان‌ش را بدست می‌آورد و سپس این نوع را در کنسول نمایش می‌دهد. من آنچه **ShowVariableType** نشان می‌دهد را به عنوان کامنت، درون متد **ImplicitlyTypedLocalVariable** اضافه کرده‌ام.

دومین انتساب (که کامنت شده است) درون متد **ImplicitlyTypedLocalVariables** منجر به یک خطای کامپایل می‌شود:

“error CS0815: Cannot assign <nul> to an implicitly-typed local variable”

چون به صورت ضمنی، **null** قابل تبدیل به هر نوع ارجاعی یا نوع مقداری تهی پذیر است؛ بنابراین، کامپایلر نمی‌تواند یک نوع مشخص برای آن تعیین کند. اما در سومین انتساب، من نشان داده‌ام که این ممکن است که یک متغیر محلی با نوع ضمنی را با **null** مقداردهی اولیه کنید اگر صریحاً یک نوع برایش تعیین کنید (در مثال من **Exception**). در حالیکه این ممکن است، آنقدر کاربرد ندارد چرا که شما می‌توانید بنویسید **Exception x = null;** تا همان نتیجه را بگیرید.

در چهارمین انتساب، شما ارزش واقعی ویژگی متغیر محلی با نوع ضمنی سی‌شارپ را می‌بینید. بدون این ویژگی، شما می‌بایست در هر دو طرف عملگر، **Dictionary<String, Single>** را تعیین کنید. نه تنها باید کلی تایپ کنید بلکه اگر تصمیم به تغییر نوع مجموعه یا هر یک از نوع پارامترهای جنریک آن بگیرید، باید در هر دو طرف عملگر، کد خود را تغییر دهید.

در حلقه‌ی **foreach** نیز از **var** استفاده کرده‌ام تا کامپایلر به صورت خودکار نوع عناصر درون مجموعه را تشخیص دهد. این نشان می‌دهد استفاده از **var** با عبارت‌های **using**، **foreach** و **for** ممکن بوده و بسیار مفید است. همچنین هنگام تست کد نیز مفید است. برای نمونه، شما یک متغیر محلی با نوع ضمنی را با نوع برگشتی یک متد مقداردهی اولیه می‌کنید و هنگام نوشتن متدتان، شاید تصمیم به تغییر نوع بازگشتی آن بگیرید. اگر این کار را بکنید، کامپایلر به صورت خودکار می‌فهمد که نوع برگشتی تغییر کرده و به صورت خودکار نوع متغیر را تغییر می‌دهد. این عالی است، اما البته، کدهای دیگری در متد که از متغیر استفاده می‌کنند شاید دیگر کامپایلر نشوند اگر کدی که بوسیله این متغیر به اعضا دسترسی دارد تصور کند که متغیر هنوز نوع قدیمی را داراست.

در ویژوال استودیو، شما می‌توانید موس خود را روی **var** در کدتان ببرید و ویرایشگر یک اعلان که در آن نوعی که کامپایلر از روی عبارت استنتاج کرده است را نشان می‌دهد. ویژگی متغیر محلی با نوع ضمنی سی‌شارپ باید هنگام کار با نوع‌های ناشناس درون یک متد، به کار رود. برای جزئیات به فصل ۱۰ "ویژگی‌ها" مراجعه کنید.

شما نمی‌توانید نوع پارامتر یک متد را با **var** مشخص کنید. علت این امر باید برای شما واضح باشد، چون کامپایلر مجبور خواهد بود که نوع پارامتر را از روی آرگومان ارسالی در مکان فراخوانی تشخیص دهد و ممکن است یک مکان فراخوانی و یا تعداد زیادی مکان فراخوانی موجود باشد. به علاوه، شما نمی‌توانید نوع یک فیلد را با **var** مشخص کنید. چندین علت وجود دارد که چرا سی‌شارپ این محدودیت را گذاشته است. یک علت آنست که فیلدها توسط چندین متد قابل دسترسی هستند و تیم سی‌شارپ فکر کردند که این قرارداد (نوع متغیر) باید صریحا بیان شود. دلیل دیگر اینکه، اجازه‌ی انجام چنین کاری به یک نوع ناشناس (توضیح داده شده در فصل ۱۰) اجازه می‌دهد که به بیرون از یک تک متد نفوذ کند.

مهم dynamic را با **var** قاطی نکنید. تعریف یک متغیر محلی با **var** فقط یک کوتاه سازی نحوی است که کامپایلر، نوع را از روی یک عبارت استنتاج می‌کند. کلمه کلیدی **var** فقط برای تعریف یک متغیر محلی درون یک متد قابل استفاده است در حالیکه کلمه کلیدی **dynamic** می‌تواند برای متغیرهای محلی، فیلدها و آرگومان‌ها استفاده شود. شما نمی‌توانید یک عبارت را به یک **var** تبدیل (cast) کنید اما می‌توانید یک عبارت را به یک **dynamic** تبدیل کنید. شما باید صریحا یک متغیر که با **var** تعریف شده است را مقداردهی اولیه کنید در حالیکه مجبور نیستید یک متغیر تعریف شده با **dynamic** را مقداردهی اولیه کنید. برای اطلاعات بیشتر درباره‌ی نوع **dynamic** سی‌شارپ، بخش "نوع اصلی dynamic" در فصل ۵ را ببینید.

ارسال پارامتر با ارجاع به یک متد

به صورت پیش فرض، اجرایی زبان مشترک (CLR) فرض می‌کند تمام پارامترهای متدها با مقدار ارسال می‌شوند. وقتی اشیاء نوع ارجاعی ارسال می‌شوند، ارجاع (یا اشاره‌گر) به شی (با مقدار) به متد ارسال می‌شود. این یعنی متد می‌تواند شی را تغییر دهد و فراخوانی کننده، تغییر را برای نمونه‌های نوع مقداری، یک کپی از نمونه به متد ارسال می‌شود. این یعنی، متد، کپی خصوصی خودش از نوع مقداری را می‌گیرد و نمونه موجود در فراخوانی کننده، تغییر پیدا نمی‌کند.

مهم در یک متد، شما باید بدانید که پارامتر ارسالی، یک نوع ارجاعی است یا یک نوع مقداری، چون کدی که پارامتر را دستکاری می‌کند می‌تواند کاملا متفاوت باشد.

CLR به شما اجازه می‌دهد که پارامترها را به جای مقدار، با ارجاع ارسال کنید. در سی‌شارپ شما این کار را با کلمات کلیدی **out** و **ref** انجام می‌دهید. هر دو کلمه کلیدی به کامپایلر سی‌شارپ می‌گویند متادیتایی تولید کند که نشان دهد این پارامتر با ارجاع ارسال شده است و کامپایلر از این استفاده می‌کند تا کدی تولید نماید که آدرس پارامتر را به جای خود پارامتر، به متد ارسال کند.

از دید CLR، **out** و **ref** یکسان هستند. یعنی بدون توجه به کلمه کلیدی مورد استفاده، کد IL یکسانی تولید می‌شود و متادیتا نیز یکسان است به جز ۱ بیت، که برای تعیین این است که شما هنگام تعریف متد، **out** یا **ref** را مشخص کرده‌اید. هرچند، کامپایلر سی‌شارپ با دو کلمه کلیدی، متفاوت برخورد می‌کند و این تفاوت بیان می‌کند که چگونه متد، مسئول مقداردهی اولیه شی‌ای که بدان ارجاع شده، می‌باشد. اگر پارامتر یک متد با **out** علامت زده شده باشد، انتظار نمی‌رود فراخوانی کننده، شی را قبل از فراخوانی متد مقداردهی اولیه کرده باشد. متد فراخوانی شده نمی‌تواند از روی مقدار بخواند و متد فراخوانی شده قبل از برگشت، باید روی مقدار بنویسد. اگر پارامتر یک متد با **ref** علامت زده شده باشد، فراخوانی کننده باید قبل از فراخوانی متد، آن را مقداردهی کرده باشد. متد فراخوانی شده می‌تواند از روی مقدار بخواند و/یا روی آن بنویسد.

نوع‌های مقداری و ارجاعی با **out** و **ref**، بسیار متفاوت رفتار می‌کنند. بگذارید اول به استفاده از **out** و **ref** با نوع‌های مقداری بپردازیم:

```
public sealed class Program {
    public static void Main() {
        Int32 x; // x is uninitialized
        GetVal(out x); // x doesn't have to be initialized.
        Console.WriteLine(x); // Displays "10"
    }

    private static void GetVal(out Int32 v) {
        v = 10; // This method must initialize v.
    }
}
```

در این کد، **x** در قاب پشت‌پشته‌ی **Main** تعریف شده است. پس، آدرس **x** به **GetVal** ارسال می‌شود. متغیر **v** از **GetVal**، یک اشاره‌گر به مقدار **Int32** در قاب پشت‌پشته‌ی **Main** است. درون **GetVal**، **Int32** که **v** بدان اشاره دارد به **10** تغییر می‌کند. وقتی **GetVal** برمی‌گردد، **x** از **Main** دارای مقدار **10** است، و **10** در کنسول نمایش داده می‌شود. استفاده از **out** با نوع‌های مقداری بزرگ کارآمد است چون هنگام فراخوانی‌های متد، از کپی شدن فیلدهای نمونه‌های نوع مقداری، جلوگیری می‌کند.

حال به مثالی نگاه کنید که به جای **out** از **ref** استفاده می‌کند:

```
public sealed class Program {
    public static void Main() {
        Int32 x; // x is uninitialized
        GetVal(out x); // x doesn't have to be initialized.
        Console.WriteLine(x); // Displays "10"
    }

    private static void GetVal(out Int32 v) {
        v = 10; // This method must initialize v.
    }
}
```

در این کد، باز هم **x** در قاب پشت‌پشته‌ی **Main** تعریف شده و به **5** مقداردهی اولیه شده است. سپس، آدرس **x** به **AddVal** ارسال می‌شود. **v** از **AddVal** یک اشاره‌گر به مقدار **Int32** در قاب پشت‌پشته‌ی **Main** است. درون **AddVal**، **Int32** که **v** بدان اشاره دارد باید از قبل مقداری داشته باشد. پس، **AddVal** می‌تواند از مقدار اولیه در هر عبارتی که بخواهد استفاده کند. همچنین **AddVal** می‌تواند مقدار را تغییر دهد و مقدار جدید به فراخوانی‌کننده "برمی‌گردد". در این مثال، **AddVal** **10** را به مقدار اولیه می‌افزاید. وقتی **AddVal** برمی‌گردد **x** از **Main** حاوی **15** خواهد بود که چیزی است که در کنسول نشان داده می‌شود. برای بخاطر‌سپاری، از دید CLR یا IL، **out** و **ref** دقیقاً کار یکسانی انجام می‌دهند، هر دو باعث می‌شوند یک اشاره‌گر به نمونه، ارسال شود. تفاوت این است که کامپایلر مطمئن می‌شود کد صحیح باشد. کد زیر که سعی دارد یک مقدار مقداردهی اولیه نشده به یک متد که انتظار یک پارامتر **ref** دارد، را ارسال کند، خطای زیر را باعث می‌شود:

"error CS0165: Use of unassigned local variable 'x'."

```
public sealed class Program {
    public static void Main() {
        Int32 x; // x is not initialized.

        // The following line fails to compile, producing
        // error CS0165: Use of unassigned local variable 'x'.
        AddVal(ref x);
    }
}
```



```

        Console.WriteLine(x);
    }

    private static void AddVal(ref Int32 v) {
        v += 10;    // This method can use the initialized value in v.
    }
}

```

مهم اغلب از من می‌پرسند چرا سی‌شارپ نیاز دارد که یک فراخوانی به یک متد باید **out** یا **ref** را تعیین کند. گذشته از این، کامپایلر می‌داند که متد فراخوانی شده به **out** یا **ref** نیاز دارد و باید بتواند کد را به درستی کامپایل کند. به نظر می‌رسد کامپایلر می‌تواند به صورت خودکار کار درست را انجام دهد. اما، طراحان زبان سی‌شارپ احساس کردند که فراخوانی کننده باید صریحاً هدفش را بیان کند. به این روش، در مکان فراخوانی، به وضوح انتظار می‌رود متدی که فراخوانی می‌شود، مقدار متغیر ارسالی را تغییر دهد. به علاوه، CLR اجازه می‌دهد که متدها را بر اساس استفاده‌ی آن‌ها از پارامترهای **out** و **ref** سربارگذاری کنید. برای نمونه، در سی‌شارپ کد زیر مجاز بوده و به خوبی کامپایل می‌شود:

```

public sealed class Point {
    static void Add(Point p) { ... }
    static void Add(ref Point p) { ... }
}

```

من مجاز نیستم متدهایی که فقط با **out** و **ref** فرق می‌کنند را سربارگذاری کنم چون نمایش متادیتای امضای متدها یکسان می‌شود. پس من همچنین نمی‌توانم متد زیر را در نوع قبلی **Point**، تعریف کنم:

```

static void Add(out Point p) { ... }

```

اگر شما سعی کنید آخرین متد **Add** در نوع **Point** را کامپایل کنید، کامپایلر سی‌شارپ این پیام را نشان می‌دهد:

“error CS0663: ‘Add’ cannot define overloaded methods that differ only on ref and out.”

استفاده از **out** و **ref** با نوع‌های مقداری، همان رفتاری را به شما می‌دهد که قبلاً هنگام ارسال نوع‌های ارجاعی با مقدار، دیدید. با نوع‌های مقداری، **out** و **ref** اجازه می‌دهند یک متد یک تک نمونه نوع مقداری را دستکاری کند. فراخوانی کننده باید حافظه را برای نمونه تخصیص دهد و فراخوانی شونده، حافظه را دستکاری می‌کند. با نوع‌های ارجاعی، فراخوانی کننده حافظه را برای یک اشاره‌گر به شی ارجاعی، تخصیص می‌دهد و فراخوانی شونده این اشاره‌گر را دستکاری می‌کند. به خاطر این رفتار، استفاده از **out** و **ref** با نوع‌های ارجاعی تنها وقتی مفید است که متد قصد دارد یک ارجاع به یک شی که درباره‌اش می‌داند را "برگرداند". کد زیر این مطلب را نمایش می‌دهد:

```

using System;
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs; // fs is uninitialized

        // Open the first file to be processed.
        StartProcessingFiles(out fs);

        // Continue while there are more files to process.
        for (; fs != null; ContinueProcessingFiles(ref fs)) {

```

```

        // Process a file.
        fs.Read(...);
    }
}

private static void StartProcessingFiles(out FileStream fs) {
    fs = new FileStream(...); // fs must be initialized in this method
}

private static void ContinueProcessingFiles(ref FileStream fs) {
    fs.Close(); // Close the last file worked on.

    // Open the next file, or if no more files, "return" null.
    if (noMoreFilesToProcess) fs = null;
    else fs = new FileStream (...);
}
}

```

همانگونه که می‌بینید، تفاوت بزرگ این کد این است که متدهایی که پارامترهای نوع ارجاعی **out** یا **ref** دارند، یک شی جدید می‌سازند و اشاره‌گر به شی جدید به فراخوانی کننده، باز می‌گردد. متوجه خواهید شد که متد **ContinueProcessingFiles** می‌تواند قبل از برگرداندن یک شی جدید، شی ارسال شده را دستکاری کند. این امر ممکن است، چون پارامتر با کلمه کلیدی **ref** علامت زده شده است. شما می‌توانید کد قبلی را کمی ساده تر کنید:

```

using System;
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs = null; // Initialized to null (required)

        // Open the first file to be processed.
        ProcessFiles(ref fs);

        // Continue while there are more files to process.
        for (; fs != null; ProcessFiles(ref fs)) {

            // Process a file.
            fs.Read(...);
        }
    }

    private static void ProcessFiles(ref FileStream fs) {
        // Close the previous file if one was open.
        if (fs != null) fs.Close(); // Close the last file worked on.

        // Open the next file, or if no more files, "return" null.
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}

```

مثال دیگری ببینید که از کلمه کلید **ref** برای پیاده‌سازی متدی استفاده می‌کند که دو نوع ارجاعی را جا به جا کند:

```
public static void Swap(ref Object a, ref Object b) {
    Object t = b;
    b = a;
    a = t;
}
```

برای جابجایی اشاره‌گر به دو شی **String**، شما احتمالاً فکر می‌کنید که می‌توانید کدی شبیه به زیر بنویسید:

```
public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";

    Swap(ref s1, ref s2);
    Console.WriteLine(s1); // Displays "Richter"
    Console.WriteLine(s2); // Displays "Jeffrey"
}
```

اما، این کد کامپایل نخواهد شد. مشکلی که هست اینست که متغیری که با ارجاع به یک متد ارسال می‌شود باید از همان نوعی باشد که در تعریف متد آمده است. به بیان دیگر، **Swap** انتظار دو اشاره‌گر **Object** را دارد، نه دو اشاره‌گر **String**. برای جابجایی دو اشاره‌گر **String**، باید این کار را بکنید:

```
public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";

    // variables that are passed by reference
    // must match what the method expects.
    Object o1 = s1, o2 = s2;
    Swap(ref o1, ref o2);

    // Now cast the objects back to strings.
    s1 = (String) o1;
    s2 = (String) o2;

    Console.WriteLine(s1); // Displays "Richter"
    Console.WriteLine(s2); // Displays "Jeffrey"
}
```

این نسخه از **SomeMethod** کامپایل شده و طبق انتظار اجرا می‌شود. علت آنکه پارامترهای ارسالی باید با پارامترهایی که متد انتظار دارد مطابق باشند اینست که مطمئن شود امنیت نوع برقرار است. کد زیر، که کامپایل هم نمی‌شود نشان می‌دهد چگونه امنیت نوع می‌تواند به خطر افتد:

```
internal sealed class SomeType {
    public Int32 m_val;
}

public sealed class Program {
    public static void Main() {
        SomeType st;

        // The following line generates error CS1503: Argument '1':
        // cannot convert from 'ref SomeType' to 'ref object'.
        GetAnObject(out st);
    }
}
```

```

        Console.WriteLine(st.m_val);
    }

    private static void GetAnObject(out Object o) {
        o = new String('x', 100);
    }
}

```

در این کد، **Main** به وضوح انتظار دارد **GetAnObject** یک شی **SomeType** برگرداند. اما چون امضای **GetAnObject** یک اشاره‌گر به یک **Object** را نشان می‌دهد، **GetAnObject** آزاد است که **o** را به شی‌ای از هر نوع، مقداره‌ی اولیه کند. در این مثال، وقتی **GetAnObject** به **Main** برمی‌گردد، **st** به یک **String** اشاره خواهد کرد، که واضح است که یک شی **SomeType** نیست. و فراخوانی به **Console.WriteLine** **GetAnObject** مطمئناً شکست می‌خورد. خوشبختانه، کامپایلر سی‌شارپ کد قبلی را کامپایل نمی‌کند چون **st** یک اشاره‌گر به **SomeType** است، اما **GetAnObject** به یک ارجاع به یک **Object** نیاز دارد. شما می‌توانید برای حل مشکل این متدها از جنریک‌ها استفاده کنید تا آن‌ها طبق انتظار شما عمل کنند. در اینجا چگونگی حل مشکل متد **Swap** که قبلاً نشان داده شد را می‌بینید:

```

public static void Swap<T>(ref T a, ref T b) {
    T t = b;
    b = a;
    a = t;
}

```

و اکنون، با بازنویسی **Swap** طبق بالا، کد زیر (یکسان با کدی که قبلاً نشان داده شد) کامپایل شده و درست کار می‌کند:

```

public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";

    Swap(ref s1, ref s2);
    Console.WriteLine(s1); // Displays "Richter"
    Console.WriteLine(s2); // Displays "Jeffrey"
}

```

برای مثال‌های دیگری که جنریک‌ها این مشکل را حل کرده‌اند به کلاس **Interlocked** از **System.Threading** و متدهای **CompareExchange** و **Exchange** آن مراجعه کنید.

ارسال تعداد متغیری از آرگومان‌ها به یک متد

گاهی این عادی است که برنامه‌نویس، متدی تعریف کند که تعداد متغیری از آرگومان‌ها را دریافت نماید. برای نمونه، نوع **System.String** متدهایی ارائه می‌کند که اجازه می‌دهند تعداد دلخواهی از رشته‌ها به هم متصل شوند و متدهایی ارائه می‌کند که اجازه می‌دهند فراخوانی کننده، مجموعه‌ای از رشته‌ها را برای فرمت کردن تعیین کند. برای تعریف یک متد که تعداد متغیری از آرگومان یافت کند، متد را طبق زیر تعریف کنید:

```

static Int32 Add(params Int32[] values) {
    // NOTE: it is possible to pass the 'values'
    // array to other methods if you want to.

    Int32 sum = 0;
    if (values != null) {
        for (Int32 x = 0; x < values.Length; x++)
            sum += values[x];
    }
    return sum;
}

```

```
}
```

همه چیز در این متد به جز کلمه کلیدی **params** که بر آخرین پارامتر متد اعمال شده، آشناست. اگر یک لحظه، کلمه کلیدی **params** را در نظر نگیرید، واضح است که این متد یک آرایه از مقادیر **Int32** دریافت و در بین آن‌ها حرکت می‌کند و تمام مقادیر را با هم جمع می‌نماید. **Sum** به عنوان نتیجه به فراخوانی کننده برگشت داده می‌شود.

مطمئناً، کد می‌تواند این متد را اینگونه فراخوانی کند:

```
public static void Main() {
    // Displays "15"
    Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 }));
}
```

واضح است که آرایه می‌تواند با تعداد دلخواهی از عناصر که برای پردازش به **Add** ارسال می‌شوند، مقداردهی اولیه گردد. اگرچه کد قبلی کامپایل شده و به خوبی کار می‌کند، کمی بد ریخت است. به عنوان برنامه‌نویس، ما مطمئناً ترجیح می‌دهیم **Add** را مثل زیر فراخوانی کنیم:

```
public static void Main() {
    // Displays "15"
    Console.WriteLine(Add(1, 2, 3, 4, 5));
}
```

احتمالاً خوشحال خواهید شد اگر بدانید می‌توانید این کار را به خاطر وجود کلمه کلیدی **params** انجام دهید. کلمه کلیدی **params** به کامپایلر می‌گوید یک نمونه از صفت سفارشی **System.ParamArrayAttribute** را بر پارامتر اعمال کند.

وقتی کامپایلر سی‌شارپ یک فراخوانی به یک متد را دریافت می‌کند، کامپایلر تمام متدهای با نام تعیین شده را بررسی می‌کند، که هیچ پارامتری صفت **ParamArray** را نداشته باشد. اگر متدی این چنین وجود داشت که بتواند فراخوانی را بپذیرد، کامپایلر کد لازم برای فراخوانی متد را تولید می‌کند. اما اگر کامپایلر یک متد این چنینی نیافت، به دنبال متدهایی که یک صفت **ParamArray** دارند می‌گردد تا ببیند آیا فراخوانی را می‌تواند انجام دهد. اگر کامپایلر یک متد این چنینی بیابد، کدی تولید می‌کند که قبل از آنکه متد را فراخوانی کند، یک آرایه ساخته و عناصرش را پر کند.

در مثال قبل، هیچ متد **Add** که پنج آرگومان **Int32** بگیرد تعریف نشده است اما کامپایلر می‌بیند که سورس‌کد، یک فراخوانی به **Add** دارد که لیستی از مقادیر **Int32** بدان ارسال شده است و متد **Add** ای وجود دارد که پارامتر آرایه‌ای از **Int32** های آن با صفت **ParamArray** علامت زده شده است. پس کامپایلر این را یک تطبیق در نظر گرفته و کدی تولید می‌کند که پارامترها را درون یک آرایه **Int32** قرار داده و سپس متد **Add** را فراخوانی کند. نتیجه نهایی اینست که شما کد را می‌نویسید، به سادگی تعدادی پارامتر به **Add** ارسال می‌کنید، اما کامپایلر کدی تولید می‌کند که گویا شما نسخه اول که صریحاً یک آرایه را ساخته و مقداردهی می‌کند را نوشته اید.

تنها آخرین پارامتر یک متد می‌تواند با کلمه کلیدی **params (ParamArrayAttribute)** علامت زده شود. این پارامتر باید یک آرایه تک بعدی که از هر نوعی می‌تواند باشد را نشان دهد. مجاز است که **null** یا اشاره‌گری به آرایه‌ای از عناصر **0** به عنوان آخرین پارامتر به متد ارسال کنید. فراخوانی زیر به **Add** به خوبی کامپایل شده، اجرا می‌شود و مجموع **0** را (طبق انتظار) برمی‌گرداند.

```
public static void Main() {
    // Both of these lines display "0"
    Console.WriteLine(Add()); // passes new Int32[0] to Add
    Console.WriteLine(Add(null)); // passes null to Add: more efficient
    // (no array allocated)
}
```

تاکنون، تمام مثال‌ها نشان دادند چگونه متدی بنویسید که تعدادی دلخواه از پارامترهای **Int32** بگیرد. چگونه شما متدی می‌نویسید که تعدادی دلخواه از پارامترها بگیرد که پارامترها از هر نوعی باشند؟ جواب ساده است. تنها متد را به گونه‌ای تغییر دهید که به جای یک **Int32[]** یک **Object[]** بگیرد. متد زیر، **Type** هر شی‌ای که ارسال می‌شود به آن را نشان می‌دهد.

```
public sealed class Program {
    public static void Main() {
        DisplayTypes(new Object(), new Random(), "Jeff", 5);
    }
}
```

```
private static void DisplayTypes(params Object[] objects) {
    if (objects != null) {
        foreach (Object o in objects)
            Console.WriteLine(o.GetType());
    }
}
}
```

اجرای این کد منجر به خروجی زیر می‌شود:

```
System.Object
System.Random
System.String
System.Int32
```

مهم آگاه باشید که فراخوانی یک متد که تعداد متغیری از آرگومان‌ها را می‌گیرد، منجر به کاهش سرعت می‌شود مگر آنکه شما صریحاً **null** را ارسال کنید. گذشته از این، یک آرایه از اشیاء باید در هیپ تخصیص یابد، عناصر آرایه باید مقداردهی اولیه شوند، و سرانجام حافظه-ی آرایه باید جمع‌آوری گردد. به منظور کاهش اثر این کار بر سرعت، شما می‌توانید چند متد سربارگذاری شده که از کلمه کلیدی **params** استفاده نمی‌کنند را تعریف کنید. برای چند نمونه، به متد **Concat** از کلاس **System.String** که سربارگذاری‌های زیر را دارد نگاه کنید:

```
public sealed class String : Object, ... {
    public static string Concat(object arg0);
    public static string Concat(object arg0, object arg1);
    public static string Concat(object arg0, object arg1, object arg2);
    public static string Concat(params object[] args);
    public static string Concat(string str0, string str1);
    public static string Concat(string str0, string str1, string str2);
    public static string Concat(string str0, string str1, string str2,
        string str3);
    public static string Concat(params string[] values);
}
```

همانگونه که می‌بینید، متد **Concat** چندین سربارگذاری تعریف می‌کند که از کلمه کلیدی **params** استفاده نمی‌کنند. این نسخه‌های متد **Concat** سربارگذاری‌هایی هستند که اغلب فراخوانی می‌شوند و این سربارگذاری‌ها برای بهبود سرعت در اغلب سناریوها، وجود دارند. سربارگذاری‌ای که از کلمه کلیدی **params** استفاده می‌کند برای سناریوهایی است که کمتر اتفاق می‌افتد، چنین سناریوهایی بر سرعت لطمه وارد می‌کنند اما خوشبختانه نادر هستند.

راهنمایی‌های مربوط به پارامتر و نوع برگشتی

هنگام تعریف نوع‌های پارامترهای یک متد شما باید ضعیف‌ترین نوع ممکن را تعیین کنید و رابطه‌ها را بر کلاس‌های پایه ترجیح دهید. برای نمونه، اگر متدی می‌نویسید که عناصر یک مجموعه را دستکاری می‌کند، بهترین کار تعریف پارامتر متد با استفاده از یک رابط مثل **IEnumerable<T>** به جای استفاده از یک نوع داده‌ای قوی مثل **List<T>** یا حتی یک نوع رابط قوی‌تر مثل **ICollection<T>** یا **IList<T>** است:

```
// Desired: This method uses a weak parameter type
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }
```

```
// Undesired: This method uses a strong parameter type
public void ManipulateItems<T>(List<T> collection) { ... }
```

علت اینست که یک نفر می‌تواند متد اول را فراخوانی کند و به آن یک آرایه از اشیاء، یک شی **List<T>**، یک شی **String** و غیره – هر شی‌ای که نوعش **IEnumerable<T>** را پیاده‌سازی می‌کند – را ارسال کند. متد دوم تنها اجازه می‌دهد اشیاء **List<T>** ارسال شوند و یک آرایه یا یک شی **String** را نمی‌پذیرد. واضح است که متد اول بهتر است چون انعطاف پذیرتر بوده و می‌تواند در سناریوهای بیشتری استفاده شود.

طبیعاً، اگر شما متدی می‌نویسید که به یک لیست نیاز دارد (نه فقط هر شی قابل شمارش) آنگاه باید نوع پارامتر را یک **IList<T>** تعریف کنید. هنوز هم باید از تعریف نوع پارامتر به عنوان **List<T>** خودداری کنید. استفاده از **IList<T>** به فراخوانی کننده اجازه می‌دهد آرایه‌ها یا هر شی دیگری که نوعش **IList<T>** را پیاده‌سازی کند ارسال نماید.

دقت کنید که مثال‌های من در مورد مجموعه‌هاست که با معماری رابط طراحی شده اند. اگر ما در مورد کلاس‌هایی صحبت می‌کردیم که بر طبق معماری کلاس پایه طراحی شده اند، مفهوم باز هم همان می‌بود. پس برای نمونه، اگر من متدی می‌نوشتیم که بایت‌های یک استریم را پردازش می‌کرد، این را داشتیم:

```
// Desired: This method uses a weak parameter type
public void ProcessBytes(Stream someStream) { ... }
```

```
// Undesired: This method uses a strong parameter type
public void ProcessBytes(FileStream fileStream) { ... }
```

اولین متد می‌تواند بایت‌هایی از هر گونه استریم را پردازش کند: یک **FileStream**، یک **NetworkStream**، یک **MemoryStream** و ... متد دوم می‌تواند تنها بر یک **FileStream** عمل کند که آن را بسیار محدودتر می‌کند.

در سوی دیگر، معمولاً بهتر است که نوع برگشتی یک متد را با قوی‌ترین نوع ممکن تعریف کنید (سعی کنید خودتان را به یک نوع خاص محدود نکنید). برای نمونه، بهتر است متدی تعریف کنید که به جای برگرداندن یک شی **Stream**، یک شی **FileStream** برگرداند:

```
// Desired: This method uses a strong return type
public FileStream OpenFile() { ... }
```

```
// Undesired: This method uses a weak return type
public Stream OpenFile() { ... }
```

در اینجا، اولین متد ترجیح داده می‌شود چرا که اجازه می‌دهد فراخوانی کننده متد، با شی برگشتی به عنوان یک شی **FileStream** یا یک شی **Stream** رفتار کند. در حالیکه متد دوم نیاز دارد فراخوانی کننده، با شی برگشتی به عنوان یک شی **Stream** برخورد کند. اساساً، بهتر است به فراخوانی کننده اجازه دهید بیشترین انعطاف پذیری را هنگام فراخوانی یک متد داشته باشد تا متد بتواند در سناریوهای بیشتری استفاده شود.

گاهی می‌خواهید قابلیت تغییر پیاده‌سازی درونی یک متد بدون تاثیر بر فراخوانی کننده را داشته باشید. در مثال نشان داده شده، متد **OpenFile** بعید است پیاده‌سازی درونی خود را تغییر دهد و چیزی جز یک شی **FileStream** (یا یک شی که نوعش از **FileStream** مشتق شده) را برگرداند. اما اگر شما متدی دارید که یک شی **List<String>** برمی‌گرداند، شاید بخواهید پیاده‌سازی درونی این متد را در آینده به گونه‌ای تغییر دهید که یک **String[]** برگرداند. در مواردی که می‌خواهید حدی از انعطاف پذیری به منظور تغییر احتمالی در آینده برای خود باقی بگذارید، یک نوع برگشتی ضعیف‌تر را انتخاب کنید. برای نمونه:

```
// Flexible: This method uses a weaker return type
public IList<String> GetStringCollection() { ... }
```

```
// Inflexible: This method uses a stronger return type
public List<String> GetStringCollection() { ... }
```

در این مثال، اگرچه متد **GetStringCollection** در درون خود از یک شی **List<String>** استفاده کرده و آنرا برمی‌گرداند، بهتر است که متد به جای آن، یک **IList<String>** برگرداند. در آینده، متد **GetStringCollection** می‌تواند مجموعه درونی خود را به یک **String[]** تغییر دهد و فراخوانی کننده‌های متد نیاز به تغییر هیچ یک از سورس‌کدهای خود ندارد. در واقع، آن‌ها حتی نیاز به کامپایل مجدد کد خود ندارند. توجه کنید در این مثال،

من از میان ضعیف ترین نوع‌ها، قوی ترین آن‌ها را استفاده کرده ام. برای نمونه، من از یک `IEnumerable<String>` یا حتی `ICollection<String>` استفاده نکردم.

ثابت بودن

در برخی زبان‌ها، مثل `C++` مدیریت نشده، این ممکن است که متدها یا پارامترها را به عنوان ثابت تعریف کنید تا مانع از تغییر هر یک از فیلدهای شی توسط یک متد نمونه شده یا از تغییر شی ارسالی به متد توسط کد، جلوگیری بعمل آورد. `CLR` این را فراهم نمی‌کند و بسیاری از برنامه‌نویسان برای از دست دادن این ویژگی ناراحتند. چون `CLR` این ویژگی را ارائه نمی‌کند، هیچ زبانی (شامل سی‌شارپ) نمی‌تواند این ویژگی را ارائه کند.

اول اینکه، شما باید توجه کنید در `C++` مدیریت نشده، علامت زدن یک متد نمونه یا پارامتر به عنوان `const` فقط اطمینان حاصل می‌کند که برنامه‌نویس نتواند کد عادی‌ای بنویسد که شی یا پارامتر را تغییر دهد. درون متد، همیشه ممکن است کدی بنویسید که بتواند با تبدیل از `const` بودن یا با بدست آوردن آدرس شی/آرگومان و سپس نوشتن به آن آدرس، شی/پارامتر را تغییر دهد. در واقع، `C++` مدیریت نشده به برنامه‌نویسان دروغ می‌گفته است و باعث می‌شده که آن‌ها باور کنند اشیاء/آرگومان‌های ثابت آن‌ها، غیر قابل نوشتن است در حالیکه قابل نوشتن بوده‌اند.

هنگام طراحی پیاده‌سازی یک نوع، برنامه‌نویس می‌تواند از نوشتن کدی که اشیاء/آرگومان‌ها را دستکاری کند خودداری نماید. برای نمونه، رشته‌ها، تغییر ناپذیرند چون کلاس `String` متدی که بتواند یک شی رشته را تغییر دهد ارائه نمی‌کند.

همچنین، برای مایکروسافت خیلی مشکل است که به `CLR` این قابلیت را بدهد که بررسی کند یک شی/آرگومان ثابت، تغییر نیافته باشد. `CLR` می‌بایست در هر نوشتن بررسی کند که نوشتن بر روی یک شی ثابت رخ نداده است و این کار به شدت بر عملکرد آن اثرگذار است. البته، یک تخلف یافت شده باعث می‌شود که `CLR` یک اکسپشن تولید کند. علاوه بر این، پشتیبانی از ثابت‌ها، پیچیدگی زیادی برای برنامه‌نویسان ایجاد می‌کند. برای نمونه اگر یک نوع غیر قابل تغییر باشد، تمام نوع‌های مشتق شده از آن نیز باید این را رعایت کنند. به علاوه، یک نوع تغییر ناپذیر، احتمالاً از نوع‌هایی تشکیل می‌شود که خود آن‌ها از نوع‌های تغییر ناپذیر هستند.

این‌ها تنها تعدادی از دلایلی است که چرا `CLR` از اشیاء/آرگومان‌های ثابت پشتیبانی نمی‌کند.

فصل ۱۰: ویژگی‌ها

در این فصل، من درباره‌ی ویژگی‌ها صحبت می‌کنم. ویژگی‌ها به سورس‌کد اجازه می‌دهند که یک متد را با نحو ساده فراخوانی کند. CLR دو گونه از ویژگی‌ها را ارائه می‌کند: ویژگی‌های بدون پارامتر که به سادگی ویژگی `properties` نامیده می‌شوند و ویژگی‌های پارامتردار که توسط زبان‌های برنامه‌نویس مختلف به نام‌های مختلفی نامیده می‌شوند. برای نمونه سی‌شارپ، ویژگی‌های پارامتردار را `indexer` می‌نامد و ویژوال بیسیک آن‌ها را ویژگی پیش فرض `default properties` می‌نامد. من همچنین پیرامون مقداردهی اولیه ویژگی‌ها با استفاده از مقداردهی‌کننده‌های شی و مجموعه و چگونگی بسته‌بندی تعدادی از ویژگی‌ها با همدیگر به کمک نوع‌های ناشناس سی‌شارپ و نوع `System.Tuple` نیز توضیح می‌دهم.

ویژگی‌های بدون پارامتر

بسیاری از نوع‌ها اطلاعات وضعیتی را تعریف می‌کنند که قابل خواندن و تغییر است. اغلب، این اطلاعات وضعیتی، به عنوان فیلدهای یک نوع پیاده‌سازی می‌شوند. برای نمونه، یک تعریف شامل دو فیلد را می‌بینید:

```
public sealed class Employee {
    public String Name; // The employee's name
    public Int32 Age; // The employee's age
}
```

اگر می‌خواستید یک نمونه از این نوع بسازید، براحتی می‌توانستید اطلاعات وضعیتی آن را با کدی مشابه زیر بدست آورده یا تغییر دهید:

```
Employee e = new Employee();
e.Name = "Jeffrey Richter"; // Set the employee's Name.
e.Age = 45; // Set the employee's Age.
```

```
Console.WriteLine(e.Name); // Displays "Jeffrey Richter"
```

خواندن و تغییر اطلاعات وضعیتی یک شی به این روش بسیار رایج است. اما، من بحث می‌کنم چرا کد قبلی هرگز نباید بدین شکل پیاده‌سازی شود. یکی از نشانه‌های طراحی و برنامه‌نویسی شی گرا، کپسوله کردن داده‌ها `data encapsulation` است. کپسوله کردن داده‌ها یعنی فیلدهای نوع شما نباید هرگز در دسترس عمومی قرار گیرد چرا که به آسانی می‌توان کدی نوشت که از فیلدها استفاده نادرست کرده و وضعیت شی را خراب کند. برای نمونه، یک برنامه‌نویس براحتی می‌تواند یک شی `Employee` را با کد زیر خراب کند:

```
e.Age = -5; // How could someone be -5 years old?
```

چندین دلایل دیگر برای کپسوله کردن دسترسی به فیلدهای داده‌ای یک نوع وجود دارد. برای نمونه، شاید شما بخواهید دسترسی به یک فیلد، اثرات جانبی داشته باشد، برخی مقادیر را کش کند، یا اشیاء داخلی بسازد. شاید شما بخواهید دسترسی به فیلد، ترد-امن باشد. یا شاید فیلد یک فیلد منطقی باشد که مقدارش با بایت‌های درون حافظه نشان داده نمی‌شود و به جای آن توسط یک الگوریتم تولید می‌گردد.

به خاطر هر یک از این دلایل، هنگام طراحی یک نوع، قویا پیشنهاد می‌کنم که تمام فیلدهای خود را `private` کنید. سپس، برای اجازه دادن به کاربری از نوعتان، جهت بدست آوردن یا نوشتن اطلاعات وضعیتی، متدهایی برای این هدف خاص در دسترس بگذارید. متدهایی که دسترسی به یک فیلد را پنهان می‌کنند، متدهای دستیابی `accessor methods` نامیده می‌شوند. این متدهای دستیابی می‌توانند به صورت انتخابی وضعیت را بررسی کرده و مطمئن شوند که وضعیت شی هرگز خراب نشده باشد. برای نمونه، من کلاس قبلی را این گونه بازنویسی می‌کنم:

```
public sealed class Employee {
    private String m_Name; // Field is now private
    private Int32 m_Age; // Field is now private

    public String GetName() {
        return(m_Name);
    }
}
```

```

public void SetName(String value) {
    m_Name = value;
}

public Int32 GetAge() {
    return(m_Age);
}

public void SetAge(Int32 value) {
    if (value < 0)
        throw new ArgumentOutOfRangeException("value", value.ToString(),
            "The value must be greater than or equal to 0");
    m_Age = value;
}
}

```

اگرچه این یک مثال ساده است، اما شما می‌توانید فایده فراوانی که کپسوله کردن فیلدهای داده‌ای دارد را ببینید. همچنین می‌توانید ببینید چقدر ساده است که ویژگی‌های فقط خواندنی یا فقط نوشتنی بسازید. فقط کفایت یکی از متدهای دستیابی را پیاده‌سازی کنید. متناوباً، شما می‌توانید تنها به نوع‌های مشتق شده اجازه تغییر مقدار را بدهید. برای این کار متد **SetXxx** را **protected** کنید. کپسوله سازی داده به طریقی که نشان داده شد، دو اشکال دارد. اول اینکه، شما مجبورید کد بیشتری بنویسید چون می‌بایست متدهای بیشتری پیاده‌سازی کنید. دوم اینکه، کاربران نوع شما باید به جای ارجاع به نام یک فیلد، اکنون متدها را فراخوانی کنند.

```

e.SetName("Jeffrey Richter"); // updates the employee's name
String EmployeeName = e.GetName(); // retrieves the employee's name
e.SetAge(41); // updates the employee's age
e.SetAge(-5); // Throws ArgumentOutOfRangeException
Int32 EmployeeAge = e.GetAge(); // retrieves the employee's age

```

شخصاً، من فکر می‌کنم این اشکال‌ها خیلی کوچک هستند. با این وجود، زبان‌های برنامه‌نویسی و CLR مکانیزمی به نام ویژگی **properties** ارائه می‌کنند که اشکال اول را کاهش داده و اشکال دوم را کاملاً برطرف می‌کنند.

کلاس نشان داده شده در اینجا، از ویژگی‌ها استفاده می‌کند و شبیه کلاس قبلی کار می‌کند:

```

public sealed class Employee {
    private String m_Name;
    private Int32 m_Age;

    public String Name {
        get { return(m_Name); }
        set { m_Name = value; } // The 'value' keyword always identifies the new value.
    }

    public Int32 Age {
        get { return(m_Age); }
        set {
            if (value < 0) // The 'value' keyword always identifies the new value.
                throw new ArgumentOutOfRangeException("value", value.ToString(),
                    "The value must be greater than or equal to 0");
            m_Age = value;
        }
    }
}

```

}

همانگونه که می‌بینید، ویژگی‌ها کمی تعریف نوع را پیچیده می‌کنند اما در جبران آن، اجازه می‌دهند کد خود را به شکل زیر بنویسید:

```
e.Name = "Jeffrey Richter"; // "sets" the employee name
String EmployeeName = e.Name; // "gets" the employee's name
e.Age = 41; // "sets" the employee's age
e.Age = -5; // Throws ArgumentOutOfRangeException
Int32 EmployeeAge = e.Age; // "gets" the employee's age
```

شما می‌توانید به ویژگی‌ها به عنوان فیلدهای هوشمند نگاه کنید: فیلدهایی با منطق اضافی در پشت آن‌ها. CLR از ویژگی‌های استاتیک، نمونه، خلاصه (abstract) و مجازی پشتیبانی می‌کند. به علاوه، ویژگی‌ها می‌توانند با هر تغییردهنده‌ی دسترس‌پذیری (که در فصل ۶ "مبانی نوع عضو" بحث شدند) علامت زده شوند و حتی درون یک رابط تعریف شوند (رابط‌ها در فصل ۱۳ "رابط‌ها" بحث می‌شوند).

هر ویژگی دارای یک نام و یک نوع (که نمی‌تواند **void** باشد) است. ویژگی‌ها غیر قابل سربرگذاری هستند (یعنی اینکه، دو ویژگی با نام یکسان دارای نوع متفاوت داشته باشید). وقتی شما یک ویژگی تعریف می‌کنید، نوعاً یک متد **get** و یک **set** تعیین می‌کنید. هر چند می‌توانید برای تعریف یک ویژگی فقط-خواندنی، متد **set** را حذف کنید و برای تعریف یک ویژگی فقط-نوشتنی، متد **get** را حذف نمایید.

همچنین بسیار رایج است که متدهای **get/set** از ویژگی، یک فیلد خصوصی تعریف شده در نوع را دستکاری کنند. این فیلد اغلب با عنوان **backing field** اطلاق می‌شود. هر چند، متدهای **get** و **set** مجبور نیستند به چنین فیلدی دسترسی پیدا کنند. برای نمونه، نوع **System.Threading.Thread**، یک ویژگی **Priority** ارائه می‌کند که مستقیماً با سیستم عامل در ارتباط است، شی **Thread**، فیلدی برای اولویت یک ترد نگهداری نمی‌کند. نمونه‌ی دیگر از ویژگی‌های بدون فیلد **backing**، ویژگی‌های فقط-خواندنی هستند که در زمان اجرا محاسبه می‌شوند – برای نمونه، طول یک آرایه یا محیط یک مستطیل، وقتی ارتفاع و عرض آن را داشته باشید. وقتی شما یک ویژگی تعریف می‌کنید، بسته به تعریف آن، کامپایلر دو یا سه مورد زیر را در اسمبلی مدیریت شده حاصل، قرار می‌دهد:

- یک متد که بیانگر متد دستیابی **get** برای ویژگی است. این متد، تنها اگر یک متد دستیابی **get** برای ویژگی تعریف شود، تولید می‌گردد.
- یک متد که بیانگر متد دستیابی **set** برای ویژگی است. این متد، تنها اگر یک متد دستیابی **set** برای ویژگی تعریف شود، تولید می‌گردد.
- یک تعریف ویژگی در متادیتای اسمبلی مدیریت شده. این مورد همواره تولید می‌شود.

به شی **Employee** که قبلاً نشان دادم برگردیم. وقتی کامپایلر، این نوع را کامپایل می‌کند، به ویژگی‌های **Name** و **Age** آن می‌رسد. چون هر دو ویژگی متدهای دستیابی **get** و **set** را دارند، کامپایلر چهار تعریف متد در نوع **Employee** قرار می‌دهد. آن مثل اینست که سورس کد اصلی اینگونه نوشته شده باشد:

```
public sealed class Employee {
    private String m_Name;
    private Int32 m_Age;

    public String get_Name(){
        return m_Name;
    }

    public void set_Name(String value) {
        m_Name = value; // The argument 'value' always identifies the new value.
    }

    public Int32 get_Age() {
        return m_Age;
    }

    public void set_Age(Int32 value) {
        if (value < 0) // The 'value' always identifies the new value.
```

```

        throw new ArgumentOutOfRangeException("value", value.ToString(),
            "The value must be greater than or equal to 0");
        m_Age = value;
    }
}

```

کامپایلر به صورت خودکار نام‌ها را برای این متدها با افزودن **get_** و **set_** به ابتدای نام ویژگی‌ها، تولید می‌کند. سی‌شارپ از ویژگی‌ها، پشتیبانی درونی می‌کند. وقتی کامپایلر سی‌شارپ کدی را می‌بیند که سعی در خواندن یا نوشتن به یک ویژگی دارد، در واقع کامپایلر یک فراخوانی به یکی از این متدها، تولید می‌کند. اگر شما از یک زبان برنامه‌نویسی که مستقیماً از ویژگی‌ها پشتیبانی نمی‌کند، استفاده می‌کنید، هنوز هم می‌توانید با فراخوانی متد دستیابی مورد نظر به ویژگی‌های دسترسی پیدا کنید. اثر، دقیقاً همان است، تنها سورس‌کد به زیبایی قبل نیست.

افزون بر تولید متدهای دستیابی، کامپایلر به ازای هر ویژگی تعریف شده در سورس‌کد، یک ورودی تعریف ویژگی در متادیتای اسمبلی مدیریت‌شده نیز قرار می‌دهد. این ورودی شامل چند پرچم و نوع ویژگی است و به متدهای دستیابی **get** و **set** اشاره می‌کند. این اطلاعات برای بیان همکاری میان مفهوم انتزاعی یک "ویژگی" و متدهای دستیابی آن است. کامپایلرها و دیگر ابزارها می‌توانند از این اطلاعات که از طریق کلاس **System.Reflection.PropertyInfo** قابل دسترسی است استفاده کنند. CLR از این اطلاعات متادیتا استفاده نمی‌کند و در زمان اجرا، تنها به متدهای دستیابی نیاز دارد.

ویژگی‌هایی که به صورت خودکار پیاده‌سازی می‌شوند

اگر شما یک ویژگی را برای کپسوله کردن یک فیلد **backing** استفاده می‌کنید، سی‌شارپ نحو ساده شده‌ای به نام ویژگی‌های پیاده‌سازی شده‌ی خودکار **automatically implemented properties (AIPs)** ارائه می‌کند، که آن را برای ویژگی **Name** در زیر می‌بینید:

```

public sealed class Employee {
    // This property is an automatically implemented property
    public String Name { get; set; }

    private Int32 m_Age;

    public Int32 Age {
        get { return(m_Age); }
        set {
            if (value < 0) // The 'value' keyword always identifies the new value.
                throw new ArgumentOutOfRangeException("value", value.ToString(),
                    "The value must be greater than or equal to 0");
            m_Age = value;
        }
    }
}

```

وقتی شما یک ویژگی تعریف می‌کنید و برای متدهای دستیابی **get/set** پیاده‌سازی را انجام ندهید، کامپایلر سی‌شارپ به صورت خودکار برای شما یک فیلد خصوصی تولید می‌کند. در این مثال، فیلد از نوع **String**، نوع ویژگی، خواهد بود. و کامپایلر، به صورت خودکار متدهای **get_Name** و **set_Name** را برای شما پیاده‌سازی می‌کند تا به ترتیب مقدار درون فیلد را برگرداند و بر مقدار درون فیلد بنویسد.

شاید شما تعجب کنید ارزش انجام چنین کاری در مقایسه با تعریف یک فیلد **public String** به نام **Name** چیست. خوب، تفاوت بزرگی وجود دارد. استفاده از نحو **AIP** یعنی اینکه شما یک ویژگی درست کرده‌اید. هر کدی که به این ویژگی دسترسی پیدا کند در واقع متدهای **get** و **set** را فراخوانی می‌کند. اگر بعداً تصمیم بگیرید به جای پیاده‌سازی پیش فرض کامپایلر، خودتان متدهای **get** و/یا **set** را پیاده‌سازی کنید، هر کدی که به ویژگی شما دسترسی دارد نیاز به کامپایلر مجدد پیدا نمی‌کند. اما اگر **Name** را به عنوان یک فیلد تعریف می‌کردید، بعداً تصمیم به تغییر آن به یک ویژگی می‌گرفتید؛ تمام کدهایی که به فیلد دسترسی داشتند، اکنون مجبور بودند برای دسترسی به متدهای ویژگی، مجدداً کامپایل شوند.

- شخصاً، من ویژگی **AIP** کامپایلر را نمی‌پسندم و معمولاً به دلایل زیر از آن خودداری می‌کنم: نحو تعریف یک فیلد می‌تواند شامل مقداردهی اولیه باشد به گونه‌ای که شما در یک خط از برنامه، هم فیلد را تعریف و هم آن را مقداردهی اولیه می‌کنید. اما نحو ساده‌ای برای تنظیم یک **AIP** به یک مقدار اولیه، وجود ندارد. پس، شما باید صریحاً هر **AIP** را در هر متد سازنده، مقداردهی اولیه کنید.
- سریالی‌کننده‌ی زمان اجرا، نام فیلد را در استریم سریالی شده، قرار می‌دهد. نام یک فیلد **backing**، برای یک **AIP** توسط کامپایلر تعیین می‌شود و با هر بار کامپایل کدتان، ممکن است نام این فیلد **backing** تغییر کند، که مانع از غیر سریالی شدن نمونه‌هایی از هر نوعی که یک **AIP** دارد، می‌شود. در هر نوعی که قصد دارید آن را سریالی و غیر سریالی کنید، از ویژگی **AIP** استفاده نکنید.
- هنگام خطایابی، شما نمی‌توانید روی متدهای **get** و **set** از یک **AIP**، نقاط توقف بگذارید. پس نمی‌توانید تعیین کنید چه هنگام یک برنامه این ویژگی را می‌خواند یا می‌نویسد. شما می‌توانید روی ویژگی‌هایی که دستی پیاده‌سازی کرده اید نقاط توقف بگذارید که هنگام ردیابی خطاها بسیار مفید است.

همچنین، باید بدانید وقتی از **AIP** استفاده می‌کنید، ویژگی باید قابل خواندن و قابل نوشتن باشد. یعنی کامپایلر باید هر دو متد **get** و **set** را تولید کند. این درست است چون یک فیلد فقط-نوشتنی، بدون قابلیت خواندن بی‌کاربرد است، همچنین یک فیلد فقط خواندنی همیشه مقدار پیشفرض خود را نگه می‌دارد. به علاوه، چون شما نام فیلد **backing** که کامپایلر تولید می‌کند را نمی‌دانید، کد شما همیشه باید با استفاده از نام ویژگی به ویژگی دسترسی داشته باشد. و اگر شما تصمیم بگیرید که صریحاً یکی از متدهای دستیابی را پیاده‌سازی کنید، می‌بایست صریحاً هر دو متد دستیابی را پیاده‌سازی نمایید و دیگر از ویژگی **AIP** استفاده نمی‌کنید. برای یک تک ویژگی، **AIP** یک معامله برای همه یا هیچی است.

تعریف هوشمندانه ویژگی‌ها

شخصاً، من ویژگی‌ها را دوست ندارم و آرزو داشتم که آن‌ها در دات‌نت فریم‌ورک میکروسافت و زبان‌های برنامه‌نویسی آن، پشتیبانی نمی‌شدند. دلیل من این است که ویژگی‌ها شبیه فیلد هستند، در حالیکه متد می‌باشند. این باعث سردرگمی زیادی شده است. وقتی یک برنامه‌نویس کدی را می‌بیند که به یک فیلد دسترسی دارد، چندین فرض را بر برنامه‌نویس لحاظ می‌کند که ممکن است برای یک ویژگی درست نباشد. برای نمونه،

- یک ویژگی ممکن است فقط-خواندنی یا فقط-نوشتنی باشد، دسترسی به فیلد همواره قابل نوشتن و قابل خواندن است. اگر یک ویژگی تعریف می‌کنید بهتر است هر دو متد دستیابی **get** و **set** را ارائه کند.
- متد یک ویژگی ممکن است یک اکسپشن تولید کند، دسترسی به فیلد هرگز اکسپشن تولید نمی‌کند.
- یک ویژگی نمی‌تواند به عنوان پارامتر **out** یا **ref** به یک متد ارسال شود، یک فیلد می‌تواند. برای نمونه، کد زیر کامپایل نمی‌شود:

```
using System;
```

```
public sealed class SomeType {
    private static String Name {
        get { return null; }
        set {}
    }

    static void MethodWithoutParam(out String n) { n = null; }

    public static void Main() {
        // For the line of code below, the C# compiler emits the following:
        // error CS0206: A property or indexer may not
        // be passed as an out or ref parameter
        MethodWithoutParam(out Name);
    }
}
```

- متد یک ویژگی می‌تواند زمان زیادی برای اجرا بگیرد، دسترسی به فیلد همیشه بلافاصله کامل می‌شود. یک دلیل رایج برای استفاده از ویژگی‌ها، انجام همزمانی تردهاست که می‌تواند ترد را برای همیشه متوقف کند و بنابراین اگر همزمانی ترد نیاز است، یک ویژگی نباید استفاده شود. در آن

وضعیت، یک متد ترجیح داده می‌شود. همچنین اگر کلاس شما از راه دور قابل دسترسی است (برای نمونه، کلاس شما از **System.MarshalByRefObject** مشتق شده است)، فراخوانی متد ویژگی بسیار کند خواهد بود و بنابراین متد بر ویژگی ترجیح داده می‌شود. به نظر من، کلاس‌هایی که از **MarshalByRefObject** مشتق می‌شوند هرگز نباید از ویژگی‌ها استفاده کنند.

- اگر در یک سطر، چندین بار فراخوانی شود، یک متد ویژگی ممکن است هر بار مقادیر متفاوتی برگرداند؛ یک فیلد هر بار مقدار یکسانی بر می‌گرداند. کلاس **System.DateTime** دارای یک ویژگی فقط خواندنی به نام **Now** است که تاریخ و ساعت کنونی را برمی‌گرداند. هر بار که به این ویژگی دسترسی پیدا می‌کنید، یک مقدار متفاوت برمی‌گرداند. این یک اشتباه است و مایکروسافت آرزو می‌کند که آن‌ها می‌توانستند با متد کردن **Now** به جای یک ویژگی این اشتباه را برطرف کنند. ویژگی **TickCount** از **Environment** نیز نمونه‌ی دیگری از این اشتباه است.

- متد یک ویژگی می‌تواند باعث اثرات جانبی قابل مشاهده شود؛ دسترسی به فیلد هرگز باعث این نمی‌شود. به بیان دیگر، کاربر یک نوع باید بتواند ویژگی‌های مختلف تعریف شده توسط نوع را به هر ترتیبی که می‌خواهد تنظیم کند بدون آنکه متوجه رفتار متفاوتی در نوع شود.

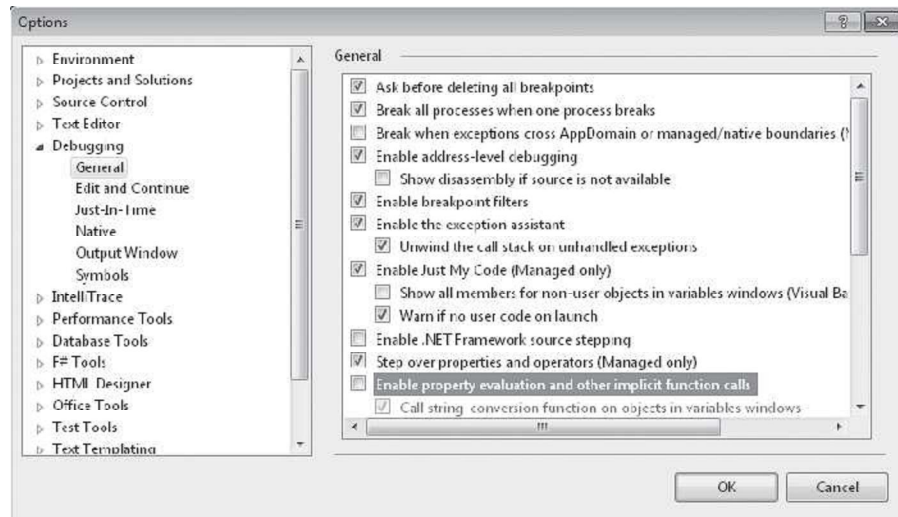
- متد یک ویژگی ممکن است به حافظه بیشتری نیاز داشته و یا اشاره‌گری به چیزی که واقعا بخشی از وضعیت شی نیست را برگرداند، پس تغییر در شی برگردانده شده اثری بر شی اصلی ندارد. خواندن یک فیلد همواره اشاره‌گری به یک شی برمی‌گرداند که به صورت تضمینی، بخشی از وضعیت شی اصلی است. کار با ویژگی‌هایی که یک کپی برمی‌گرداند می‌تواند بسیار گیج کننده باشد و این مشخصه اغلب مستند سازی نمی‌شود.

من توجه کرده‌ام که مردم از ویژگی‌ها بیش از آنچه که باید استفاده می‌کنند. اگر شما این لیست تفاوت‌ها بین ویژگی‌ها و فیلدها را بررسی کنید، می‌بینید تنها مورد خاصی است که تعریف یک ویژگی واقعا مفید بوده و باعث سردرگمی برنامه‌نویس نمی‌شود. تنها سودی که ویژگی‌ها برای شما دارند، نحو ساده‌تر آنهاست، در مقایسه با فراخوانی یک متد غیر ویژگی، مزیتی در اجرا ندارند و قابلیت درک کد هم کاهش می‌یابد. اگر من در طراحی دات‌نت فریمورک و کامپایلرها دخیل بودم، در کل، ویژگی‌ها را ارائه نمی‌کردم و به جای آن، برنامه‌نویسان متدهای **GetXxx** و **SetXxx** را طبق خواسته‌ی خود پیاده‌سازی می‌کردند. آنگاه اگر کامپایلرها می‌خواستند نحو ساده‌ای برای فراخوانی این متدها ارائه کنند، مشکلی وجود نداشت. اما من دوست دارم کامپایلر از نحوی متفاوت با نحو دسترسی به فیلد استفاده کند تا برنامه‌نویسان بدانند چه کار می‌کنند – فراخوانی یک متد.

ویژگی‌ها و دیباگر ویژوال استودیو

ویژوال استودیو به شما اجازه می‌دهد یک ویژگی را در پنجره‌ی **watch** دیباگر وارد کنید. وقتی این کار را می‌کنید، هر بار که به یک نقطه توقف می‌رسید، دیباگر، متد دستیابی **get** را فراخوانی می‌کند و مقدار برگشتی را نشان می‌دهد. این امر در ردیابی خطاها می‌تواند بسیار مفید باشد اما می‌تواند باعث خطاهایی نیز شود و سرعت خطایابی شما را کاهش دهد. برای نمونه، فرض کنید شما یک **FileStream** برای یک فایل در یک اشتراک شبکه ساخته‌اید و سپس ویژگی **Length** از **FileStream** را به پنجره **watch** افزوده‌اید. حال هر بار که به نقطه توقف می‌رسید، دیباگر، متد دستیابی **get** از **Length** را فراخوانی می‌کند، که در درون، یک درخواست در شبکه به سرور ارسال می‌شود و طول کنونی فایل را برمی‌گرداند!

به طریق مشابه، اگر متد دستیابی **get** از ویژگی شما، اثر جانبی داشته باشد، آنگاه این اثر جانبی هر بار که به نقطه توقف برسید، اجرا می‌شود. برای نمونه، فرض کنید که متد دستیابی **get** از ویژگی شما، هربار یک شمارنده را افزایش می‌دهد، اکنون این شمارنده با هر بار رسیدن به نقطه توقف نیز افزوده می‌شود. به خاطر این مشکلات احتمالی، ویژوال استودیو به شما اجازه می‌دهد ارزیابی ویژگی را برای ویژگی‌های درون پنجره **watch** غیر فعال کنید. برای غیر فعال کردن ارزیابی ویژگی در ویژوال استودیو، منوی **Tools**، **Options**، **Debugging** و **General** را انتخاب کنید و در لیست نشان داده شده در شکل ۱۰-۱، تیک گزینه‌ی **Enable Property Evaluation And Other Implicit Function Calls** را بردارید. توجه کنید که حتی با برداشتن این گزینه، می‌توانید ویژگی را به پنجره‌ی **watch** ویژوال استودیو اضافه کنید و به صورت دستی، ویژوال استودیو را با کلیک بر دایره‌ی اجبار ارزیابی، مجبور کنید که آن را ارزیابی کند.



شکل ۱۰-۱ تنظیمات عمومی دیباگر ویژوال استودیو

مقداردهی کننده های شی و مجموعه

خیلی رایج است که یک شی بسازید و سپس برخی از ویژگی‌ها (یا فیلدها)ی عمومی شی را مقداردهی کنید. برای ساده سازی این الگوی رایج برنامه‌نویسی، زبان سی‌شارپ از نحو ویژه‌ای برای مقداردهی اولیه شی پشتیبانی می‌کند. نمونه‌ای را ببینیم:

```
Employee e = new Employee() { Name = "Jeff", Age = 45 };
```

با این یک بیان (statement)، من یک شی **Employee** می‌سازم، سازنده‌ی بدون پارامترش را فراخوانی می‌کنم و سپس ویژگی عمومی **Name** آن را به "Jeff" و ویژگی عمومی **Age** آن را به 45 مقداردهی می‌کنم. در واقع، کد بالا معادل کد زیر است که می‌توانید با بررسی کد IL برای هر دو تکه کد این را ببینید:

```
Employee e = new Employee();
e.Name = "Jeff";
e.Age = 45;
```

فایده‌ی اصلی در نحو مقداردهی کننده‌ی شی، اینست که به شما اجازه می‌دهد در متن یک عبارت (expression) (در مقابل متن یک بیان) کدنویسی کرده و توابع را ترکیب کنید که باعث افزایش خوانایی کند می‌شود. برای نمونه، اکنون من می‌توانم کد زیر را بنویسم:

```
String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

پس اینجا، در یک بیان، یک شی **Employee** ساخته‌ام، سازنده‌اش را فراخوانی کرده‌ام، دو ویژگی عمومی‌اش را مقداردهی اولیه نموده‌ام و سپس در عبارت حاصل، **ToString** را همراه با **ToUpper** روی آن فراخوانی کرده‌ام. برای اطلاعات بیشتر در مورد ترکیب توابع، بخش "متدهای گسترشی" در فصل ۸ "متدها" را ببینید.

به عنوان یک نکته جانبی و کوچک، سی‌شارپ همچنین اجازه می‌دهد پرانتزها را قبل از آکولات ({) حذف کنید در صورتیکه بخواهید یک سازنده‌ی بدون پارامتر را فراخوانی کنید. کد زیر همان IL ای را تولید می‌کند که کد قبلی:

```
String s = new Employee { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

اگر نوع یک ویژگی، رابط **IEnumerable** یا **IEnumerable<T>** را پیاده‌سازی کند، آنگاه ویژگی به عنوان یک مجموعه در نظر گرفته می‌شود و مقداردهی اولیه یک مجموعه در مقابل یک عمل جایگزینی، عملی اضافی است. برای نمونه فرض کنید من تعریف کلاس زیر را دارم:

```
public sealed class Classroom {
    private List<String> m_students = new List<String>();
    public List<String> Students { get { return m_students; } }

    public Classroom() {}
}
```

اکنون من می‌توانم کدی داشته باشم که یک شی **Classroom** بسازد و مجموعه‌ی **Students** را مثل زیر مقداردهی اولیه کند:

```
public static void M() {
    Classroom classroom = new Classroom {
        Students = { "Jeff", "Kristin", "Aidan", "Grant" }
    };

    // show the 4 students in the classroom
    foreach (var student in classroom.Students)
        Console.WriteLine(student);
}
```

هنگام کامپایل این کد، کامپایلر می‌بیند که ویژگی **Students** از نوع **List<String>** است و این نوع، رابط **IEnumerable<String>** را پیاده‌سازی می‌کند. اکنون، کامپایلر فرض می‌کند که نوع **List<String>** متدی به نام **Add** (چون اکثر کلاس‌های مجموعه در واقع یک متد **Add** که اقلام را به مجموعه می‌افزاید، دارند) را ارائه می‌کند. کامپایلر کدی تولید می‌کند که متد **Add** مجموعه را فراخوانی کند. پس کد نشان داده شده در بالا، توسط کامپایلر به کد زیر تبدیل می‌شود:

```
public static void M() {
    Classroom classroom = new Classroom();
    classroom.Students.Add("Jeff");
    classroom.Students.Add("Kristin");
    classroom.Students.Add("Aidan");
    classroom.Students.Add("Grant");
}
```



```
// Show the 4 students in the classroom
foreach (var student in classroom.Students)
    Console.WriteLine(student);
}
```

اگر نوع ویژگی، **IEnumerable** یا **IEnumerable<T>** را پیاده‌سازی کند اما مجموعه، یک متد **Add** ارائه نکند، آنگاه کامپایلر به شما اجازه نمی‌دهد که از نحو مقداردهی مجموعه برای افزودن ارقام به مجموعه استفاده کنید و در عوض کامپایلر پیامی شبیه به این تولید می‌کند:

“error CS0117: ‘System.Collections.Generic.IEnumerable<string> does not contain a definition for ‘Add’.”

متدهای **Add** در برخی مجموعه‌ها، چندین آرگومان می‌گیرند. برای نمونه متد **Add** از **Dictionary**:

```
public void Add(TKey key, TValue value);
```

شما می‌توانید چندین آرگومان به یک متد **Add** با استفاده از آکولات‌های تودرتو در مقداردهی کننده‌ی مجموعه، ارسال کنید:

```
var table = new Dictionary<String, Int32> {
    { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }
};
```

کد فوق معادل این کد است:

```
var table = new Dictionary<String, Int32>();
table.Add("Jeffrey", 1);
table.Add("Kristin", 2);
table.Add("Aidan", 3);
table.Add("Grant", 4);
```

نوع‌های ناشناس

ویژگی نوع‌های ناشناس **anonymous types** سی‌شارپ به شما اجازه می‌دهد به صورت خودکار یک نوع چندتایی تغییرناپذیر با استفاده از نحو بسیار ساده و روان ایجاد کنید. یک نوع چندتایی **tuple type**^{۴۸} نوعی است که شامل یک مجموعه از ویژگی‌های بهم مرتبط می‌باشد. در خط بالایی از کد زیر، من یک کلاس با دو ویژگی (**Name** از نوع **String** و **Year** از نوع **Int32**) تعریف کرده‌ام، یک نمونه از این نوع ساخته‌ام، و ویژگی **Name** آن را **“Jeff”** و ویژگی **Year** آن را **1964** قرار داده‌ام.

```
// Define a type, construct an instance of it, & initialize its properties
```

```
var o1 = new { Name = "Jeff", Year = 1964 };
```

```
// Display the properties on the console:
```

```
Console.WriteLine("Name={0}, Year={1}", o1.Name, o1.Year); // Displays: Name=Jeff, Year=1964
```

خط بالایی از کد، یک نوع ناشناس می‌سازد چون من نام نوعی بعد از کلمه کلیدی **new** تعیین نکرده‌ام. پس کامپایلر یک نام نوع به صورت خودکار برای من می‌سازد و به من نمی‌گوید آن چیست (که بدین دلیل است که به آن، نوع ناشناس می‌گویند). این خط از کد، از نحو مقداردهی شی که در بخش قبلی برای تعریف و مقداردهی اولیه ویژگی‌ها استفاده شد، بهره می‌برد. همچنین چون من (برنامه‌نویس) نام نوع را در زمان کامپایل نمی‌دانم، من نمی‌دانم نوع متغیر **o1** را چه چیزی بگذارم. هر چند، این مشکلی ایجاد نمی‌کند، تا زمانی که می‌توانم از ویژگی متغیر محلی با نوع ضمنی (**var**) سی‌شارپ که در فصل ۹ "پارامترها" بحث شد، استفاده کنم تا کامپایلر، نوع را از روی عبارت سمت راست عملگر انتساب (=) استنتاج کند.

حال بگذارید تمرکز کنیم که کامپایلر دقیقاً چه کاری می‌کند وقتی شما کدی شبیه به این می‌نویسید:

```
var o = new { property1 = expression1, ..., propertyN = expressionN };
```

کامپایلر، نوع هر عبارت را استنتاج می‌کند، از روی این نوع‌های استنتاج شده فیلدهای خصوصی می‌سازد، برای هر یک از فیلدها، ویژگی عمومی فقط-خواندنی ایجاد کرده و یک سازنده که تمام این عبارت‌ها را دریافت کند، ایجاد می‌نماید. کد سازنده، فیلدهای خصوصی فقط-خواندنی را از روی نتایج عبارت

^{۴۸} این واژه به عنوان یک انتزاع از "ذنباله" نشات می‌گیرد:

های ارسالی، مقداردهی اولیه می‌کند. علاوه بر این، کامپایلر، متدهای **ToString**، **GetHashCode** و **Equals** از **Object** را بازنویسی کرده و کد درون تمام این متدها را تولید می‌کند. در عمل، کلاسی که کامپایلر تولید می‌کند، شبیه به این است:

```
[CompilerGenerated]
internal sealed class <>f__AnonymousType0<...>: Object {
    private readonly t1 f1;
    public t1 p1 { get { return f1; } }

    ...

    private readonly tn fn;
    public tn pn { get { return fn; } }

    public <>f__AnonymousType0<...>(t1 a1, ..., tn an) {
        f1 = a1; ...; fn = an; // Set all fields
    }

    public override Boolean Equals(Object value) {
        // Return false if any fields don't match; else true
    }

    public override Int32 GetHashCode() {
        // Returns a hash code generated from each fields' hash code
    }

    public override String ToString() {
        // Return comma-separated set of property name = value pairs
    }
}
```

کامپایلر متدهای **Equals** و **GetHashCode** را تولید می‌کند تا نمونه‌های نوع ناشناس بتوانند در یک مجموعه جدول هش جای بگیرند. ویژگی‌ها به جای خواندن/نوشتنی، فقط-خواندنی هستند تا به عدم تغییر کد هش شی کمک کنند. تغییر کد هش برای یک شی که در یک جدول هش به عنوان کلید است می‌تواند مانع از یافتن شی شود. کامپایلر، متد **ToString** را تولید می‌کند تا هنگام خطایابی کمک کند. در دیباگر ویژوال استودیو، شما می‌توانید موس را بر روی یک متغیر که به یک نمونه از یک نوع ناشناس اشاره دارد قرار دهید و ویژوال استودیو متد **ToString** را فراخوانی می‌کند تا رشته نتیجه را در یک پنجره نمایش دهد. ضمناً، **IntelliSense** ویژوال استودیو هنگامی که در ویرایشگر می‌نویسید، نام ویژگی‌ها را پیشنهاد می‌کند - یک ویژگی (!) بسیار عالی. کامپایلر از دو نحو دیگر برای تعریف یک ویژگی درون یک نوع ناشناس، پشتیبانی می‌کند، جاییکه بتواند نام‌ها و نوع‌های ویژگی‌ها را از روی متغیرها استنتاج کند:

```
String Name = "Grant";
DateTime dt = DateTime.Now;

// Anonymous type with two properties
// 1. String Name property set to Grant
// 2. Int32 Year property set to the year inside the dt
var o2 = new { Name, dt.Year };
```

در این مثال، کامپایلر تعیین می‌کند که اولین ویژگی باید **Name** نامیده شود. چون **Name** نام یک متغیر محلی است، کامپایلر نوع ویژگی را یکسان با نوع متغیر محلی قرار می‌دهد: **String**. برای ویژگی دوم، کامپایلر از نام فیلد/ویژگی استفاده می‌کند: **Year.Year** یک ویژگی **Int32** از کلاس **DateTime** است پس ویژگی **Year** در نوع ناشناس نیز از یک **Int32** خواهد بود. اکنون، وقتی کامپایلر، یک نمونه از این نوع ناشناس می‌سازد، ویژگی

Name از نمونه را همان مقداری می‌گذارد که متغیر محلی **Name** دارد، پس ویژگی **Name** به همان شی رشته **"Grant"** اشاره می‌کند. کامپایلر، ویژگی **Year** را به همان مقدار برگشتی ویژگی **Year** از **dt** تنظیم می‌کند. کامپایلر در تعریف نوع‌های ناشناس بسیار هوشمند است. اگر کامپایلر ببیند که شما چندین نوع ناشناس در سورس‌کدتان تعریف می‌کنید که دارای ساختار یکسان هستند، کامپایلر تنها یک تعریف نوع ناشناس درست می‌کند و چندین نمونه از روی آن نوع می‌سازد. منظورم از "ساختار یکسان" این است که نوع‌های ناشناس باید نوع و نام یکسان برای هر یک از ویژگی‌ها داشته باشند و ترتیب این ویژگی‌های نیز یکی باشد. در نمونه کدهای بالا، نوع متغیر **o1** و نوع متغیر **o2** از یک نوع است. چون هر دو خط کد، یک نوع ناشناس با یک ویژگی **Name/String** و یک ویژگی **Year/Int32** تعریف می‌کنند و **Name** قبل از **Year** می‌آید.

چون دو متغیر از یک نوع هستند، به نتایج جالبی خواهیم رسید، مثل بررسی اینکه آیا دو شی، مقادیر یکسان دارند و انتساب یک ارجاع به یک شی به متغیر دیگر، همانند زیر:

```
// One type allows equality and assignment operations.
console.WriteLine("Objects are equal: " + o1.Equals(o2));
o1 = o2; // Assignment
```

همچنین، به خاطر برابری هویت آن‌ها، ما می‌توانیم یک آرایه با نوع ضمنی (که در بخش "مقداردهی اولیه عناصر آرایه" در فصل ۱۶ "آرایه‌ها" بحث می‌شوند) از نوع‌های ناشناس داشته باشیم:

```
// This works because all of the objects are of the same anonymous type
var people = new[] {
    o1, // From earlier in this section
    new { Name = "Kristin", Year = 1970 },
    new { Name = "Aidan", Year = 2003 },
    new { Name = "Grant", Year = 2008 }
};
```

```
// This shows how to walk through the array of anonymous types (var is required)
foreach (var person in people)
```

```
    Console.WriteLine("Person={0}, Year={1}", person.Name, person.Year);
```

نوع‌های ناشناس اغلب با تکنولوژی زبان یکپارچه پرس و جو (LINQ) استفاده می‌شوند، جاییکه شما یک پرس و جو را اجرا می‌کنید که به جواب یک مجموعه از اشیاء که همه نوع‌های ناشناس یکسان هستند، منجر می‌شود. سپس شما اشیاء درون مجموعه جواب را پردازش می‌کنید. تمام این‌ها در یک متد رخ می‌دهد. نمونه‌ای را می‌بینید که تمام فایل‌های دایرکتوری **MyDocument** که در طی هفت روز قبل تغییر کرده‌اند را نشان می‌دهد:

```
String myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var query =
    from pathname in Directory.GetFiles(myDocuments)
    let LastWriteTime = File.GetLastWriteTime(pathname)
    where LastWriteTime > (DateTime.Now - TimeSpan.FromDays(7))
    orderby LastWriteTime
    select new { Path = pathname, LastWriteTime }; // Set of anonymous type objects
```

```
foreach (var file in query)
```

```
    Console.WriteLine("LastWriteTime={0}, Path={1}", file.LastWriteTime, file.Path);
```

نمونه‌های نوع‌های ناشناس به بیرون از متد راه نمی‌یابند. یک متد نمی‌تواند یک پارامتر از یک نوع ناشناس دریافت کند چون راهی برای تعیین نوع ناشناس وجود ندارد. به طریق مشابه، یک متد نمی‌تواند بیان کند که یک اشاره‌گر به یک نوع ناشناس را برمی‌گرداند. در حالیکه می‌توان با یک نمونه از یک نوع ناشناس به عنوان یک **Object** برخورد کرد (چون تمام نوع‌های نمونه از **Object** مشتق شده‌اند)، راهی برای تبدیل یک شی **Object** به نوع ناشناس وجود ندارد چون شما نام نوع ناشناس را در زمان کامپایل نمی‌دانید. اگر می‌خواهید یک چندتایی را ارسال کنید، آنگاه باید نوع **System.Tuple** که در بخش بعدی بحث می‌شود را در نظر بگیرید.

نوع System.Tuple

در فضای نام **System**، مایکروسافت چندین نوع جنریک **Tuple** (که تماما از **Object** مشتق شده‌اند) ارائه می‌کند که از لحاظ **arity** (تعداد پارامترهای جنریک) متفاوتند. ساده‌ترین و پیچیده‌ترین آن‌ها اساسا شبیه به این هستند:

```
// This is the simplest:
[Serializable]
public class Tuple<T1> {
    private T1 m_Item1;
    public Tuple(T1 item1) { m_Item1 = item1; }
    public T1 Item1 { get { return m_Item1; } }
}

// This is the most complex:
[Serializable]
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> {
    private T1 m_Item1; private T2 m_Item2; private T3 m_Item3; private T4 m_Item4;
    private T5 m_Item5; private T6 m_Item6; private T7 m_Item7; private TRest m_Rest;

    public Tuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7,
        TRest t) {
        m_Item1 = item1; m_Item2 = item2; m_Item3 = item3; m_Item4 = item4;
        m_Item5 = item5; m_Item6 = item6; m_Item7 = item7; m_Rest = rest;
    }

    public T1 Item1 { get { return m_Item1; } }
    public T2 Item2 { get { return m_Item2; } }
    public T3 Item3 { get { return m_Item3; } }
    public T4 Item4 { get { return m_Item4; } }
    public T5 Item5 { get { return m_Item5; } }
    public T6 Item6 { get { return m_Item6; } }
    public T7 Item7 { get { return m_Item7; } }
    public TRest Rest { get { return m_Rest; } }
}
```

همانند نوع‌های ناشناس، وقتی یک **Tuple** ساخته می‌شود، تغییرناپذیر است (تمام ویژگی‌ها فقط-خواندنی هستند). من این را اینجا نشان ندادم، اما کلاس‌های **Tuple** متدهای **GetHashCode**، **Equals**، **CompareTo** و **ToString** به همراه یک ویژگی **Size** را ارائه می‌کنند. به علاوه، تمام نوع‌های **Tuple** رابط‌های **IStructuralComparable**، **IStructuralEquatable** و **IComparable** را ارائه می‌کنند تا شما بتوانید دو شیء **Tuple** را با همدیگر مقایسه کنید تا ببینید چگونه فیلدها با همدیگر مقایسه می‌شوند. برای اطلاعات بیشتر در مورد این متدها و رابط‌ها به **SDK** دات‌نت فریمورک مراجعه کنید.

نمونه‌ای از یک متد که از یک نوع **Tuple** برای برگرداندن اطلاعات به فراخوانی کننده استفاده می‌کند:

```
// Returns minimum in Item1 & maximum in Item2
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {
    return new Tuple<Int32, Int32>(Math.Min(a, b), Math.Max(a, b));
}

// This shows how to call the method and how to use the returned Tuple
private static void TupleTypes() {
```

```
var minmax = MinMax(6, 2);
Console.WriteLine("Min={0}, Max={1}", minmax.Item1, minmax.Item2); // Min=2, Max=6
}
```

البته خیلی مهم است که تولید کننده و مصرف کننده‌ی **Tuple** درک دقیقی از آنچه در ویژگی‌های **Item#** برمی‌گردد داشته باشد. در نوع‌های ناشناس، ویژگی‌ها، نام‌های واقعی بر اساس سورس کدی که نوع ناشناس را تعریف می‌کند، می‌گیرند. در نوع‌های **Tuple**، ویژگی‌ها توسط میکروسافت اسامی **Item#** می‌گیرند و شما اصلاً نمی‌توانید آن‌ها را تغییر دهید. متأسفانه نام‌ها، معنی یا اهمیت واقعی ندارند، پس بستگی به تولید کننده و مصرف کننده دارد که به آن‌ها معنی دهد. این همچنین، خوانایی کد و قابلیت نگهداری آن را کاهش می‌دهد و شما باید به سورس کدتان کامنت‌هایی اضافه کرده و آنچه تولید کننده/مصرف کننده درک می‌کند را توضیح دهید.

کامپایلر می‌تواند نوع‌های جنریک را تنها هنگام فراخوانی یک متد جنریک استنتاج کند، نه وقتی شما یک سازنده را فراخوانی می‌کنید. به همین دلیل، فضای نام **System** دارای یک کلاس استاتیک غیرعمومی **Tuple** نیز است که حاوی تعدادی متد **Create** می‌باشد که می‌تواند نوع‌های جنریک را از روی آرگومان‌ها استنتاج کند. در اینجا متد **MinMax** که قبلاً نشان داده شده را با کلاس استاتیک **Tuple** بازنویسی می‌کنیم:

```
// Returns minimum in Item1 & maximum in Item2
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {
    return Tuple.Create(Math.Min(a, b), Math.Max(a, b)); // Simpler syntax
}
```

اگر خواستید یک **Tuple** با بیش از ۸ آرگومان بسازید، آنگاه باید **Tuple** دیگری را به عنوان پارامتر **Rest** ارسال کنید:

```
var t = Tuple.Create(0, 1, 2, 3, 4, 5, 6, Tuple.Create(7, 8));
Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}",
    t.Item1, t.Item2, t.Item3, t.Item4, t.Item5, t.Item6, t.Item7,
    t.Rest.Item1.Item1, t.Rest.Item1.Item2);
```

مهم علاوه بر نوع‌های ناشناس و نوع‌های **Tuple**، شاید شما بخواهید به کلاس **System.Dynamic.ExpandoObject** (تعریف شده در اسمبلی **System.Core.dll**) نیز نگاهی بیاندازید. وقتی شما این کلاس را با نوع **dynamic** سی‌شارپ (که در فصل ۵ "نوع-های اصلی، ارجاعی و مقداری" بحث شد) به کار ببرید، راه دیگری برای گروه بندی یک مجموعه از ویژگی‌ها (جفت‌های کلید/مقدار) با همدیگر دارید. این روش در زمان کامپایل، نوع-امن نیست اما نحو آن خوب به نظر می‌رسد (اگرچه پشتیبان **IntelliSense** ندارد)، و شما می‌توانید اشیاء **ExpandoObject** را بین سی‌شارپ و زبان‌های پویا مثل **Python** ارسال کنید. نمونه کدی که از یک **ExpandoObject** استفاده می‌کند را می‌بینید:

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6; // Add an Int32 'x' property whose value is 6
e.y = "Jeff"; // Add a String 'y' property whose value is "Jeff"
e.z = null; // Add an Object 'z' property whose value is null

// See all the properties and their values:
foreach (var v in (IDictionary<String, Object>)e)
    Console.WriteLine("key={0}, v={1}", v.Key, v.Value);

// Remove the 'x' property and its value
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

ویژگی های پارامتردار

در بخش قبلی، متدهای دستیابی **get** برای ویژگی‌ها هیچ پارامتری دریافت نمی‌کردند، برای همین، من این ویژگی‌ها را ویژگی‌های بدون پارامتر نامیدم. این ویژگی‌ها براحتی درک می‌شوند چون حس دسترسی به یک فیلد را به دست می‌دهند. علاوه بر این ویژگی‌های شبیه به فیلد، زبان‌های برنامه‌نویسی از چیزی پشتیبانی می‌کنند که من آن‌ها را ویژگی‌های پارامتردار **parameterful properties** می‌نامم که متدهای دستیابی **get** آن‌ها می‌تواند یک یا بیشتر پارامتر دریافت کند و متدهای دستیابی **set** آن‌ها می‌تواند دو یا بیشتر پارامتر بگیرد. زبان‌های برنامه‌نویسی مختلف ویژگی‌های پارامتردار را به طرق مختلف پیاده‌سازی می‌کنند. همچنین، زبان‌ها از واژه‌های مختلف برای اشاره به ویژگی‌های پارامتردار استفاده می‌کنند. سی‌شارپ آن‌ها را **indexer** (ایندکسر) می‌نامد و ویژوال بیسیک آن‌ها را ویژگی‌های پیش فرض **default properties**. در این بخش، من بر این تمرکز می‌کنم که سی‌شارپ چگونه ایندکسر-هایش را با استفاده از ویژگی‌های پارامتردار ارائه می‌کند.

در سی‌شارپ، ویژگی‌های پارامتردار (ایندکسر) با استفاده از نحو شبیه به آرایه پیاده‌سازی می‌شوند. به بیان دیگر، شما می‌توانید به ایندکسر به عنوان راهی که یک برنامه‌نویس سی‌شارپ عملگر **[]** را سربارگذاری کند نگاه کنید. نمونه‌ای از یک کلاس **BitArray** که اجازه می‌دهد با نحو شبیه به آرایه به مجموعه بیت‌هایی که توسط یک نمونه از کلاس نگهداری می‌شود دسترسی پیدا کنید:

```
using System;
```

```
public sealed class BitArray {
    // Private array of bytes that hold the bits
    private Byte[] m_byteArray;
    private Int32 m_numBits;

    // Constructor that allocates the byte array and sets all bits to 0
    public BitArray(Int32 numBits) {
        // validate arguments first.
        if (numBits <= 0)
            throw new ArgumentOutOfRangeException("numBits must be > 0");

        // Save the number of bits.
        m_numBits = numBits;

        // Allocate the bytes for the bit array.
        m_byteArray = new Byte[(numBits + 7) / 8];
    }

    // This is the indexer (parameterful property).
    public Boolean this[Int32 bitPos] {

        // This is the indexer's get accessor method.
        get {
            // validate arguments first
            if ((bitPos < 0) || (bitPos >= m_numBits))
                throw new ArgumentOutOfRangeException("bitPos");
            // Return the state of the indexed bit.
            return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0;
        }

        // This is the indexer's set accessor method.
        set {
```

```

if ((bitPos < 0) || (bitPos >= m_numBits))
    throw new ArgumentOutOfRangeException("bitPos", bitPos.ToString());
if (value) {
    // Turn the indexed bit on.
    m_byteArray[bitPos / 8] = (Byte)
        (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
} else {
    // Turn the indexed bit off.
    m_byteArray[bitPos / 8] = (Byte)
        (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
}
}
}
}

```

استفاده از ایندکسر کلاس **BitArray** واقعا ساده است:

```

// Allocate a BitArray that can hold 14 bits.
BitArray ba = new BitArray(14);

// Turn all the even-numbered bits on by calling the set accessor.
for (Int32 x = 0; x < 14; x++) {
    ba[x] = (x % 2 == 0);
}

// Show the state of all the bits by calling the get accessor.
for (Int32 x = 0; x < 14; x++) {
    Console.WriteLine("Bit " + x + " is " + (ba[x] ? "on" : "off"));
}

```

در مثال **BitArray**، ایندکسر، یک پارامتر **Int32**، **bitPos** می‌گیرد. تمام ایندکسرها حداقل یک پارامتر دارند، اما می‌توانند بیشتر نیز داشته باشند. این پارامترها (به همراه نوع برگشتی) می‌توانند از هر نوعی باشند (به جز **void**). یک نمونه از یک ایندکسر که بیش از یک پارامتر می‌گیرد را می‌توان در کلاس **System.Drawing.Imaging.ColorMatrix** که با اسمبلی **System.Drawing.dll** عرضه می‌شود، یافت.

بسیار رایج است که برای یافتن مقادیر در یک آرایه، یک ایندکسر ساخت. در واقع نوع **System.Collections.Generic.Dictionary** یک ایندکسر ارائه می‌کند که یک کلید را گرفته و مقدار همراه با آن کلید را برمی‌گرداند. بر خلاف ویژگی‌های بدون پارامتر، یک نوع می‌تواند چندین ایندکسر سربارگذاری شده تعریف کند تا زمانی که امضاهای متفاوتی داشته باشند.

همانند متدهای دستیابی **set** از یک ویژگی بدون پارامتر، متد دستیابی **set** از یک ایندکسر، نیز دارای یک پارامتر مخفی است که در سی‌شارپ **value** نامیده می‌شود. این پارامتر مقدار جدید مورد نظر برای "عصر ایندکس شده" را تعیین می‌کند.

CLR بین ویژگی‌های بدون پارامتر و ویژگی‌های پارامتردار تفاوت قایل نمی‌شود؛ برای **CLR**، هر دوی آن‌ها تعدادی متد و مقداری متادیتای تعریف شده در نوع هستند. همانگونه که قبلا گفته شد، زبان‌های برنامه‌نویسی مختلف نیاز به نحو متفاوتی در ساخت و استفاده از ویژگی‌های پارامتردار دارند. حقیقت اینکه سی‌شارپ نیاز به **this[...]** به عنوان نحو بیان یک ایندکسر دارد تماما تصمیمی است که توسط تیم سی‌شارپ گرفته شده است. معنی این انتخاب اینست که سی‌شارپ اجازه می‌دهد که ایندکسر تنها روی نمونه‌هایی از اشیاء ساخته شود. سی‌شارپ نحوی ارائه نمی‌کند که به یک برنامه‌نویس اجازه دهد یک ویژگی ایندکسر استاتیک تعریف کند گرچه **CLR** از ویژگی‌های پارامتردار استاتیک پشتیبانی می‌نماید.

چون **CLR** با ویژگی‌های پارامتردار همانند ویژگی‌های بدون پارامتر برخورد می‌کند، کامپایلر دو یا سه مورد زیر را در اسمبلی مدیریت‌شده‌ی خروجی قرار می‌دهد:

- یک متد که بیانگر متد دستیابی **get** از ویژگی پارامتردار است. این متد تنها اگر شما یک متد دستیابی **get** برای ویژگی تعریف کنید، تولید می‌شود.

- یک متد که بیانگر متد دستیابی **set** از ویژگی پارامتردار است. این متد تنها اگر شما یک متد دستیابی **set** برای ویژگی تعریف کنید، تولید می‌شود.
- یک تعریف ویژگی در متادیتای اسمبلی مدیریت شده که همواره تولید می‌شود. هیچ جدول تعریف متادیتای خاصی برای ویژگی پارامتردار وجود ندارد چون برای CLR ویژگی‌های پارامتردار شبیه ویژگی‌ها هستند.

برای کلاس **BitArray** که قبلاً نشان داده شد، کامپایلر، ایندکسر را به گونه‌ای کامپایل می‌کند که گویا سورس کد اینگونه نوشته شده است:

```
public sealed class BitArray {

    // This is the indexer's get accessor method.
    public Boolean get_Item(Int32 bitPos) { /* ... */ }

    // This is the indexer's set accessor method.
    public void set_Item(Int32 bitPos, Boolean value) { /* ... */ }
}
```

کامپایلر به صورت خودکار نام‌ها را برای این متدها با افزودن **get_** و **set_** به ابتدای نام ایندکسر درست می‌کند چون نحو سی‌شارپ به برنامه‌نویس اجازه‌ی تعیین یک نام ایندکسر را نمی‌دهد، تیم کامپایلر سی‌شارپ مجبور بودند یک نام پیش فرض برای متدهای دستیابی انتخاب کنند. آن‌ها **Item** را انتخاب کردند. بنابراین، نام متدهای تولید شده توسط کامپایلر، **get_Item** و **set_Item** است. هنگام بررسی مستندات مرجع دات‌نت فریمورک، شما می‌توانید (با جستجوی یک ویژگی به نام **Item**) بگویید که آیا یک نوع، یک ایندکسر ارائه می‌کند یا نه. برای نمونه نوع **System.Collections.Generic.List** یک ویژگی نمونه عمومی به نام **Item** ارائه می‌کند. این ویژگی، ایندکسر **List** است.

شما وقتی در سی‌شارپ کد می‌زنید، هرگز نام **Item** را نمی‌بینید، پس در حالت عادی اهمیت نمی‌دهید که کامپایلر این نام را برای شما انتخاب کرده است. اما اگر شما یک ایندکسر طراحی می‌کنید و یک نوع که در زبان برنامه‌نویسی دیگری نوشته شده، به آن دسترسی دارد، شاید بخواهید نام پیش فرض، **Item**، که به متدهای دستیابی **set** و **get** ایندکسر شما داده شده است را تغییر دهید. کد زیر نحوه‌ی انجام این کار را نمایش می‌دهد:

```
using System;
using System.Runtime.CompilerServices;

public sealed class BitArray {

    [IndexerName("Bit")]
    public Boolean this[Int32 bitPos] {
        // At least one accessor method is defined here
    }
}
```

اکنون کامپایلر متد **get_Bit** و **set_Bit** را به جای **get_Item** و **set_Item** تولید می‌کند. هنگام کامپایل، کامپایلر سی‌شارپ صفت **IndexerName** را می‌بیند و این به کامپایلر می‌گوید که چگونه متدها و متادیتای ویژگی را نام گذاری کند. خود صفت در متادیتای اسمبلی قرار نمی‌گیرد.^{۴۹}

کد ویژوال بیسیک زیر نحوه‌ی دسترسی به این ایندکسر سی‌شارپ را نشان می‌دهد:

```
' Construct an instance of the BitArray type.
Dim ba as New BitArray(10)

' Visual Basic uses () instead of [] to specify array elements.
Console.WriteLine(ba(2)) ' Displays True or False

' Visual Basic also allows you to access the indexer by its name.
```

^{۴۹} به همین دلیل، کلاس **IndexerNameAttribute** بخشی از استاندارد سازی ECMA از CLR و زبان سی‌شارپ نیست.


```
Console.WriteLine(ba.Bit(2)) ' Displays same as previous line
```

در سی‌شارپ، یک تک نوع می‌تواند چندین ایندکسر تعریف کند تا زمانی که تمام ایندکسرها، مجموعه پارامترهای متفاوتی می‌گیرند. در زبان‌های برنامه‌نویسی دیگر، صفت **IndexerName** به شما اجازه می‌دهد چندین ایندکسر با امضای یکسان تعریف کنید چون هر ایندکسر نام متفاوتی دارد. اینکه سی‌شارپ اجازه‌ی انجام چنین کاری را نمی‌دهد به این خاطر است که نحو آن، به ایندکسر از طریق نام اشاره نمی‌کند و کامپایلر نخواهد دانست که شما به کدام ایندکسر اشاره دارید. سعی در کامپایل کد سی‌شارپ زیر باعث می‌شود کامپایلر این پیام را تولید کند:

"error CS0111: Type 'SomeType' already defines a member call 'this' with the same parameter types."

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {

    // Define a get_Item accessor method.
    public Int32 this[Boolean b] {
        get { return 0; }
    }

    // Define a get_Jeff accessor method.
    [IndexerName("Jeff")]
    public String this[Boolean b] {
        get { return null; }
    }
}
```

شما می‌توانید به وضوح ببینید که سی‌شارپ به ایندکسرها به عنوان راهی برای سربارگذاری عملگر [] نگاه می‌کند و این عملگر نمی‌تواند برای مبهم ساختن ویژگی‌های پارامتردار با نام‌های مختلف و مجموعه پارامترهای یکسان استفاده شود.

اتفاقاً، نوع **System.String** یک نمونه از نوعی است که نام ایندکسرش را تغییر داده است. نام ایندکسر **String**، به جای **Chars.Item** است. این ویژگی فقط-خواندنی به شما اجازه می‌دهد که به یک تک کاراکتر در رشته دسترسی داشته باشید. برای زبان‌های برنامه‌نویسی که از عملگر [] برای دسترسی به این ویژگی استفاده نمی‌کنند، **Chars** نام معنادارتری است.

انتخاب ویژگی پارامتردار اصلی

محدودیت‌های سی‌شارپ در رابطه با ایندکسرها، دو سوال زیر را در پی دارد:

- اگر یک نوع در یک زبان برنامه‌نویسی که اجازه‌ی تعریف چندین ویژگی پارامتردار را می‌دهد تعریف شود، چه رخ می‌دهد؟
- چگونه این نوع می‌تواند توسط سی‌شارپ استفاده شود؟

پاسخ هر دو سوال اینست که یک نوع باید یکی از نام‌های ویژگی‌های پارامتردارش را به عنوان ویژگی پیش فرض تعیین کند. برای این کار باید یک نمونه از کلاس `System.Reflection.DefaultMemberAttribute` را بر خودش اعمال نماید. `DefaultMemberAttribute` می‌تواند به یک کلاس، یک ساختار یا یک رابط اعمال شود. در سی‌شارپ وقتی شما یک نوع که یک ویژگی پارامتردار را تعریف می‌کند، کامپایل می‌کنید، کامپایلر به صورت خودکار یک نمونه از صفت `DefaultMember` را بر نوع تعریف کننده اعمال می‌کند و وقتی شما از صفت `IndexerName` استفاده می‌کنید آن را به کار می‌برد. سازنده‌ی این صفت، نامی را تعیین می‌کند که می‌بایست به عنوان ویژگی پارامتردار پیش فرض نوع، استفاده شود.

پس، در سی‌شارپ، اگر شما یک نوع که دارای ویژگی پارامتردار است تعریف کنید و صفت `IndexerName` را تعیین نکنید، نوع تعریف کننده، یک صفت `DefaultMember` دارد که `Item` را نشان می‌دهد. اگر شما صفت `IndexerName` را به یک ویژگی پارامتردار اعمال کنید، نوع تعریف کننده یک صفت `DefaultMember` دارد که نام تعیین شده در صفت `IndexerName` را نشان می‌دهد. به خاطر بسپارید، سی‌شارپ کدی که شامل ویژگی‌های پارامتردار با چند نام مختلف باشد را کامپایل نخواهد کرد.

برای یک زبان که از چندین ویژگی پارامتردار پشتیبانی می‌کند، یکی از نام‌های متد ویژگی باید انتخاب و به عنوان صفت `DefaultMember` نوع، شناسایی شود. این تنها ویژگی پارامترداری است که سی‌شارپ می‌تواند به آن دسترسی داشته باشد.

وقتی کامپایلر سی‌شارپ کدی را می‌بیند که سعی در خواندن یا نوشتن به یک ایندکسر دارد، کامپایلر در واقع یک فراخوانی به یکی از این متدها تولید می‌کند. برخی زبان‌های برنامه‌نویسی ممکن است ویژگی‌های پارامتردار را پشتیبانی نکنند. برای دسترسی به ویژگی‌های پارامتردار از یکی از این زبان‌ها، شما باید متد دستیابی مورد نظر را صریحاً فراخوانی کنید. برای CLR، تفاوتی بین ویژگی‌های بدون پارامتر و ویژگی‌های پارامتردار وجود ندارد، پس شما از همان کلاس `System.Reflection.PropertyInfo` برای یافتن رابطه بین یک ویژگی پارامتردار و متدهای دستیابی‌اش استفاده می‌کنید.

کارایی در فرخوانی متدهای دستیابی ویژگی

برای متدهای دستیابی `get` و `set` ساده، کامپایلر JIT کد را خطی (inline) می‌کند، پس استفاده از ویژگی‌ها در مقابل استفاده از فیلدها اثر بدی بر سرعت اجرا نخواهد داشت. خطی کردن وقتی رخ می‌دهد که کد یک متد (یا متد دستیابی در این مورد) مستقیماً درون متدی که فراخوانی را انجام داده، کامپایل شود. این کار سرباری که با فراخوانی در زمان اجرا همراه است را در قبال بزرگتر کردن کد متد، حذف می‌کند. چون متدهای دستیابی ویژگی اغلب کد بسیار کوچکی دارند، خطی کردن آنها، کد اصلی را کوچک تر کرده و اجرای آن را تسریع می‌بخشد.

توجه کنید که کامپایلر JIT هنگام خطایابی کد، متدهای دستیابی ویژگی را خطی نمی‌کند چون خطایابی کد خطی شده مشکل تر است. این یعنی کارایی دسترسی به یک ویژگی در ساخت نهایی سریع و در ساخت خطایابی آهسته است. دسترسی به فیلد در هر دو ساخت خطایابی و نهایی سریع است.

دسترس پذیری متدهای دستیابی ویژگی

به ندرت، هنگام طراحی یک نوع، پیش می‌آید که بخواهیم یک دسترس پذیری برای یک متد دستیابی `get` و یک دسترس پذیری متفاوت برای یک متد دستیابی `set` داشته باشیم. رایج ترین سناریو، داشتن یک دستیابی `get` عمومی و یک دستیابی `set` محافظت شده است:

```
public class SomeType {
    private String m_name;
    public String Name {
        get { return m_name; }
        protected set {m_name = value; }
    }
}
```

همانگونه که در کد فوق می‌بینید، خود ویژگی **Name** به عنوان یک ویژگی **public** تعریف شده و این یعنی متد دستیابی **get** عمومی بوده و توسط هر کدی قابل فراخوانی است. هرچند، توجه داشته باشید که دستیابی **set** به عنوان **protected** تعریف شده و فقط توسط کدی تعریف شده درون **SomeType** یا توسط کدی در یک کلاس مشتق شده از **SomeType** قابل فراخوانی است.

هنگام تعریف یک ویژگی با متدهای دستیابی که دسترس پذیری متفاوتی دارند، نحو سی‌شارپ نیاز دارد که خود ویژگی به دسترس پذیری با کمترین محدودیت تعریف شده باشد و دسترس پذیری محدودتر به یکی از متدهای دستیابی اعمال گردد. در مثال فوق، ویژگی، **public** است و دستیابی **set** **protected** (محدودتر از **public**) است.

متدهای جنریک دستیابی ویژگی

چون ویژگی‌ها فقط متدهایی هستند، و چون سی‌شارپ و CLR اجازه می‌دهند متدها جنریک باشند، گاهی مردم می‌خواهند ویژگی‌هایی تعریف کنند که پارامترهای نوع جنریک خودشان (در مقابل استفاده از پارامترهای نوع جنریک از نوع) را معرفی کنند. هرچند سی‌شارپ اجازه‌ی چنین کاری را نمی‌دهد. دلیل اصلی اینکه چرا ویژگی‌ها نمی‌توانند پارامترهای نوع جنریک خودشان را معرفی کنند اینست که از لحاظ مفهومی، معنایی ندارد. یک ویژگی برای بیان یک مشخصه از یک شی است که می‌تواند خوانده یا نوشته شود. معرفی یک پارامتر نوع جنریک به این معنی است که رفتار خواندن/نوشتن می‌تواند تغییر کند، اما از لحاظ مفهومی، به نظر نمی‌رسد که یک ویژگی دارای رفتار باشد. اگر شما بخواهید که شی تان رفتاری را ارائه کند – جنریک یا غیر جنریک – به جای یک ویژگی، یک متد تعریف کنید.

فصل ۱۱: رویدادها

در این فصل، من در مورد آخرین گونه از اعضای که یک نوع می‌تواند تعریف کند، صحبت می‌کنم: رویداد. یک نوع که یک عضو رویداد تعریف می‌کند به نوع (یا نمونه‌های نوع) اجازه می‌دهد دیگر اشیاء را مطلع کند که چیز خاصی اتفاق افتاده است. برای نمونه، کلاس **Button** یک رویداد به نام **Click** ارائه می‌کند. وقتی شی **Button** کلیک می‌شود، یک یا بیشتر اشیاء در یک برنامه شاید بخواهند اطلاعی در مورد این رویداد دریافت کنند تا عملی را انجام دهند. رویدادها اعضای هستند که این تبدلات را فراهم می‌کنند. به خصوص، تعریف یک عضو رویداد یعنی اینکه یک نوع، قابلیت‌های زیر را ارائه می‌کند:

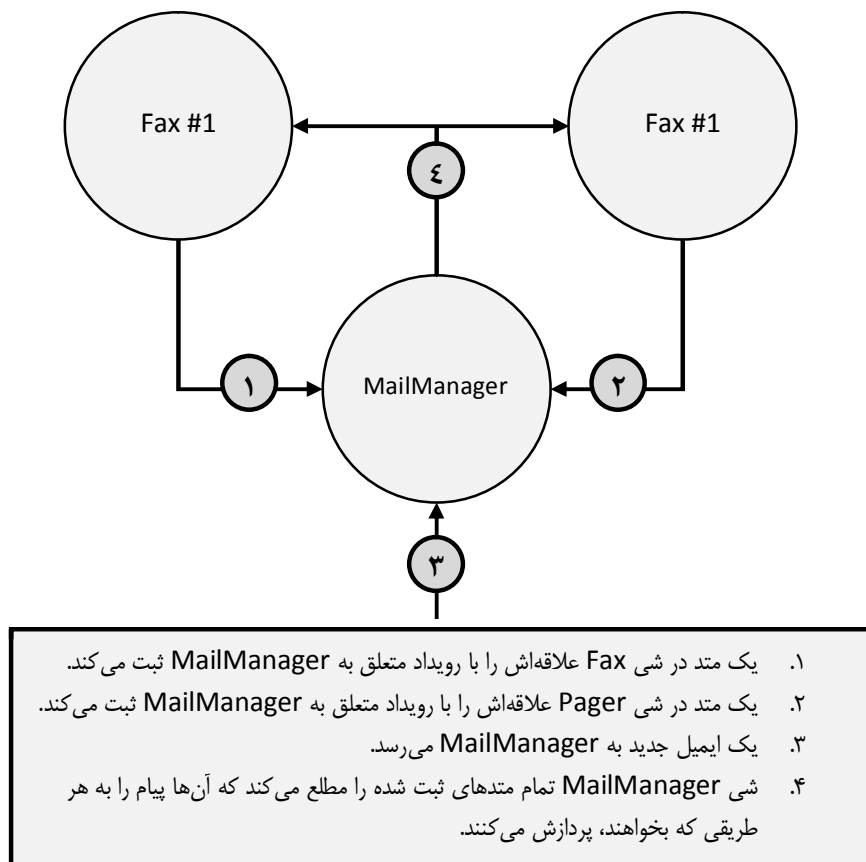
- یک متد می‌تواند علاقه‌اش به رویداد را ثبت کند.
- یک متد می‌تواند علاقه‌اش به رویداد را حذف کند.
- متدهای ثبت شده هنگامی که رویداد رخ می‌دهد، مطلع می‌شوند.

نوع‌ها هنگام تعریف یک رویداد می‌توانند این کاربرد را ارائه کنند چون لیستی از متدهای ثبت شده را نگهداری می‌کنند. وقتی رویداد رخ می‌دهد، نوع، تمام متدهای ثبت شده در مجموعه را مطلع می‌کند. مدل رویداد **CLR** بر اساس نماینده‌ها **delegates** است. یک نماینده یک روش نوع-امن برای فراخوانی یک متد کالباک^{۵۰} است. متدهای کالباک وسیله‌هایی هستند که به کمک آن‌ها اشیاء، خبر (اطلاع)‌هایی که مشترک آن هستند را دریافت می‌کنند. در این فصل، من از نماینده‌ها استفاده می‌کنم ولی توضیح کامل آن‌ها را تا فصل ۱۷ "نماینده‌ها" ارائه نمی‌دهم.

برای آنکه شما کاملاً درک کنید چگونه رویدادها درون **CLR** کار می‌کنند، من با یک سناریو که در آن رویدادها کاربرد دارند شروع می‌کنم. تصور کنید می‌خواهید یک برنامه ایمیل طراحی کنید. وقتی ایمیلی دریافت می‌شود، شاید کاربر بخواهد پیام را به یک ماشین فکس یا یک پیجر ارسال کند. در معماری این برنامه، بگوییم که شما ابتدا یک نوع به نام **MailManager** طراحی می‌کنید که پیام‌های رسیده را دریافت می‌کند. **MailManager** رویدادی به نام **NewMail** را ارائه می‌کند. دیگر نوع‌ها (مثل **Fax** و **Pager**) ممکن است علاقه‌شان به این رویداد را ثبت کنند. وقتی **MailManager** ایمیل جدیدی دریافت می‌کند، رویداد را فعال کرده، باعث می‌شود پیام بین هر یک از اشیاء ثبت شده توزیع شود. هر شی می‌تواند پیام را به هر نحوی که دوست دارد پردازش کند.

وقتی برنامه مقاردهی اولیه می‌شود، بگذارید فقط یک نمونه از **MailManager** را نمونه سازی کنیم – برنامه می‌تواند هر تعدادی از روی نوع‌های **Fax** و **Pager** نمونه سازی کند. شکل ۱-۱۱ نشان می‌دهد چگونه برنامه مقاردهی اولیه شده و هنگامی که ایمیل جدید می‌آید چه رخ می‌دهد.

^{۵۰} به دلیل نبود معادل مناسب برای **callback** از لفظ آن استفاده می‌کنم.



شکل ۱-۱۱ معماری یک برنامه برای استفاده از رویدادها

نحوه‌ی کار برنامه‌ای که در شکل ۱-۱۱ نمایش داده شد، اینگونه است: برنامه با ساختن یک نمونه از **MailManager** مقداردهی اولیه می‌شود. **MailManager** یک رویداد **NewMail** ارائه می‌کند. وقتی اشیاء **Fax** و **Pager** ساخته می‌شوند، یک متد نمونه را با رویداد **NewMail** از **MailManager** ثبت می‌کنند تا **MailManager** بداند هنگام رسیدن ایمیل جدید، اشیاء **Fax** و **Pager** را مطلع کند. اکنون، وقتی **MailManager** یک ایمیل جدید دریافت می‌کند (زمانی در آینده) رویداد **NewMail** را فعال می‌کند و به تمام متدهای ثبت شده فرصت پردازش پیام جدید به طریق دلخواه را می‌دهد.

طراحی یک نوع که یک رویداد را ارائه می‌کند

مراحل زیادی وجود دارد که یک برنامه‌نویس باید طی کند تا نوعی را تعریف نماید که یک یا بیشتر عضو رویداد ارائه کند. در این بخش، من در طی هر کدام از این مراحل حرکت می‌کنم. نمونه برنامه **MailManager** (که از سایت <http://wintellect.com> قابل دانلود است) تمام سورس کد نوع **MailManager**، نوع **Fax** و نوع **Pager** را نشان می‌دهد. شما متوجه خواهید شده که نوع **Pager** عملاً معادل نوع **Fax** است.

قدم اول: تعریف نوعی که اطلاعات اضافی که می‌بایست به دریافت کننده‌ی خبر رویداد، ارسال شود را نگهداری می‌کند

وقتی رویدادی فعال می‌شود، شی‌ای که رویداد را فعال کرده است شاید بخواهد اطلاعات اضافی را به اشیاء دریافت کننده‌ی خبر رویداد، ارسال کند. این اطلاعات اضافی باید در کلاس مخصوصی کپسوله شوند که معمولاً شامل تعدادی فیلد خصوصی همراه با تعدادی ویژگی فقط-خواندنی عمومی برای دسترسی به فیلدها می‌باشد. طبق قرارداد، کلاس‌هایی که اطلاعات رویداد را جهت ارسال به مدیریت کننده‌ی رویداد، نگهداری می‌کنند باید از **System.EventArgs** مشتق شوند و نام کلاس باید با پسوند **EventArgs** همراه شود. در این مثال، کلاس **NewMailEventArgs** فیلدهایی برای شناسایی ارسال کننده ایمیل (**m_from**)، دریافت کننده ایمیل (**m_to**) و موضوع ایمیل (**m_subject**) دارد.

```
// Step #1: Define a type that will hold any additional information that
```

```
// should be sent to receivers of the event notification
internal class NewMailEventArgs : EventArgs {

    private readonly String m_from, m_to, m_subject;

    public NewMailEventArgs(String from, String to, String subject) {
        m_from = from; m_to = to; m_subject = subject;
    }

    public String From { get { return m_from; } }
    public String To { get { return m_to; } }
    public String Subject { get { return m_subject; } }
}
}
```

نکته کلاس **EventArgs** در کتابخانه کلاس دات‌نت فریم‌ورک (FCL) تعریف و شبیه به زیر پیاده سازی شده است:

```
[ComVisible(true), Serializable]
public class EventArgs {
    public static readonly EventArgs Empty = new EventArgs();
    public EventArgs() { }
}
}
```

همانگونه که می‌بینید، این نوع چیز خاصی برای گفتن ندارد. تنها به عنوان یک کلاس پایه که دیگر نوع‌ها از آن مشتق شوند عمل می‌کند. بسیاری از رویدادها اطلاعات اضافی برای ارسال ندارند. برای نمونه، وقتی یک **Button** دریافت کننده‌های ثبت شده‌اش را از کلیک شدن خود خبر می‌کند، فراخوانی متد کالیک کافیسست. وقتی شما رویدادی تعریف می‌کنید که هیچ اطلاعات اضافی برای ارسال ندارد، به جای ساخت یک شی **EventArgs** جدید فقط از **EventArgs.Empty** استفاده کنید.

قدم دوم: تعریف عضو رویداد

یک عضو رویداد به کمک کلمه کلیدی **event** در سی‌شارپ تعریف می‌شود. هر عضو رویداد دارای دسترس پذیری (که اغلب **public** می‌باشد تا دیگر کدها بتوانند بدان دسترسی داشته باشند)، یک نوع نماینده که فرم کلی متدهایی که فراخوانی می‌شوند را تعیین می‌کند، و یک نام (که هر شناسه‌ی مجاز می‌تواند باشد) می‌باشد. عضو رویداد در کلاس **MailManager** بدین شکل است:

```
internal class MailManager {

    // Step #2: Define the event member
    public event EventHandler<NewMailEventArgs> NewMail;
    ...
}
}
```

NewMail نام این رویداد است. نوع عضو رویداد **EventHandler<NewMailEventArgs>** است که یعنی تمام دریافت کننده‌های خبر رویداد باید یک متد کالیک که فرم کلی آن با نوع نماینده **EventHandler<NewMailEventArgs>** مطابقت دارد را فراهم کنند. چون نماینده‌ی جنریک **System.EventHandler** به شکل زیر تعریف شده است:

```
public delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e)
    where TEventArgs: EventArgs;
```

فرم کلی متد باید شبیه به این باشد:

```
void MethodName(Object sender, NewMailEventArgs e);
```

نکته افراد زیادی تعجب می‌کنند چرا الگوی رویداد نیاز دارد که پارامتر **sender** همیشه از نوع **Object** باشد. گذشته از این، چون **MailManger** تنها نوعی است که یک رویداد با شی **NewMailEventArgs** را فعال می‌کند، به نظر می‌رسد بهتر است متد کالیک به فرم کلی زیر باشد:

```
void MethodName(MailManager sender, NewMailEventArgs e);
```

اینکه الگوی رویداد نیاز دارد که پارامتر **sender** از نوع **Object** باشد بیشتر به دلیل وراثت است. اگر **MailManager** به عنوان کلاس پایه برای **SmtplibMailManager** استفاده شود آنگاه چگونه؟ در این حالت، متد کالیک نیاز دارد که پارامتر **sender** به جای **MailManger** به فرم کلی **SmtplibMailManger** باشد، اما این نمی‌تواند اتفاق بیافتد چون **SmtplibMailManager** فقط رویداد **NewMail** را به ارث برده است. پس کدی که انتظار دارد **SmtplibMailManager** رویداد را فعال کند باز هم مجبور است که پارامتر **sender** را به **SmtplibMailManager** تبدیل کند. به بیان دیگر، باز هم تبدیل نیاز است. پس بهتر است که پارامتر **sender** از نوع **Object** باشد.

دلیل بعدی برای اینکه پارامتر **sender** از نوع **Object** باشد انعطاف پذیری است. این کار به نماینده اجازه می‌دهد که توسط نوع‌های مختلفی که رویدادی را تعریف می‌کنند که یک شی **NewMailEventArgs** ارسال می‌کند، استفاده شود. برای نمونه، یک کلاس **PopMailManager** می‌توانست از نماینده‌ی رویداد استفاده کند حتی اگر این کلاس از **MailManager** مشتق نشده باشد.

الگوی رویداد همچنین نیاز دارد که تعریف نماینده و متد کالیک، پارامتر از نوع مشتق شده از **EventArgs** را **e** بنامند. تنها دلیل این کار افزودن ثبات بیشتر به الگوست تا برنامه‌نویسان راحت تر الگو را یاد گرفته و پیاده سازی کنند. ابزارهایی که سورس کد تولید می‌کنند، (مثل ویژوال استودیو) هم می‌دادند که پارامتر را **e** بنامند.

سرانجام، الگوی رویداد نیاز دارد که تمام مدیریت کننده‌های رویداد، نوع برگشتی **void** داشته باشند. این لازم است چون فعال کردن یک رویداد ممکن است متدهای کالیک مختلفی را فراخوانی کند و راهی برای دریافت مقدار برگشتی از تمام این متدها وجود ندارد. داشتن نوع برگشتی **void** به متد کالیک اجازه‌ی برگرداندن مقداری را نمی‌دهد. متأسفانه، تعدادی مدیریت کننده‌ی رویداد در **FCL** وجود دارند، مثل **ResolveEventHandler** که از الگوی تعریف شده توسط مایکروسافت پیروی نمی‌کنند چون آن‌ها شی‌ای از نوع **Assembly** برمی‌گردانند.

قدم سوم: تعریف یک متد که مسئول فعال کردن رویداد جهت اطلاع به اشیاء ثبت شده درباره‌ی رخداد رویداد است

طبق قرارداد، کلاس باید یک متد مجازی و محافظت شده تعریف کند که توسط کد درون کلاس و کلاس‌های مشتق شده هنگام فعال شدن رویداد فراخوانی می‌شود. این متد یک پارامتر دریافت می‌کند، یک شی **NewMailEventArgs**. که شامل اطلاعات جهت ارسال به اشیایی است که خبر را دریافت می‌کنند. پیاده‌سازی پیش فرض این متد بررسی می‌کند آیا اشیایی علاقه‌شان به رویداد را ثبت کرده‌اند و اگر چنین است، رویداد فعال می‌شود و در نتیجه، متدهای ثبت شده را از رخداد رویداد مطلع می‌کند. متد درون کلاس **MailManager** شبیه به این است:

```
internal class MailManager {
    ...
    // Step #3: Define a method responsible for raising the event
    // to notify registered objects that the event has occurred
    // If this class is sealed, make this method private and nonvirtual
    protected virtual void OnNewMail(NewMailEventArgs e) {

        // Copy a reference to the delegate field now into a
        // temporary field for thread safety
```

```
EventHandler<EventArgs> temp =  
    Interlocked.CompareExchange(ref NewMail, null, null);  
  
    // If any methods registered interest with our event, notify them  
    if (temp != null) temp(this, e);  
}  
...  
}
```


فعال کردن یک رویداد به روش ترد-امن

وقتی اولین بار داتانت فریمورک عرضه شد، روش توصیه شده برای برنامه‌نویسان جهت فعال کردن یک رویداد با کدی شبیه به این بود:

```
// Version 1
protected virtual void OnNewMail(NewMailEventArgs e) {
    if (NewMail != null) NewMail(this, e);
}
```

مشکل با متد **OnNewMail** این بود که ترد می‌توانست ببیند که **NewMail**، **null** نیست و آنگاه، دقیقاً قبل از فراخوانی **NewMail**، ترد دیگری می‌توانست یک نماینده را از زنجیره حذف و **NewMail** را **null** کند، در نتیجه یک **NullReferenceException** تولید می‌شد. برای حل این رقابت، اکثر برنامه‌نویسان **OnNewMail** را به روش زیر می‌نوشتند:

```
// Version 2
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = NewMail;
    if (temp != null) temp(this, e);
}
```

روش تفکر در اینجا این بود که یک اشاره‌گر به **NewMail** درون یک متغیر موقتی **temp** کپی می‌شود که به زنجیره‌ی نماینده‌ها در لحظه‌ای که عملیات انتساب انجام شده اشاره دارد. حال، این متد، **temp** و **null** را مقایسه می‌کند و **temp** را فراخوانی می‌کند، پس دیگر مهم نیست که ترد دیگری **NewMail** را پس از انتساب به **temp** تغییر دهد. به خاطر داشته باشید آنچه اکثر برنامه‌نویسان درک نمی‌کنند اینست که این کد می‌تواند توسط کامپایلر بهینه‌سازی شده و متغیر محلی **temp** را کاملاً حذف کند. اگر این اتفاق رخ دهد، این نسخه کد معادل نسخه اول است، پس بروز یک **NullReferenceException** باز هم ممکن است.

برای حل واقعی این مشکل شما باید **OnNewMail** را اینگونه بنویسید:

```
// Version 3
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = Thread.VolatileRead(ref NewMail);
    if (temp != null) temp(this, e);
}
```

فراخوانی **Thread.VolatileRead**، **NewMail** را مجبور می‌کند درست در زمان فراخوانی خوانده شود و اشاره‌گر اکنون واقعا در متغیر **temp** کپی می‌شود. سپس، **temp** تنها در صورتی که **null** نباشد فراخوانی می‌گردد. متأسفانه، نوشتن کد به روش فوق میسر نیست چون یک سربارگذاری جنریک از متد **VolatileRead** وجود ندارد. هر چند، یک سربارگذاری جنریک از **Interlocked.CompareExchange** وجود دارد که شما می‌توانید استفاده کنید:

```
// Version 4
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp =
        Interlocked.CompareExchange(ref NewMail, null, null);
    if (temp != null) temp(this, e);
}
```

در اینجا، **CompareExchange**، اشاره‌گر **NewMail** را اگر **null** باشد تغییر می‌دهد و اگر **null** نباشد **NewMail** را دست نمی‌زند. به بیان دیگر، **CompareExchange** مقدار **NewMail** را اصلاً تغییر نمی‌دهد اما مقدار درون **NewMail** را به روش اتمی و ترد-امن برمی‌گرداند. برای اطلاعات بیشتر درباره‌ی **Thread.VolatileRead** و متدهای **Interlocked** و **CompareExchange**، فصل ۲۸ "ساخت‌های اصلی همزمانی ترد" را نگاه کنید.

در حالیکه چهارمین نسخه‌ی این کد بهترین از لحاظ فنی و صحیح‌ترین است، شما می‌توانید در واقع از نسخه‌ی دوم استفاده کنید چون کامپایلر JIT همیشه از این الگو آگاه است و می‌داند که متغیر محلی **temp** را بهینه نکند. مخصوصاً، تمام کامپایلرهای JIT مایکروسافت به این قاعده که خواندن‌های جدید برای حافظه هیپ، تولید نکنند احترام می‌گذارند. بنابراین کش کردن یک متغیر محلی این اطمینان را حاصل می‌کند که اشاره‌گر هیپ فقط یکبار دسترسی شده است. این نکته مستند سازی نشده است و در تئوری می‌تواند تغییر کند، به همین علت است که شما باید از نسخه‌ی چهارم استفاده کنید. اما در واقعیت، کامپایلر JIT مایکروسافت هرگز تغییری که این الگو را بر هم زند اعمال نمی‌کند، چون برنامه‌های بسیار زیادی از کار خواهد افتاد. افزون بر این، رویدادها اغلب در سناریوهای تک تردی (Silverlight، WPF، Windows Forms) استفاده می‌شوند و در نتیجه امنیت ترد مسئله نیست. بسیار مهم است که بدانید به خاطر همین رقابت (race condition) تردها، همچنین ممکن است یک متد پس از آنکه از زنجیره‌ی نماینده‌های رویداد حذف شد، باز هم فراخوانی شود.

طبق قاعده، شما می‌توانید یک متد گسترشی تعریف کنید (که در فصل ۸ "متدها" بحث شدند) که این منطق ترد-امن را کپسوله کند. یک متد گسترشی شبیه به این تعریف کنید:

```
public static class EventArgsExtensions {
    public static void Raise<TEventArgs>(this TEventArgs e,
        Object sender, ref EventHandler<TEventArgs> eventDelegate)
        where TEventArgs : EventArgs {

        // Copy a reference to the delegate field now into a
        // temporary field for thread safety
        EventHandler<TEventArgs> temp =
            Interlocked.CompareExchange(ref eventDelegate, null, null);

        // If any methods registered interest with our event, notify them
        if (temp != null) temp(sender, e);
    }
}
```

و اکنون، ما می‌توانیم متد **OnNewMail** را اینگونه بازنویسی کنیم:

```
protected virtual void OnNewMail(NewMailEventArgs e) {
    e.Raise(this, ref m_NewMail);
}
```

یک کلاس که از **MailManager** به عنوان نوع پایه استفاده می‌کند آزاد است متد **OnNewMail** را بازنویسی کند. این قابلیت به کلاس مشتق شده کنترل فعال کردن رویداد را می‌دهد. کلاس مشتق شده می‌تواند پیام ایمیل جدید را به هر روش که بخواهد، مدیریت کند. معمولاً، یک نوع مشتق شده، متد **OnNewMail** از کلاس پایه را فراخوانی می‌کند تا متد(های) ثبت شده از رخداد رویداد مطلع شوند. هر چند کلاس مشتق شده ممکن است اجازه‌ی ارسال رویداد را ندهد.

قدم چهارم: تعریف یک متد که ورودی را به رویداد مطلوب ترجمه می‌کند

کلاس شما باید تعدادی متد داشته باشد که ورودی‌هایی دریافت کرده و آن را به فعال شدن رویداد ترجمه کند. در مثال **MailManger** من، متد **SimulateNewMail** فراخوانی می‌شود تا بیان کند که یک ایمیل جدید به **MailManger** رسیده است:

```
internal class MailManager {

    // Step #4: Define a method that translates the
    // input into the desired event
```

```

public void SimulateNewMail(String from, String to, String subject) {

    // Construct an object to hold the information we wish
    // to pass to the receivers of our notification
    NewMailEventArgs e = new NewMailEventArgs(from, to, subject);

    // Call our virtual method notifying our object that the event
    // occurred. If no type overrides this method, our object will
    // notify all the objects that registered interest in the event
    OnNewMail(e);
}
}

```

SimulateNewMail اطلاعات درباره‌ی ایمیل را دریافت می‌کند و یک شی **NewMailEventArgs** می‌سازد و اطلاعات پیام را به سازنده‌اش ارسال می‌کند. سپس متد مجازی **OnNewMail** از **MailManager** فراخوانی می‌شود تا رسماً شی **MailManger** را از ایمیل جدید با خبر سازد. معمولاً، این باعث می‌شود رویداد فعال گردد و تمام متدهای ثبت شده با خبر شوند. (همانطور که گفته شد، یک کلاس که از **MailManager** به عنوان کلاس پایه استفاده می‌کند می‌تواند این رفتار را بازنویسی کند).

کامپایلر چگونه یک رویداد را پیاده سازی می کند

اکنون که شما می‌دانید چگونه یک کلاس که یک عضو رویداد را ارائه می‌کند، تعریف کنید، بگذارید دقیق تر نگاه کنیم که یک رویداد چیست و چگونه کار می‌کند. در کلاس **MailManger**، یک خط از کد داریم که خود عضو رویداد را تعریف می‌کند:

```

public event EventHandler<NewMailEventArgs> NewMail;

    وقتی کامپایلر سی‌شارپ کد فوق را کامپایل می‌کند، این خط را به سه ساخت زیر ترجمه می‌نماید:

// 1. A PRIVATE delegate field that is initialized to null
private EventHandler<NewMailEventArgs> NewMail = null;

// 2. A PUBLIC add_Xxx method (where Xxx is the Event name)
// Allows methods to register interest in the event.
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
    // The loop and the call to CompareExchange is all just a fancy way
    // of adding a delegate to the event in a thread-safe way
    EventHandler<NewMailEventArgs>prevHandler;
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
    do {
        prevHandler = newMail;
        EventHandler<NewMailEventArgs>newHandler =
            (EventHandler<NewMailEventArgs>) Delegate.Combine(prevHandler, value);
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
            ref this.NewMail, newHandler, prevHandler);
    } while (newMail != prevHandler);
}

// 3. A PUBLIC remove_Xxx method (where Xxx is the Event name)
// Allows methods to unregister interest in the event.
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
    // The loop and the call to CompareExchange is all just a fancy way
    // of removing a delegate from the event in a thread-safe way

```

```

EventHandler<NewMailEventArgs> prevHandler;
EventHandler<NewMailEventArgs> newMail = this.NewMail;
do {
    prevHandler = newMail;
    EventHandler<NewMailEventArgs> newHandler =
        (EventHandler<NewMailEventArgs>) Delegate.Remove(prevHandler, value);
    newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
        ref this.NewMail, newHandler, prevHandler);
} while (newMail != prevHandler);
}

```

اولین ساخت، تعریف یک فیلد از نوع نماینده‌ی مناسب است. این فیلد، ارجاعی به ابتدای لیستی از نماینده‌هاست که هنگام رخداد رویداد مطلع می‌شوند. این فیلد به **null** مقداردهی اولیه شده است به این معنی که هیچ کس علاقه‌اش به رویداد را ثبت نکرده است. وقتی یک متد علاقه‌اش به رویداد را ثبت می‌کند، این فیلد به یک نمونه از نماینده **EventHandler<NewMailEventArgs>** اشاره می‌کند که ممکن است به دیگر نماینده‌های **EventHandler<NewMailEventArgs>** اشاره کند. وقتی یک شونده (دریافت کننده خبر)، علاقه‌اش به متد را ثبت می‌کند، شونده یک نمونه از نوع نماینده را به لیست می‌افزاید. واضح است که حذف کردن یعنی حذف نماینده از لیست.

شما متوجه خواهید شد که فیلد نماینده، در این مثال **NewMail**، همیشه **private** است اگرچه سورس کد اصلی، رویداد را **public** تعریف می‌کند. علت **private** کردن فیلد نماینده آن است که کد خارج از کلاس تعریف کننده، نتواند به اشتباه آن را دستکاری کند. اگر فیلد **public** بود، هر کدی می‌توانست مقدار فیلد را تغییر داده و تمام نماینده‌هایی که علاقه شان به متد را ثبت کرده بودند، کاملاً از بین می‌رفتند.

دومین ساختی که کامپایلر سی‌شارپ تولید می‌کند یک متد است که اجازه می‌دهد دیگر اشیاء علاقه شان به رویداد را ثبت کنند. کامپایلر سی‌شارپ به صورت خودکار این تابع را با افزودن **_add** به ابتدای نام رویداد (**NewMail**) نامگذاری می‌کند. کامپایلر سی‌شارپ به صورت خودکار، کد درون این متد را تولید می‌کند. کد همیشه متد استاتیک **Combine** از **System.Delegate** را فراخوانی می‌کند، که یک نمونه از یک نماینده را به لیست نمایندگان می‌افزاید و ابتدای جدید از لیست را برمی‌گرداند که دوباره در فیلد ذخیره می‌شود. سومین ساختی که کامپایلر سی‌شارپ تولید می‌کند یک متد است که اجازه می‌دهد یک شی علاقه‌اش به متد را بردارد. مجدداً، کامپایلر سی‌شارپ این تابع را با افزودن **_remove** به ابتدای نامگذاری می‌کند. کد درون متد همیشه متد استاتیک **Remove** از **Delegate** را فراخوانی می‌کند که یک نماینده را از لیست نمایندگان حذف کرده و ابتدای جدید لیست را برمی‌گرداند که دوباره در فیلد ذخیره می‌گردد.

هشدار اگر شما سعی در حذف متدی که هرگز اضافه نشده است بکنید، آنگاه متد **Remove** از **Delegate** کاری انجام نمی‌دهد. یعنی هیچ اکسپشن یا هشداری از هر نوع دریافت نمی‌کنید، مجموعه‌ی متدهای رویداد، دست نخورده باقی می‌ماند.

نکته متدهای **add** و **remove** از الگوی شناخته شده‌ای برای آپدیت یک مقدار به روش ترد-امن استفاده می‌کنند. این الگو در بخش "الگوی Interlocked Anything" در فصل ۲۸ بحث شده است.

در این مثال، متدهای **add** و **remove** **public** هستند. علت آنکه آن‌ها **public** هستند اینست که کد اصلی، رویداد را **public** تعریف کرده است. اگر رویداد **protected** تعریف شده بود، متدهای **add** و **remove** که توسط کامپایلر تولید می‌شدند نیز **protected** بودند. بنابراین، وقتی رویدادی را در یک نوع تعریف می‌کنید، دسترس پذیری رویداد تعیین می‌کند چه کدی می‌تواند علاقه‌اش به رویداد را اضافه و حذف کند، اما تنها خود نوع است که می‌تواند به فیلد نماینده دسترسی مستقیم داشته باشد. اعضای رویداد به عنوان **static** یا **virtual** نیز می‌توانند تعریف شوند، که متدهای **add** و **remove** تولید شده توسط کامپایلر نیز به ترتیب **static** یا **virtual** خواهند بود.

علاوه بر تولید سه ساخت مذکور، کامپایلر یک ورودی تعریف رویداد در متادیتای اسمبلی مدیریت شده قرار می‌دهد. این ورودی شامل تعدادی پرچم و نوع نماینده است و به متدهای دستیابی **add** و **remove** اشاره دارد. این اطلاعات تنها برای نمایش ارتباط بین مفهوم خلاصه‌ی یک "رویداد" و متدهای دستیابی آن است. کامپایلرها و دیگر ابزارها می‌توانند از این اطلاعات استفاده کنند و این اطلاعات همچنین با استفاده از کلاس

System.Reflection.EventInfo نیز می‌تواند بدست آید. هرچند، CLR خودش از این اطلاعات متادیتا استفاده نمی‌کند و در زمان اجرا تنها به متدهای دستیابی نیاز دارد.

طراحی یک نوع که از یک رویداد استفاده می‌کند

تا این لحظه، کار سخت را انجام داده‌اید. در این بخش، من نشان می‌دهم چگونه یک نوع تعریف کنید که از رویداد تولید شده توسط نوعی دیگر استفاده کند. بگذارید با بررسی کد نوع **Fax** شروع کنیم:

```
internal sealed class Fax {
    // Pass the MailManager object to the constructor
    public Fax(MailManager mm) {

        // Construct an instance of the EventHandler<NewMailEventArgs>
        // delegate that refers to our FaxMsg callback method.
        // Register our callback with MailManager's NewMail event
        mm.NewMail += FaxMsg;
    }

    // This is the method the MailManager will call
    // when a new e-mail message arrives
    private void FaxMsg(Object sender, NewMailEventArgs e) {

        // 'sender' identifies the MailManager object in case
        // we want to communicate back to it.

        // 'e' identifies the additional event information
        // the MailManager wants to give us.

        // Normally, the code here would fax the e-mail message.
        // This test implementation displays the info in the console
        Console.WriteLine("Faxing mail message:");
        Console.WriteLine(" From={0}, To={1}, Subject={2}",
            e.From, e.To, e.Subject);
    }

    // This method could be executed to have the Fax object unregister
    // itself with the NewMail event so that it no longer receives
    // notifications
    public void Unregister(MailManager mm) {

        // Unregister with MailManager's NewMail event
        mm.NewMail -= FaxMsg;
    }
}
```

وقتی برنامه ایمیل مقاردهی اولیه می‌شود، ابتدا یک شی **MailManager** می‌سازد و ارجاع به این شی را در یک متغیر ذخیره می‌کند. سپس برنامه یک شی **Fax** ساخته و ارجاع به شی **MailManager** را به عنوان پارامتر به سازنده ارسال می‌کند. در سازنده‌ی **Fax**، شی **Fax** علاقه‌اش به رویداد **NewMail** از **MailManager** را با استفاده از عملگر **+=** سی‌شارپ ثبت می‌کند:

```
mm.NewMail += FaxMas;
```

چون کامپایلر سی شارپ پشتیبانی داخلی برای رویدادها دارد، کامپایلر، استفاده از عملگر += را به خط زیر ترجمه می کند تا علاقه ی شی به رویداد را اضافه کند:

```
mm.add_NewMail(new EventHandler<NewMailEventArgs>(this.FaxMsg));
```

همانگونه که می بینید، کامپایلر سی شارپ کدی را تولید کرده است که یک شی نماینده **EventHandler<NewMailEventArgs>** می سازد که متد **FaxMsg** از کلاس **Fax** را می پوشاند. سپس کامپایلر سی شارپ متد **add_NewMail** از **MailManager** را فراخوانی کرده، نماینده ی جدید را به آن ارسال می کند. البته، شما می توانید تمام این را با کامپایلر کد و نگاه به کد IL توسط ابزاری مثل **ILDasm.exe** بررسی کنید. حتی اگر شما از یک زبان برنامه نویسی استفاده می کنید که مستقیماً رویدادها را پشتیبانی نمی کند باز هم می توانید با فراخوانی صریح متد دستیابی **add**، یک نماینده را با رویداد ثبت کنید. اثر آن همانند قبلی است، اما سورس کد به زیبایی قبل نخواهد بود. در واقع متد **add** است که نماینده را با رویداد بوسیله افزودن آن به لیست نمایندگان رویداد، ثبت می کند. وقتی شی **MailManager**، رویداد را فعال می کند، متد **FaxMsg** از شی **Fax** فراخوانی می شود. یک ارجاع به شی **MailManager** به عنوان اولین پارامتر، **sender**، به متد ارسال می شود. اغلب اوقات، این پارامتر نادیده گرفته می شود اما اگر شی **Fax** بخواهد در پاسخ به خبر رویداد به اعضای شی **MailManager** دسترسی پیدا کند، می تواند از این پارامتر استفاده کند. پارامتر دوم یک ارجاع به یک شی **NewMailEventArgs** است. این شی شامل هرگونه اطلاعات اضافی است که طراح **MailManger** و **NewMailEventArgs** فکر کرده است برای دریافت کننده رویداد می تواند مفید باشد.

از طریق شی **NewMailEventArgs**، متد **FaxMsg** به آسانی به اطلاعات ارسال کننده پیام، و دریافت کننده ی پیام و موضوع پیام دسترسی دارد. در یک شی **Fax** واقعی، این اطلاعات به مکانی فکس می شوند. در این مثال، اطلاعات فقط در کنسول نمایش داده می شوند.

وقتی شی ای دیگر مایل به دریافت خبر رویداد نباشد، باید علاقه اش را حذف کند. برای نمونه، شی **Fax** باید علاقه اش به رویداد **NewMail** را حذف کند اگر کاربر دیگر نخواهد ایمیل هایش به فکسی ارسال شوند. تا زمانی که یک شی یکی از متدهایش را با یک رویداد ثبت کرده است، شی غیر قابل جمع آوری است. اگر نوع شما متد **Dispose** از **IDisposable** را پیاده سازی کند، این پیاده سازی باید باعث شود که علاقه به تمام رویدادها حذف شوند. (برای اطلاعات بیشتر پیرامون **IDisposable** به فصل ۲۱ "مدیریت حافظه خودکار (جمع آوری زباله)" مراجعه کنید.)

کدی که نشان می دهد چگونه علاقه به یک رویداد را باید حذف کرد در متد **Unregister** از **Fax** نمایش داده شده است. این متد عملاً معادل کد سازنده ی **Fax** است. تنها فرق اینست که به جای += از -= استفاده کرده است. وقتی کامپایلر سی شارپ کدی را می بیند که از عملگر -= برای حذف علاقه به یک رویداد استفاده می کند، کامپایلر یک فراخوانی به متد **remove** رویداد، تولید می کند:

```
mm.remove_NewMail(new EventHandler<NewMailEventArgs>(FaxMsg));
```

همانند عملگر +=، حتی اگر شما از یک زبان برنامه نویسی استفاده می کنید که پشتیبانی مستقیم از رویدادها ندارد، باز هم می توانید با فراخوانی صریح متد دستیابی **remove** نماینده ای را از یک رویداد حذف کنید. متد **remove**، نماینده را از رویداد با جستجوی یک نماینده در لیست که همان متدی که بدان ارسال شده است را می پوشاند، حذف می کند. اگر تطابقی یافت شد، نماینده ی موجود از لیست نمایندگان حذف می گردد. اگر تطابقی یافت نشد، خطایی رخ نمی دهد و لیست دست نخورده باقی می ماند. ضمناً سی شارپ نیاز دارد که کد شما از عملگرها += و -= برای افزودن یا حذف نماینده ها از لیست، استفاده کند. اگر شما سعی کنید صریحاً **add** یا **remove** را فراخوانی کنید، کامپایلر سی شارپ پیام خطای CS0571 "Cannot explicitly call operator or accessor" را تولید می کند.

پیاده سازی صریح یک رویداد

نوع **System.Windows.Forms.Control** نزدیک به ۷۰ رویداد تعریف می کند. اگر نوع **Control** رویدادها را به گونه ای پیاده سازی کرده بود که به کامپایلر اجازه دهد به صورت ضمنی متدهای دستیابی **add** و **remove** و فیلدهای نماینده را تولید کند، آنگاه هر شی **Control**، ۷۰ فیلد نماینده تنها برای رویدادها داشت. چون اغلب برنامه نویسان تنها از چند رویداد استفاده می کنند، حجم عظیمی از حافظه برای هر شی که از روی نوع مشتق شده از **Control**، ساخته می شد به هدر می رفت. در ضمن، نوع **System.Web.UI.Control** از **ASP.NET** و **System.Windows.UIElement** از **WPF**، تعداد زیادی رویداد تعریف می کنند که اغلب برنامه نویسان از آن ها استفاده نمی کنند.

در این بخش، من بحث می کنم چگونه کامپایلر سی شارپ به برنامه نویسی یک کلاس اجازه می دهد صریحاً یک رویداد را پیاده سازی کند تا به برنامه نویسی اجازه ی کنترل متدهای **add** و **remove** در دستکاری نماینده های کالک را بدهد. من می خواهم نشان دهم چگونه پیاده سازی صریح یک رویداد می تواند برای پیاده سازی کارای یک کلاس که رویدادهای زیادی دارد استفاده شود. هرچند، مطمئناً سناریوهای دیگری وجود دارد که شاید شما بخواهید رویداد یک نوع را صریحاً پیاده سازی کنید. برای ذخیره ی کارآمد نمایندگان رویداد، هر شی ای که رویدادهایی ارائه می کند یک مجموعه (معمولاً یک دیکشنری) با گونه-

ای از شناسه‌ی رویداد به عنوان کلید و لیستی از نمایندگان به عنوان مقدار، را نگهداری می‌نماید. وقتی شی جدیدی ساخته می‌شود، این مجموعه خالی است. وقتی علاقه‌ای به یک رویداد ثبت می‌شود، شناسه‌ی رویداد در مجموعه جستجو می‌شود. اگر شناسه‌ی رویداد وجود داشت، نماینده‌ی جدید با لیست نمایندگان این رویداد ترکیب می‌شود. اگر شناسه‌ی رویداد در مجموعه نبود، شناسه‌ی رویداد با نماینده به مجموعه افزوده می‌شود.

وقتی شی نیاز به فعال سازی یک رویداد دارد، شناسه‌ی رویداد در مجموعه جستجو می‌شود. اگر مجموعه هیچ ورودی‌ای برای شناسه‌ی رویداد نداشته باشد، هیچ کس علاقه‌اش به رویداد را ثبت نکرده و نیاز نیست هیچ نماینده‌ای فراخوانی شود. اگر شناسه‌ی رویداد در مجموعه بود، لیست نمایندگان مربوط به شناسه‌ی رویداد فراخوانی می‌شود. پیاده‌سازی این الگوی طراحی بر عهده‌ی برنامه‌نویسی است که نوعی که رویدادها را تعریف می‌کند، طراحی می‌نماید. برنامه‌نویسی که از نوع استفاده می‌کند، درباره پیاده‌سازی درونی رویدادها چیزی نمی‌داند.

نمونه‌ای را در اینجا می‌بینید که این الگو را به کار برده است. در ابتدا من یک کلاس **EventSet** پیاده‌سازی کرده‌ام که مجموعه‌ی رویدادها و لیست نمایندگان رویداد را نشان می‌دهد:

```
using System;
using System.Collections.Generic;

// This class exists to provide a bit more type safety and
// code maintainability when using EventSet
public sealed class EventKey : Object { }

public sealed class EventSet {
    // The private dictionary used to maintain EventKey -> Delegate mappings
    private readonly Dictionary<EventKey, Delegate> m_events =
        new Dictionary<EventKey, Delegate>();

    // Adds an EventKey -> Delegate mapping if it doesn't exist or
    // combines a delegate to an existing EventKey
    public void Add(EventKey eventKey, Delegate handler) {
        Monitor.Enter(m_events);
        Delegate d;
        m_events.TryGetValue(eventKey, out d);
        m_events[eventKey] = Delegate.Combine(d, handler);
        Monitor.Exit(m_events);
    }

    // Removes a delegate from an EventKey (if it exists) and
    // removes the EventKey -> Delegate mapping the last delegate is removed
    public void Remove(EventKey eventKey, Delegate handler) {
        Monitor.Enter(m_events);
        // Call TryGetValue to ensure that an exception is not thrown if
        // attempting to remove a delegate from an EventKey not in the set
        Delegate d;
        if (m_events.TryGetValue(eventKey, out d)) {
            d = Delegate.Remove(d, handler);
            // If a delegate remains, set the new head else remove the EventKey
            if (d != null) m_events[eventKey] = d;
            else m_events.Remove(eventKey);
        }
        Monitor.Exit(m_events);
    }
}
```

```
// Raises the event for the indicated EventKey
public void Raise(EventKey eventKey, Object sender, EventArgs e) {
    // Don't throw an exception if the EventKey is not in the set
    Delegate d;
    Monitor.Enter(m_events);
    m_events.TryGetValue(eventKey, out d);
    Monitor.Exit(m_events);
    if (d != null) {
        // Because the dictionary can contain several different delegate types,
        // it is impossible to construct a type-safe call to the delegate at
        // compile time. So, I call the System.Delegate type's DynamicInvoke
        // method, passing it the callback method's parameters as an array of
        // objects. Internally, DynamicInvoke will check the type safety of the
        // parameters with the callback method being called and call the method.
        // If there is a type mismatch, then DynamicInvoke will throw an exception.
        d.DynamicInvoke(new Object[] { sender, e });
    }
}
```

نکته FCL یک نوع `System.ComponentModel.EventHandlerList` تعریف می‌کند که اساساً کار کلاس `EventSet` من را می‌کند. نوع‌های `System.Web.UI.Control` و `System.Windows.Forms.Control` در درون خود برای نگهداری مجموعه‌ی رویدادها از نوع `EventHandlerList` استفاده می‌کنند. اگر بخواهید می‌توانید از نوع `EventHandlerList` استفاده کنید. تفاوت بین `EventHandlerList` و نوع `EventSet` من اینست که `EventHandlerList` از یک لیست پیوندی به جای یک جدول هش استفاده می‌کند. این یعنی دسترسی به عناصری که `EventHandlerList` مدیریت می‌کند کندتر از `EventSet` من است. به علاوه، `EventHandlerList` هیچ روش ترد-امنی برای دسترسی به رویدادها ارائه نمی‌کند و اگر شما نیاز به این دارید باید پوشاننده‌ی ترد-امن خود را اطراف مجموعه `EventHandlerList` پیاده سازی کنید.

اکنون، یک کلاس که از کلاس `EventSet` من استفاده می‌کند را نشان می‌دهم. این کلاس فیلدی دارد که به یک شی `EventSet` اشاره می‌کند و هر رویداد این کلاس صریحاً پیاده سازی شده است به گونه‌ای که متد `add` از هر رویداد نماینده‌ی کالباک تعیین شده را در شی `EventSet` ذخیره می‌کند و متد `remove` از هر رویداد، نماینده‌ی کالباک تعیین شده را (در صورت یافتن) حذف می‌کند:

```
using System;

// Define the EventArgs-derived type for this event.
public class FooEventArgs : EventArgs { }

public class TypewithLotsOfEvents {

    // Define a private instance field that references a collection.
    // The collection manages a set of Event/Delegate pairs.
    // NOTE: The EventSet type is not part of the FCL, it is my own type.
    private readonly EventSet m_eventSet = newEventSet();

    // The protected property allows derived types access to the collection.
    protected EventSet EventSet { get { return m_eventSet; } }
```



```

#region Code to support the Foo event (repeat this pattern for additional events)
// Define the members necessary for the Foo event.
// 2a. Construct a static, read-only object to identify this event.
// Each object has its own hash code for looking up this
// event's delegate linked list in the object's collection.
protected static readonly EventKey s_fooEventKey = newEventKey();

// 2d. Define the event's accessor methods that add/remove the
// delegate from the collection.
public event EventHandler<FooEventArgs> Foo {
    add { m_eventSet.Add(s_fooEventKey, value); }
    remove { m_eventSet.Remove(s_fooEventKey, value); }
}

// 2e. Define the protected, virtual On method for this event.
protected virtual void OnFoo(FooEventArgs e) {
    m_eventSet.Raise(s_fooEventKey, this, e);
}

// 2f. Define the method that translates input to this event.
public void SimulateFoo() {OnFoo(newFooEventArgs());}
#endregion
}

```

کدی که از نوع **TypeWithLotsOfEvents** استفاده می‌کند نمی‌تواند بگوید که آیا رویدادها به صورت ضمنی توسط کامپایلر یا صریحاً توسط برنامه‌نویس پیاده‌سازی شده‌اند. آن‌ها فقط رویدادها را با نحو عادی ثبت می‌کنند. کد زیر این مطلب را نشان می‌دهد:

```

public sealed class Program {
    public static void Main() {
        TypewithLotsOfEvents twle = newTypewithLotsOfEvents();

        // Add a callback here
        twle.Foo += HandleFooEvent;

        // Prove that it worked
        twle.SimulateFoo();
    }

    private static void HandleFooEvent(object sender, FooEventArgs e) {
        Console.WriteLine("Handling Foo Event here...");
    }
}

```

فصل ۱۲: جنریک ها

برنامه‌نویسانی که با برنامه‌نویس شی گرا آشنا هستند مزایای آن را می‌دانند. یکی از بزرگترین فوایدی که برنامه‌نویسان را بسیار بهره‌ور می‌کند استفاده مجدد از کد است که قابلیت مشتق کردن یک کلاس است که تمام توانایی‌های یک کلاس پایه را به ارث می‌برد. کلاس مشتق شده به سادگی می‌تواند متدهای مجازی را بازنویسی کند و برای سفارشی سازی رفتار کلاس پایه طبق نیاز برنامه‌نویس، متدهای جدید را اضافه می‌کند. جنریک‌ها Generics مکانیزم دیگری است که توسط CLR و زبان‌های برنامه‌نویسی ارائه می‌شود که سطحی بالاتر در استفاده مجدد از کد را فراهم می‌کند: استفاده مجدد از الگوریتم.

اساساً، یک برنامه‌نویس، یک الگوریتم مثل مرتب سازی، جستجو، جایجایی، مقایسه یا تبدیل را تعریف می‌کند. اما برنامه‌نویسی که الگوریتم را تعریف می‌کند نوع داده (هایی) که الگوریتم بر روی آن عمل می‌کند را تعیین نمی‌کند، الگوریتم می‌تواند به صورت عمومی بر اشیایی از انواع مختلف اعمال شود. برنامه‌نویس دیگری از این الگوریتم موجود تا زمانی که او نوع داده (هایی) که الگوریتم بر روی آن باید عمل کند را تعیین نماید، می‌تواند استفاده کند. برای نمونه، یک الگوریتم مرتب سازی که روی **Int32** ها، **String** و غیره عمل می‌کند، یا یک الگوریتم مقایسه روی **DateTime** ها، **Version** و غیره عمل می‌کند. اغلب الگوریتم‌ها در یک نوع کپسوله می‌شوند و CLR اجازه‌ی تعریف نوع‌های ارجاعی جنریک و نوع‌های مقداری جنریک را می‌دهد اما اجازه‌ی ساخت نوع‌های شمارشی جنریک را نمی‌دهد. به علاوه، CLR اجازه‌ی ساخت رابط‌های جنریک و نماینده‌های جنریک را می‌دهد. به ندرت، یک تک متد می‌تواند یک الگوریتم مفید را کپسوله کند و بنابراین CLR اجازه‌ی ساخت متدهای جنریکی که در یک نوع ارجاعی، نوع مقداری یا رابط تعریف شده‌اند را می‌دهد.

بگذارید به مثال نگاه کنیم. **FCL** یک الگوریتم لیست جنریک تعریف می‌کند که می‌داند چگونه مجموعه‌ای از اشیاء را مدیریت کند، نوع داده این اشیاء توسط الگوریتم جنریک تعیین نشده است. کسی که بخواهد از الگوریتم لیست جنریک استفاده کند می‌تواند بعداً نوع دقیق داده را برای استفاده تعیین کند.

کلاسی از **FCL** که الگوریتم لیست جنریک را کپسوله می‌کند **List<T>** (تلفظ می‌شود، لیستی از تی **List of Tee**) نامیده می‌شود، و این کلاس در فضای نام **System.Collections.Generic** تعریف شده است. تعریف این کلاس شبیه به این است (کد بسیار خلاصه شده است):

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
IList, ICollection, IEnumerable {

    public List();
    public void Add(T item);
    public Int32 BinarySearch(T item);
    public void Clear();
    public Boolean Contains(T item);
    public Int32 IndexOf(T item);
    public Boolean Remove(T item);
    public void Sort();
    public void Sort(IComparer<T> comparer);
    public void Sort(Comparison<T> comparison);
    public T[] ToArray();
    public Int32 Count { get; }
    public T this[Int32 index] { get; set; }
}
```

برنامه‌نویسی که کلاس **جنریک List** را تعریف کرده، با گذاشتن **<T>** بلافاصله بعد از نام کلاس، بیان کرده که این کلاس با نوع داده‌ای تعیین نشده‌ای کار می‌کند. هنگام تعریف یک متد یا نوع جنریک، هر متغیری که برنامه‌نویس برای نوع‌ها تعیین کند (مثل **T**)، پارامترهای نوع **type parameteres** نامیده می‌شوند. **T** نام یک متغیر است که می‌تواند در هر جایی از سورس کد که یک نوع داده استفاده می‌شود، مورد استفاده قرار گیرد. برای نمونه، در تعریف کلاس **List**، شما می‌بینید که **T** برای پارامترهای متد (متد **Add** یک پارامتر از نوع **T** می‌پذیرد) و مقادیر برگشتی (متد **ToArray** یک آرایه تک بعدی از نوع **T** برمی‌گرداند) استفاده شده است. مثال دیگر، متد **ایندکسر** (در سی‌شارپ **this** نامیده می‌شود) است. **ایندکسر** یک متد دستیابی **get** دارد که یک مقدار

از نوع **T** برمی‌گرداند و یک متد دستیابی **set** دارد که یک پارامتر از نوع **T** می‌پذیرد. چون متغیر **T** می‌تواند هر جایی که یک نوع داده تعیین می‌شود، استفاده گردد، همچنین این ممکن است که شما از **T** هنگام تعریف متغیرهای محلی درون یک متد یا هنگام تعریف فیلدهای درون یک نوع استفاده کنید.

نکته توصیه‌های مایکروسافت در طراحی بیان می‌کند که متغیرهای پارامتر جنریک باید **T** نامیده شوند یا حداقل با یک حرف **T** بزرگ (مثل **TKey** و **TValue**) شروع شوند. حرف **T** بزرگ برای نوع **type** است دقیقاً همانند یک حرف **I** بزرگ برای رابط **interface** (مثل **Comparable**).

حال که نوع جنریک **List<T>** تعریف شده است، دیگر برنامه‌نویسان می‌توانند از این الگوریتم جنریک با تعیین نوع داده‌ای که می‌خواهند الگوریتم بر روی آن عمل کند، استفاده نمایند. هنگام استفاده از یک نوع یا متد جنریک، نوع‌های داده‌ای تعیین شده با عنوان آرگومان‌های نوع **type arguments** اطلاع می‌شوند. برای نمونه، یک برنامه‌نویس شاید بخواهد از الگوریتم **List** با تعیین یک آرگومان نوع **DateTime** استفاده کند. کد زیر این را نشان می‌دهد:

```
private static void SomeMethod() {
    // Construct a List that operates on DateTime objects
    List<DateTime> dtList = new List<DateTime>();

    // Add a DateTime object to the list
    dtList.Add(DateTime.Now); // No boxing

    // Add another DateTime object to the list
    dtList.Add(DateTime.MinValue); // No boxing

    // Attempt to add a String object to the list
    dtList.Add("1/1/2004"); // Compile-time error

    // Extract a DateTime object out of the list
    DateTime dt = dtList[0]; // No cast required
}
```

جنریک‌ها، فواید زیر را برای برنامه‌نویس به همراه دارند:

- **محافظت از سورس کد** برنامه‌نویسی که از یک الگوریتم جنریک استفاده می‌کند نیاز به دسترسی به سورس کد الگوریتم ندارد. هرچند در استفاده از قالب‌های **C++** یا جنریک‌های **Java**، سورس کد الگوریتم باید در دسترس برنامه‌نویسی که از الگوریتم استفاده می‌کند باشد.
- **امنیت نوع** وقتی یک الگوریتم جنریک با یک نوع داده‌ای خاص استفاده می‌شود، کامپایلر و **CLR** این را درک می‌کنند و مطمئن می‌شوند فقط اشیایی که با نوع داده‌ی تعیین شده سازگارند با الگوریتم استفاده می‌شوند. سعی در استفاده از یک شی از نوع ناسازگار منجر به یک خطای کامپایلر یا تولید یک اکسپشن در زمان اجرا می‌شود. در مثال، سعی در ارسال یک شی **String** به متد **Add** باعث می‌شود کامپایلر یک خطا اعلام کند.
- **کد تمیز تر** چون کامپایلر امنیت نوع را اجبار می‌کند، تبدیل‌های کمتری در سورس کد شما نیاز است به این معنی که کد شما راحت تر نوشته و نگهداری می‌شود. در آخرین خط از **SomeMethod**، برنامه‌نویس نیاز ندارد که از یک تبدیل (**DateTime**) استفاده کند تا نتیجه‌ی ایندکس (درخواست عنصر موجود در اندیس **0**) را در متغیر **dt** ذخیره نماید.
- **عملکرد بهتر** قبل از جنریک‌ها، روش تعریف یک الگوریتم کلی و عمومی، تعریف تمام اعضایش از نوع **Object** بود. اگر شما می‌خواستید الگوریتم را با نمونه‌های نوع مقداری استفاده کنید، **CLR** مجبور بود نمونه‌ی نوع مقداری را قبل از فراخوانی اعضای الگوریتم، بسته‌بندی کند. همانگونه که در فصل ۵ "نوع‌های اصلی، ارجاعی و مقداری" بحث شد، بسته‌بندی باعث تخصیص حافظه در هیپ مدیریت شده می‌شود که منجر به رخداد مکرر جمع آوری زباله می‌گردد و در نتیجه سرعت برنامه پایین می‌آید. چون اکنون می‌توان یک الگوریتم جنریک ساخت که با نوع مقداری تعیین شده کار کند، نمونه‌های نوع مقداری با مقدار ارسال می‌شوند و دیگر **CLR** نیاز به بسته‌بندی کردن ندارد. به علاوه، چون تبدیلی نیاز نیست (مورد قبلی را ببینید) **CLR** نیاز ندارد امنیت نوع را در تبدیل بررسی کند و این باز هم کد را سریعتر می‌کند.

برای آنکه فواید عملکردی جنریک‌ها را ببینیم، برنامه‌ای نوشته‌ام که عملکرد الگوریتم جنریک **List** را در مقابل الگوریتم غیر جنریک **ArrayList** از FCL بررسی می‌کند. من عملکرد هر دو الگوریتم را با استفاده از هر دوی اشیاء نوع مقداری و اشیاء نوع ارجاعی آزمایش کرده‌ام. خود برنامه در اینجا آمده است:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

public static class Program {
    public static void Main() {
        ValueTypePerfTest();
        ReferenceTypePerfTest();
    }

    private static void ValueTypePerfTest() {
        const Int32 count = 10000000;

        using (new OperationTimer("List<Int32>")) {
            List<Int32> l = new List<Int32>(count);
            for (Int32 n = 0; n < count; n++) {
                l.Add(n);
                Int32 x = l[n];
            }
            l = null; // Make sure this gets GC'd
        }
        using (new OperationTimer("ArrayList of Int32")) {
            ArrayList a = new ArrayList();
            for (Int32 n = 0; n < count; n++) {
                a.Add(n);
                Int32 x = (Int32) a[n];
            }
            a = null; // Make sure this gets GC'd
        }
    }

    private static void ReferenceTypePerfTest() {
        const Int32 count = 10000000;

        using (new OperationTimer("List<String>")) {
            List<String> l = new List<String>();
            for (Int32 n = 0; n < count; n++) {
                l.Add("x");
                String x = l[n];
            }
            l = null; // Make sure this gets GC'd
        }
        using (new OperationTimer("ArrayList of String")) {
            ArrayList a = new ArrayList();
```

```

        for (Int32 n = 0; n < count; n++) {
            a.Add("X");
            String x = (String) a[n];
        }
        a = null; // Make sure this gets GC'd
    }
}

// This class is useful for doing operation performance timing
internal sealed class OperationTimer : IDisposable {
    private Int64 m_startTime;
    private String m_text;
    private Int32 m_collectionCount;

    public OperationTimer(String text) {
        PrepareForOperation();

        m_text = text;
        m_collectionCount = GC.CollectionCount(0);

        // This should be the last statement in this
        // method to keep timing as accurate as possible
        m_startTime = Stopwatch.GetTimestamp();
    }

    public void Dispose() {
        Console.WriteLine("{0,6:###.00} seconds (GCs={1,3}) {2}",
            (Stopwatch.GetTimestamp() - m_startTime) /
            (Double) Stopwatch.Frequency,
            GC.CollectionCount(0) - m_collectionCount, m_text);
    }

    private static void PrepareForOperation() {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
    }
}

```

وقتی من برنامه را کامپایل کردم و یک ساخت نهایی از آن را روی کامپیوترم اجرا کردم (همراه با فعال کردن بهینه سازی کد) خروجی زیر را گرفتم:

```

.10 seconds (GCs= 0) List<Int32>
3.02 seconds (GCs= 45) ArrayList of Int32
.47 seconds (GCs= 6) List<String>
.51 seconds (GCs= 6) ArrayList of String

```

خروجی نشان می‌دهد که استفاده از الگوریتم جنریک **List** با نوع **Int32** بسیار سریعتر از استفاده از الگوریتم غیرجنریک **ArrayList** با **Int32** است. در واقع تفاوت شگفت‌انگیز است: ۰.۱ ثانیه در مقابل ۳ ثانیه. ۳۰ برابر سریعتر است! به علاوه، استفاده از یک نوع مقداری (**Int32**) با **ArrayList** باعث رخداد تعداد زیادی عملیات بسته‌بندی می‌شود که منجر به ۴۵ جمع آوری زباله گردیده است. در حالیکه، الگوریتم **List** به ۰ جمع آوری زباله نیاز دارد.

نتیجه آزمایش که از نوع‌های ارجاعی استفاده می‌کند زیاد مهم نیست. در اینجا می‌بینیم که زمان‌ها و اعداد جمع آوری زباله تقریباً یکسان است. پس به نظر نمی‌رسد که الگوریتم جنریک **List** در اینجا مزیتی داشته باشد. به هر حال، به یاد داشته باشید وقتی از یک الگوریتم جنریک استفاده می‌کنید، کد تمیزتر بوده و امنیت نوع را در زمان کامپایل دارید. پس اگرچه بهبود سرعت زیاد نیست، فوایدی که یک الگوریتم جنریک برای شما دارد، واقعاً یک بهبود و پیشرفت به حساب می‌آید.

نکته شما نیاز دارید که درک کنید CLR کد اصلی برای هر متد را اولین بار که متد برای یک نوع داده خاص فراخوانی می‌شود، تولید می‌کند. این، حجم کاری یک برنامه را افزایش می‌دهد و به عملکرد برنامه ضربه می‌زند. من در این باره در بخش "زیر ساختار جنریک" در این فصل صحبت می‌کنم.

جنریک‌ها در کتابخانه کلاس فریمورک

مطمئننا، مهمترین استفاده از جنریک‌ها با کلاس‌های مجموعه‌هاست، و **FCL** چندین کلاس مجموعه جنریک برای استفاده‌ی شما تعریف می‌کند. اغلب این کلاس‌ها را می‌توان در فضای نام **System.Collections.Generic** و فضای نام **System.Collections.ObjectModel** یافت. همچنین کلاس‌های مجموعه جنریک تر-امن نیز در فضای نام **System.Collections.Concurrent** وجود دارند. مایکروسافت توصیه می‌کند که برنامه-نویسان از کلاس‌های مجموعه جنریک استفاده کنند و استفاده از کلاس‌های مجموعه غیر جنریک را به دلایل مختلف منع می‌کند. اول اینکه کلاس‌های مجموعه غیر جنریک، جنریک نیستند، و بنابراین شما امنیت نوع، کد تمیزتر و عملکرد بهتری که هنگام استفاده از کلاس‌ها مجموعه جنریک دارید را نخواهید داشت. دوم اینکه مدل شی بهتری از کلاس‌های غیر جنریک دارند. برای نمونه، متدهای کمتری مجازی هستند که باعث عملکرد بهتری می‌شود. اعضای جدیدی به مجموعه‌های جنریک اضافه شده‌اند تا کاربردهای جدیدی را فراهم کنند.

کلاس‌های مجموعه، رابط‌های زیادی را پیاده‌سازی می‌کنند و اشیایی که شما در مجموعه قرار می‌دهید می‌توانند رابط‌هایی را پیاده‌سازی کنند که کلاس-های مجموعه برای عملیات‌هایی مثل مرتب‌سازی و جستجو از آن‌ها استفاده می‌کنند. **FCL** همراه با تعدادی تعاریف رابط‌های جنریک عرضه می‌شود تا فواید جنریک‌ها هنگام استفاده از رابط‌ها نیز احساس شود. رابط‌های پرکاربرد در فضای نام **System.Collection.Generic** قرار دارند.

رابط‌های جنریک جدید جایگزین رابط‌های غیر جنریک قدیمی نیستند، در سناریوهای فراوانی، شما مجبورید از هر دوی آن‌ها استفاده کنید. علت آن، سازگاری با نسخه‌های پیشین است. برای نمونه، اگر کلاس **List<T>** فقط رابط **IList<T>** را پیاده‌سازی کند، هیچ کدی نمی‌تواند یک شی **List<DateTime>** را به عنوان یک **IList** در نظر بگیرد.

همچنین باید اشاره کنم که کلاس **System.Array**، کلاس پایه‌ی تمام نوع‌های آرایه، چندین متد استاتیک جنریک مثل **AsReadOnly**، **IndexOf**، **ForEach**، **FindLastIndex**، **FindLast**، **FindIndex**، **FindAll**، **Find**، **Exists**، **ConvertAll**، **BinarySearch**، **LastIndexOf**، **Sort**، **Resize** و **TrueForAll** را دارد. مثال زیر نشان می‌دهد برخی از این متدها شبیه به چه هستند:

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable,
    IStructuralComparable, IStructuralEquatable {

    public static void Sort<T>(T[] array);
    public static void Sort<T>(T[] array, IComparer<T> comparer);

    public static int BinarySearch<T>(T[] array, T value);
    public static int BinarySearch<T>(T[] array, T value,
        IComparer<T> comparer);
    ...
}
```

کد زیر از برخی از این متدها استفاده می‌کند:

```
public static void Main() {

    // Create & initialize a byte array
```

```

Byte[] byteArray = new Byte[] { 5, 1, 4, 2, 3 };

// Call Byte[] sort algorithm
Array.Sort<Byte>(byteArray);

// Call Byte[] binary search algorithm
Int32 i = Array.BinarySearch<Byte>(byteArray, 1);
Console.WriteLine(i); // Displays "0"
}

```

کتابخانه ی Power Collections از Wintellect

به درخواست مایکروسافت، Wintellect جهت فراهم کردن کلاس‌های مجموعه از C++ Standard Template Library برای برنامه‌نویسان CLR، کتابخانه‌ی Power Collections را تولید کرده است. این کتابخانه یک دسته از کلاس‌های مجموعه است که هر کسی می‌تواند به رایگان آن را دانلود کند. برای جزئیات بیشتر به <http://Wintellect.com> مراجعه کنید. این کلاس‌های مجموعه، خودشان جنریک بوده و استفاده‌ی زیادی از جنریک‌ها می‌کنند. جدول ۱-۱۲ لیستی از کلاس‌های مجموعه که شما در کتابخانه‌ی Power Collections می‌باید را نمایش می‌دهد:

جدول ۱-۱۲ کلاس‌های مجموعه جنریک در کتابخانه‌ی Power Collections از Wintellect

کلاس مجموعه	توضیح
BigList<T>	مجموعه‌ای از اشیاء T مرتب شده. بسیار کارآمد هنگام کار با بیش از ۱۰۰ آیتم.
Bag<T>	مجموعه‌ای از اشیاء T مرتب نشده. مجموعه، هش نشده است و تکراری نیز مجاز است.
OrderedBag<T>	مجموعه‌ای از اشیاء T مرتب شده. تکراری مجاز است.
Set<T>	مجموعه‌ای از آیتم‌های T مرتب نشده. تکراری مجاز نیست.
OrderedSet<T>	مجموعه‌ای از آیتم‌های T مرتب شده، تکراری مجاز نیست.
Deque<T>	صف دو طرفه، شبیه به یک لیست اما کاراتر از لیست در افزودن/حذف آیتم‌ها در ابتدای لیست.
OrderedDictionary<TKey, TValue>	دیکشنری که در آن کلیدها مرتب شده هستند و هر کدام می‌توانند یک مقدار داشته باشند.
MultiDictionary<TKey, TValue>	دیکشنری که در آن یک کلید می‌تواند چندین مقدار داشته باشد. کلیدها هش شده‌اند، تکراری مجاز است و آیتم‌ها مرتب نشده‌اند.
OrderedMultiDictionary<TKey, TValue>	دیکشنری که در آن کلیدها مرتب شده‌اند و هر کدام می‌تواند چندین مقدار داشته باشد (که می‌توانند آن‌ها را مرتب شده نگه دارند). کلیدهای تکراری مجاز است.

زیر ساختار جنریک‌ها

جنریک‌ها در نسخه 2.0 از CLR اضافه شدند و این کار اساسی بود که افراد زیادی برای زمانی طولانی روی آن کار کردند. به خصوص، برای آنکه جنریک‌ها بتوانند کار کنند، مایکروسافت مجبور به انجام این کارها بود:

- ساخت دستورات جدید زبان میانی (IL) که از آرگومان‌های نوع مطلع باشند.
- تغییر فرمت جدول‌های متادیتای کنونی تا نام‌های نوع و متدهای دارای پارامتر جنریک قابل بیان باشند.
- تغییر زبان‌های برنامه‌نویسی مختلف (سی‌شارپ، ویژوال بیسیک دات‌نت و ...) برای پشتیبانین از نحو جدید تا به برنامه‌نویسان اجازه‌ی تعریف و ارجاع به نوع‌ها و متدهای جنریک را بدهد.
- تغییر کامپایلرها تا دستورات IL جدید و فرمت متادیتای تغییر یافته را تولید کنند.
- تغییر کامپایلر فقط-در-لحظه (JIT) تا دستورات IL جدید که از آرگومان‌های نوع مطلعند را پردازش کرده و کد اصلی صحیح را تولید کند.

- ساخت اعضای رفلکشن جدید تا برنامه‌نویسان بتوانند نوع‌ها و اعضا را درباره‌ی داشتن پارامتر جنریک مورد بررسی قرار دهند. همچنین، رفلکشن جدید اعضایی تولید می‌کند که باید تعریف شوند تا برنامه‌نویسان بتوانند تعاریف نوع و متدهای جنریک را در زمان کامپایل بسازند.
- تغییر دیباگر جهت نشان دادن و دستکاری نوع‌های جنریک، اعضا، فیلدها و متغیرهای محلی.
- تغییر ویژگی IntelliSense ویژوال استودیو به منظور نمایش عضوی خاص وقتی که از یک نوع جنریک یا یک متد با نوع داده‌ای خاص استفاده می‌شود.

حال بگذارید در مورد این بحث کنیم که CLR در درون خود چگونه جنریک‌ها را مدیریت می‌کند. این اطلاعات می‌تواند در چگونگی معماری و طراحی یک الگوریتم جنریک توسط شما اثرگذار باشد. این همچنین می‌تواند بر تصمیم شما در استفاده یا عدم استفاده از یک الگوریتم جنریک موجود، اثر گذارد.

نوع‌های باز و بسته

در فصل‌های متنوعی از کتاب، من بحث کرده‌ام CLR چگونه یک ساختمان داده داخلی برای هر نوعی که برنامه از آن استفاده می‌کند، می‌سازد. این ساختمان داده‌های داخلی، اشیاء نوع `type object` نامیده می‌شوند. خوب، یک نوع با پارامترهای نوع جنریک هنوز هم یک نوع در نظر گرفته می‌شود و CLR یک شی نوع داخلی برای هر کدام از این‌ها درست می‌کند. این بر نوع‌های ارجاعی (کلاس‌ها)، نوع‌های مقداری (ساختارها)، نوع‌های رابط و نوع‌های نماینده اعمال می‌شود. هرچند، یک نوع با پارامترهای نوع جنریک یک نوع باز `open type` نامیده می‌شود و CLR اجازه نمی‌دهد هیچ نمونه‌ای از یک نوع باز ساخته شود (شبیبه به آنچه CLR از ساخت یک نمونه از یک نوع رابط جلوگیری می‌کند).

وقتی که، یک نوع جنریک را ارجاع می‌کند، آن می‌تواند مجموعه‌ای از پارامترهای نوع جنریک را تعیین نماید. اگر نوع‌های واقعی برای تمام آرگومان‌های نوع ارسال شود، نوع، یک نوع بسته `closed type` نامیده می‌شود. و CLR اجازه‌ی ساخت نمونه‌هایی از یک نوع بسته را می‌دهد. هرچند، این برای کدی که یک نوع جنریک را ارجاعی می‌کند ممکن است که تا چند آرگومان نوع جنریک را تعیین نشده رها کند. این یک نوع باز جدید در CLR ایجاد می‌کند و نمونه‌هایی از این نوع قابل ساخت نیستند. کد زیر مطلب را روشن می‌کند:

```
using System;
using System.Collections.Generic;

// A partially specified open type
internal sealed class DictionaryStringKey<TValue> :
    Dictionary<String, TValue> {
}

public static class Program {
    public static void Main() {
        Object o = null;

        // Dictionary<,> is an open type having 2 type parameters
        Type t = typeof(Dictionary<,>);

        // Try to create an instance of this type (fails)
        o = CreateInstance(t);
        Console.WriteLine();

        // DictionaryStringKey<> is an open type having 1 type parameter
        t = typeof(DictionaryStringKey<>);

        // Try to create an instance of this type (fails)
        o = CreateInstance(t);
        Console.WriteLine();
    }
}
```



```

// DictionaryStringKey<Guid> is a closed type
t = typeof(DictionaryStringKey<Guid>);

// Try to create an instance of this type (succeeds)
o = CreateInstance(t);

// Prove it actually worked
Console.WriteLine("Object type=" + o.GetType());
}

private static Object CreateInstance(Type t) {
    Object o = null;
    try {
        o = Activator.CreateInstance(t);
        Console.WriteLine("Created instance of {0}", t.ToString());
    }
    catch (ArgumentException e) {
        Console.WriteLine(e.Message);
    }
    return o;
}
}

```

وقتی من کد فوق را کامپایل و اجرا می‌کنم، خروجی زیر را می‌گیرم:

```

Cannot create an instance of System.Collections.Generic.
Dictionary`2[TKey,TValue] because Type.ContainsGenericParameters is true.

```

```

Cannot create an instance of DictionaryStringKey`1[TValue] because
Type.ContainsGenericParameters is true.

```

```

Created instance of DictionaryStringKey`1[System.Guid]
Object type=DictionaryStringKey`1[System.Guid]

```

همانطور که می‌توانید ببینید، متد **CreateInstance** از **Activator** وقتی شما از آن بخواهید یک نمونه از یک نوع باز بسازد، یک اکسپشن **ArgumentException** تولید می‌کند. در واقع، پیام اکسپشن می‌گوید که نوع هنوز دارای چند پارامتر جنریک است.

در خروجی، شما متوجه خواهید شد که نام نوع‌ها با یک (') و به دنبال آن یک عدد، همراه است. این عدد، **arity** نوع را نشان می‌دهد که تعداد پارامترهای نوع که نوع نیاز دارد را بیان می‌کند. برای نمونه، کلاس **DictionaryStringKey** دارای **arity** مقدار ۲ است چرا که آن نیاز دارد نوع‌هایی برای **TKey** و **TValue** تعیین شود. کلاس **DictionaryStringKey** دارای **arity** با مقدار ۱ است چون آن نیاز دارد که تنها یک نوع برای **TValue** تعیین شود.

همچنین باید اشاره کنم که CLR، فیلدهای استاتیک یک نوع را درون شی نوع اختصاص می‌دهد (همانگونه که در فصل ۴ "مبانی نوع" بحث کردم)، پس هر نوع بسته دارای فیلدهای استاتیک خودش است، به بیان دیگر، اگر **List<T>** فیلد استاتیکی تعریف کرده باشد، این فیلدها بین **List<DateTime>** و **List<String>** به اشتراک گذاشته نمی‌شوند؛ هر شی نوع بسته، فیلدهای استاتیک خودش را دارد. همچنین اگر یک نوع جنریک یک سازنده استاتیک (بحث شده در فصل ۸ "متدها") تعریف کند، این سازنده به ازای هر نوع بسته، یکبار اجرا می‌شود. گاهی افراد، یک سازنده استاتیک برای یک نوع جنریک تعریف می‌کنند تا مطمئن شوند که آرگومان‌های نوع شرایط خاصی را دارند. برای نمونه، اگر می‌خواهید یک نوع جنریک تعریف کنید که تنها با نوع‌های شمارشی استفاده شود، می‌توانید شبیه کد زیر را بنویسید:

```

internal sealed class GenericTypeThatRequiresAnEnum<T> {
    static GenericTypeThatRequiresAnEnum() {

```

```

    if (!typeof(T).IsEnum) {
        throw new ArgumentException("T must be an enumerated type");
    }
}
}

```

CLR قابلیت به نام محدودیت‌ها **constraints** دارد که روش بهتری به شما ارائه می‌کند تا یک نوع جنریک تعریف کنید که تعیین کند چه آرگومان‌های نوعی برای آن مجازند. من محدودیت‌ها را در انتهای فصل توضیح می‌دهم. متأسفانه، محدودیت‌ها از این قابلیت پشتیبانی نمی‌کنند که یک آرگومان نوع فقط از نوع شمارشی باشد، که به همین دلیل است که مثال قبلی نیاز به سازنده‌ی استاتیک دارد تا مطمئن شود که نوع، یک نوع شمارشی است.

نوع‌های جنریک و وراثت

یک نوع جنریک یک نوع است و به همین خاطر می‌تواند از هر نوع دیگری مشتق شود. وقتی شما از یک نوع جنریک استفاده می‌کنید و پارامترهای نوع را تعیین می‌نمایید، شما یک شی نوع جدید در CLR تعریف می‌کنید و شی نوع جدید از هر آنچه نوع جنریک از آن مشتق شده است، مشتق می‌شود. به بیان دیگر، چون **List<T>** از **Object** مشتق شده است، **List<String>** و **List<Guid>** نیز از **Object** مشتق می‌شوند. به طریق مشابه، چون **DictionaryStringKey<TValue>** از **Dictionary<String, TValue>** مشتق شده است، **DictionaryStringKey<Guid>** نیز از **Dictionary<String, Guid>** مشتق می‌شود. درک این نکته که تعیین پارامترهای نوع، کاری با سلسله مراتب وراثت ندارد، به شما کمک می‌کند تا سازماندهی کنید که چه تبدیل‌هایی را می‌توانید و کدام را نمی‌توانید انجام دهید.

برای نمونه، اگر یک کلاس لیست پیوندی شبیه به این تعریف شده باشد:

```

internal sealed class Node<T> {
    public T m_data;
    public Node<T> m_next;

    public Node(T data) : this(data, null) {
    }

    public Node(T data, Node<T> next) {
        m_data = data; m_next = next;
    }

    public override string ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
}

```

آنگاه من می‌توانم کدی بنویسم که یک لیست پیوندی بسازد:

```

private static void SameDataLinkedList() {
    Node<Char> head = new Node<Char>('C');
    head = new Node<Char>('B', head);
    head = new Node<Char>('A', head);
    Console.WriteLine(head.ToString());
}

```

در کلاس **Node** نشان داده شده، فیلد **m_next** باید به گره‌ی دیگری اشاره کند که نوع داده‌ای آن با نوع فیلد **m_data** ی آن یکسان باشد. این یعنی لیست پیوندی باید شامل گره‌هایی باشد که تمام آیتم‌های داده از یک نوع (یا نوع مشتق شده) باشند. برای نمونه، من نمی‌توانم از کلاس **Node** برای ساخت یک لیست پیوندی که در آن یک عنصر حاوی یک **Char**، عنصر دیگر حاوی یک **DateTime** و عنصر دیگر حاوی یک **String** باشد، استفاده

کنم. خوب، من می‌توانم این کار را بکنم اگر از **Node<Object>** در هر جایی استفاده می‌کردم اما در عوض امنیت نوع در زمان کامپایل را از دست می‌دادم و نوع‌های مقداری، بسته‌بندی می‌شدند.

پس راه بهتر این است که یک کلاس پایه غیرجنریک **Node** تعریف کنم و سپس یک کلاس جنریک **TypedNode** (با استفاده از کلاس **Node** به عنوان کلاس پایه) تعریف نمایم. اکنون، من می‌توانم یک لیست پیوندی داشته باشم که هر گره از نوعی خاص باشد (نه **Object**)، امنیت نوع در زمان کامپایل را داشته باشم و از بسته‌بندی نوع‌های مقداری خودداری کنم. تعاریف جدید کلاس‌ها اینگونه است:

```
internal class Node {
    protected Node m_next;

    public Node(Node next) {
        m_next = next;
    }
}

internal sealed class TypedNode<T> : Node {
    public T m_data;

    public TypedNode(T data) : this(data, null) {
    }

    public TypedNode(T data, Node next) : base(next) {
        m_data = data;
    }

    public override String ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
```

من اکنون می‌توانم کدی بنویسم که یک لیست پیوندی که در آن هر گره از یک نوع متفاوت است بسازد. کد شبیه به این خواهد بود:

```
private static void DifferentDataLinkedList() {
    Node head = new TypedNode<Char>('.');
    head = new TypedNode<DateTime>(DateTime.Now, head);
    head = new TypedNode<String>("Today is ", head);
    Console.WriteLine(head.ToString());
}
```

هویت یک نوع جنریک

گاهی نحو جنریک برنامه‌نویسان را گیج می‌کند. گذشته از این، تعداد زیادی علامت کوچکتر از (<) و بزرگتر از (>) در سراسر کد شما پخش شده است و این خوانایی کد را کاهش می‌دهد. برای بهتر کردن نحو، برخی برنامه‌نویسان یک کلاس غیر جنریک جدید تعریف می‌کنند که از یک نوع جنریک مشتق شده است و تمام آرگومان‌های نوع آن را تعیین کرده است. برای نمونه، برای ساده کردن کدی مثل این:

```
List<DateTime> dtl= new List<DateTime>();
```

برخی برنامه‌نویسان ابتدا کلاسی شبیه به این تعریف می‌کنند:

```
internal sealed class DateTimeList : List<DateTime> {
    // No need to put any code in here!
}
```

اکنون، کدی که یک لیست را می‌سازد آسانتر (بدون علامت‌های کوچکتر از و بزرگتر از) بازنویسی می‌شود:

```
DateTimeList dtl = new DateTimeList();
```

اگرچه این راحت تر به نظر می‌رسد، مخصوصاً اگر شما از نوع‌های جدید برای پارامترها، متغیرهای محلی و فیلد استفاده می‌کنید، شما نباید هرگز یک کلاس جدید را صریحاً به منظور خواناتر کردن کدتان تعریف کنید. دلیل آنست که هویت و برابری نوع را از دست می‌دهید، همانگونه که در کد زیر می‌توانید ببینید:

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

وقتی کد فوق اجرا می‌شود، **SomeType** با **False** مقداردهی اولیه می‌شود چون شما دو شی با نوع مختلف را مقایسه می‌کنید. این همچنین یعنی متدی که برای دریافت یک **DateTimeList** تعریف شده است نمی‌تواند یک **List<DateTime>** را بپذیرد. هرچند متدی که برای دریافت یک **List<DateTime>** تعریف شده است می‌تواند یک **DateTimeList** ارسالی را دریافت کند، چون **DateTimeList** از **List<DateTime>** مشتق شده است. برنامه‌نویسان به راحتی با این مطلب گیج می‌شوند.

خوشبختانه، سی‌شارپ برای استفاده از نحو ساده شده جهت ارجاع به یک نوع جنریک بسته، راهی ارائه می‌کند که اصلاً بر برابری نوع اثر نمی‌گذارد. شما می‌توانید از دستور **using** قدیمی در بالای فایل سورس کدتنا استفاده کنید. مثال را ببینیم:

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

در اینجا دستور **using** فقط یک نماد به نام **DateTimeList** تعریف می‌کند. وقتی کد کامپایل می‌شود، کامپایلر هر جا در سورس کد **DateTimeList** را ببیند، آن را با **System.Collections.Generic.List<System.DateTime>** تعویض می‌کند. این به برنامه‌نویس اجازه می‌دهد که از نحو ساده بدون اثر بر معنی واقعی کد استفاده کند و بنابراین هویت و برابری نوع حفظ می‌شود. پس اکنون، وقتی کد زیر اجرا شود، **SameType** به **true** مقداردهی اولیه می‌گردد:

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

برای راحتی بیشتر، شما می‌توانید از ویژگی متغیر محلی با نوع ضمنی سی‌شارپ استفاده کنید، جاییکه کامپایلر نوع متغیر محلی یک متد را از نوع عبارتی که به آن انتساب می‌کنید، استنتاج می‌کند:

```
using System;
```

```
using System.Collections.Generic;
```

```
...
```

```
internal sealed class SomeType {
```

```
    private static void SomeMethod () {
```

```
        // Compiler infers that DateTimeList is of type
```

```
        // System.Collections.Generic.List<System.DateTime>
```

```
        var dtl = List<DateTime>();
```

```
        ...
```

```
    }
```

```
}
```

انفجار کد

وقتی یک متد که از پارامترهای نوع جنریک استفاده می‌کند، JIT کامپایل می‌شود، CLR، IL متد را برداشته، آرگومان‌های نوع تعیین شده را جایگزینی کرده و سپس کد اصلی مخصوص آن متد که بر روی نوع‌های داده‌ای تعیین شده عمل می‌کند را تولید می‌نماید. این، حقیقتاً چیزی است که شما می‌خواهید و یکی از ویژگی‌های اصلی جنریک‌هاست. هرچند، اشکالی بر آن وارد است: CLR برای هر ترکیب از متد/نوع، کد اصلی تولید می‌کند. این با عنوان انفجار کد **code explosion** اطلاق می‌شود. این کار، حجم کاری برنامه را افزایش می‌دهد و بر عملکرد آن اثر می‌گذارد.

خوشبختانه، CLR بهینه‌سازی‌هایی برای کاهش انفجار کد انجام می‌دهد. اول اینکه اگر یک متد برای یک آرگومان نوع خاص فراخوانی شود و بعداً متد مجدداً با همان آرگومان نوع فراخوانی گردد، CLR کد این ترکیب متد/نوع را فقط یکبار کامپایل می‌کند. پس اگر یک اسمبلی از **List<DateTime>** استفاده کند و اسمبلی کاملاً متفاوتی (بارگذاری شده در همان **AppDomain**) هم از **List<DateTime>** استفاده کند، CLR متدها را برای **List<DateTime>** فقط یکبار کامپایل می‌کند. این کار به صورت قابل ملاحظه‌ای انفجار کد را کاهش می‌دهد.

CLR بهینه سازی دیگری نیز دارد: CLR تمام آرگومان های نوع ارجاعی را برابر در نظر می گیرد و باز هم کد به اشتراک گذاشته می شود. برای نمونه، کدی که CLR برای متدهای **List<String>** کامپایل می کند، می تواند برای متدهای **List<Stream>** نیز استفاده شود چون **String** و **Stream** هر دو نوع ارجاعی هستند. در واقع، برای هر نوع ارجاعی، کد یکسانی استفاده می شود. CLR می تواند این بهینه سازی را انجام دهد چون تمام آرگومان یا متغیرهای نوع ارجاعی فقط اشاره گرهایی (در سیستم ویندوز ۳۲ بیتی همه ۳۲ بیتی و در سیستم ویندوز ۶۴ بیتی همه ۶۴ بیتی) به اشیاء درون هیپ هستند و اشاره گرهای شی همه به یک روش استفاده می شوند. اما برای هر آرگومان نوعی که یک نوع مقداری باشد، CLR باید کد اصلی مخصوص به نوع مقداری تولید کند. علت اینست که نوع های مقداری اندازه های متفاوتی دارند. و حتی اگر دو نوع مقداری اندازه ی یکسان داشته باشند (مثل **Int32** و **UInt32** که هر دو ۳۲ بیتی هستند)، CLR باز هم نمی تواند کد را به اشتراک بگذارد چون دستورات اصلی پردازنده متفاوتی برای دستکاری هر کدام از این نوع ها می تواند استفاده شود.

رابط های جنریک

بدیهی است که قابلیت تعریف نوع های مقداری و ارجاعی جنریک، ویژگی اصلی جنریک هاست. اما، برای CLR حیاتی است که استفاده از رابط های جنریک را مجاز کند. بدون رابط های جنریک، هر بار که شما بخواهید یک نوع مقداری را بوسیله یک رابط غیر جنریک (مثل **IComparable**) دستکاری کنید، مجددا بسته بندی و از دست دادن امنیت نوع در زمان کامپایل رخ می دهد. این به شدت سودمندی نوع های جنریک را محدود می کند. پس CLR از رابط های جنریک پشتیبانی می کند. یک نوع مقداری یا ارجاعی می تواند یک رابط جنریک را با تعیین آرگومان های نوع، پیاده سازی کند، یا یک نوع می تواند یک رابط جنریک را با رها کردن و تعیین نکردن آرگومان های نوع پیاده سازی کند. به تعدادی مثال نگاه کنیم.

در اینجا، تعریف یک رابط جنریک که به عنوان بخشی از FCL (در فضای نام **System.Collections.Generic**) عرضه می شود، آمده است:

```
public interface IEnumerable<T> : IDisposable, IEnumerator {
    T Current { get; }
}
```

مثالی از یک نوع که این رابط جنریک را پیاده سازی و آرگومان های نوع را تعیین کرده است را در زیر می بینیم. توجه کنید که شی **Triangle** می تواند مجموعه ای از اشیاء **Object** را بشمارد. همچنین دقت کنید که ویژگی **Current** از نوع **Point** است:

```
internal sealed class Triangle : IEnumerable<Point> {
    private Point[] m_vertices;

    // IEnumerable<Point>'s Current property is of type Point
    public Point Current { get { ... } }

    ...
}
```

حال بگذارید به مثالی نگاه کنیم که یک نوع، همان رابط جنریک را پیاده سازی کرده است اما آرگومان های نوع را تعیین نکرده رها نموده است:

```
internal sealed class ArrayEnumerator<T> : IEnumerable<T> {
    private T[] m_array;

    // IEnumerable<T>'s Current property is of type T
    public T Current { get { ... } }

    ...
}
```

توجه داشته باشید که یک شی **ArrayEnumerator** می تواند مجموعه ای از اشیاء **T** را بشمارد (که **T** تعیین نشده است و به سورس کدی که از نوع **ArrayEnumerator** استفاده می کند اجازه می دهد که بعداً یک نوع برای **T** تعیین کند). همچنین دقت کنید که ویژگی **Current** اکنون از نوع تعیین نشده **T** است. اطلاعات بسیار بیشتری درباره ی رابط های جنریک در فصل ۱۳ "رابط ها" آمده است.

نماینده های جنریک

CLR از نماینده های جنریک پشتیبانی می کند تا مطمئن شود هر نوعی از شی می تواند به یک متد کالیک با روشی نوع-امن ارسال شود. علاوه بر این، نماینده های جنریک اجازه می دهند یک نمونه نوع مقداری بدون بسته بندی به یک متد کالیک ارسال شود. همانگونه که در فصل ۱۷ "نماینده ها" بحث می شود، یک نماینده (delegate) فقط یک تعریف کلاس با چهار متد است: یک سازنده، یک متد **Invoke**، یک متد **BeginInvoke** و یک متد **EndInvoke**. وقتی شما یک نوع نماینده که پارامترهای نوع را تعیین می کند تعریف می نمایید، کامپایلر متدهای کلاس نماینده را تعریف می کند و پارامترهای نوع را به هر متدی که پارامتر/مقدار برگشتی از پارامتر نوع تعیین شده دارد، اعمال می کند.

برای نمونه، اگر شما یک نماینده ی جنریک شبیه به این تعریف کنید:

```
public delegate TReturn CallMe<TReturn, TKey, TValue>(TKey key, TValue value);
```

کامپایلر آن را به کلاسی که منطقاً شبیه به کلاس زیر است، تبدیل می کند:

```
public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {
    public CallMe(Object object, IntPtr method);
    public virtual TReturn Invoke(TKey key, TValue value);
    public virtual IAsyncResult BeginInvoke(TKey key, TValue value,
        AsyncCallback callback, Object object);
    public virtual TReturn EndInvoke(IAsyncResult result);
}
```

نکته توصیه می شود هر جا ممکن است از نماینده های **Action** و **Func** جنریک که در FCL از پیش تعریف شده اند، استفاده کنید. من این نوع-های نماینده را در بخش "قبلاً به اندازه ی کافی با تعاریف نماینده ها (نماینده های جنریک) بوده ایم" از فصل ۱۷ "نماینده ها" توضیح می دهم.

آرگومان های نوع جنریک Contravariant و Covariant رابط ها و نماینده ها

هر یک از پارامترهای نوع جنریک از یک نماینده می توانند به عنوان Covariant یا Contravariant علامت زده شوند. این ویژگی به شما اجازه می دهد که یک متغیر از نوع نماینده جنریک را به همان نوع نماینده تبدیل کنید در حالیکه نوع های پارامتر جنریک متفاوت هستند. یک پارامتر نوع جنریک می تواند هر یک از این موارد باشد:

- **Invariant** یعنی پارامتر نوع جنریک نمی تواند تغییر کند. تاکنون در این فصل من فقط پارامترهای نوع جنریک این چینی را به شما نشان داده ام.
- **Contravariant** یعنی پارامتر نوع جنریک می تواند از یک کلاس به کلاسی مشتق شده از آن تغییر کند. در سی شارپ، شما پارامترهای نوع جنریک Contravariant را با کلمه کلیدی **in** مشخص می کنید. پارامترهای نوع جنریک Contravariant در مکان های ورودی مثل آرگومان یک متد می توانند ظاهر شوند.
- **Covariant** یعنی آرگومان نوع جنریک می تواند از یک کلاس به یکی از کلاس های پایه اش تغییر کند. در سی شارپ، شما پارامترهای نوع جنریک Covariant را با کلمه کلیدی **out** نشان می دهید. پارامترهای نوع جنریک Covariant می توانند تنها در مکان ها خروجی مثل نوع برگشتی یک متد ظاهر شوند.

برای نمونه، تعریف نوع نماینده ی زیر وجود دارد:

```
public delegate TResult Func<in T, out TResult>(T arg);
```

در اینجا، پارامتر نوع جنریک **T** با کلمه کلیدی **in** علامت زده شده است که آن را Contravariant می کند. پارامتر نوع جنریک **TResult** با کلمه کلیدی **out** علامت زده شده است که آن را Covariant می کند:

پس اکنون، اگر من متغیری مثل زیر تعریف کنم:

```
Func<Object, ArgumentException> fn1 = null;
```

می توانم آن را به نوع **Func** دیگری که پارامتر نوع جنریک آن متفاوت است تبدیل کنم:

```
Func<String, Exception>fn2 = fn1;// No explicit cast is required here
```

```
Exception e = fn2("");
```

آنچه این کد می‌گوید اینست که **fn1** به تابعی که یک **Object** دریافت و یک **ArgumentException** برمی‌گرداند، اشاره دارد. متغیر **fn2** می‌خواهد به متدی که یک **String** گرفته و یک **Exception** برمی‌گرداند اشاره کند. چون شما می‌خواهید یک **String** را به متدی ارسال کنید که یک **Object** می‌خواهد (چون **String** از **Object** مشتق شده است) و چون شما می‌خواهید نتیجه‌ی یک متد که یک **ArgumentException** برمی‌گرداند را گرفته و به عنوان یک **Exception** با آن رفتار کنید و چون **Exception** کلاس پایه‌ی **ArgumentException** است، کد فوق کامپایل می‌شود و امنیت نوع در زمان کامپایل را حفظ می‌کند.

نکته واریانس (انتساب به نماینده‌ای با پارامتر جنریک متفاوت) تنها در صورتی اعمال می‌شود که کامپایلر بتواند تعیین کند که یک تبدیل ارجاع بین نوع‌های وجود دارد. به بیان دیگر، واریانس برای نوع‌های مقداری غیر ممکن است چون بسته بندی نیاز خواهد شد. به نظر من، این محدودیت باعث می‌شود، ویژگی‌های واریانس آنقدر هم سودمند نباشد. برای نمونه، اگر من متد زیر را داشته باشم:

```
void ProcessCollection(IEnumerable<Object> collection) { ... }
```

نمی‌توانم آن را با ارسال یک ارجاع به شی **List<DateTime>** فراخوانی کنم چون یک تبدیل ارجاع بین نوع مقداری **DateTime** و **Object** وجود ندارد، هرچند که **DateTime** از **Object** مشتق شده است.

شما این مشکل را با تعریف **ProcessCollection** به شکل زیر حل می‌کنید:

```
void ProcessCollection<T>(IEnumerable<T> collection) { ... }
```

به اضافه، مزیت بزرگ **ProcessCollection(IEnumerable<Object> Collection)** اینست که تنها یک کد JIT شده وجود دارد. هرچند با **ProcessCollection<T>(IEnumerable<T> Collection)** نیز تنها یک کد JIT شده که توسط تمام **T** هایی که نوع ارجاعی هستند به اشتراک گذاشته می‌شود. شما برای **T** هایی که نوع مقداری هستند نسخه‌های دیگری از کد JIT را خواهید داشت. اما اکنون حداقل شما می‌توانید متد را با ارسال یک مجموعه از نوع‌های مقداری فراخوانی کنید.

همچنین واریانس روی یک پارامتر نوع جنریک مجاز نیست اگر آرگومانی از آن نوع به یک متد با استفاده از کلمه کلیدی **out** یا **ref** ارسال شده است. برای نمونه، کد زیر باعث می‌شود کامپایلر خطای زیر را اعلام کند:

“Invalid variance: The type parameter ‘T’ must be invariantly valid on ‘SomeDelegate<T>.Invoke(ref T)’. ‘T’ is contravariant.”

```
delegate void SomeDelegate<in T>(ref T t);
```

هنگام استفاده از نماینده‌هایی که آرگومان و نوع برگشتی جنریک دارند، توصیه می‌شود که همیشه کلمه کلیدی **in** و **out** را برای **contravariance** و **covariance** هر جا ممکن است تعیین کنید، که انجام این کار اثر بدی نداشته و نماینده‌ی شما را قادر می‌سازد در سناریوهای بیشتری استفاده شود. همانند نماینده‌ها، یک رابط با پارامترهای نوع جنریک می‌تواند پارامترهای نوع **contravariant** یا **covariant** خودش را داشته باشد. نمونه‌ای از یک رابط با یک پارامتر نوع جنریک **contravariant**:

```
public interface IEnumerator<out T> : IEnumerator {
    Boolean MoveNext();
    T Current { get; }
}
```

چون **T** **contravariant** است کامپایلر کد زیر ممکن است و آن با موفقیت اجرا می‌شود:

```
// This method accepts an IEnumerable of any reference type
Int32 Count(IEnumerable<Object> collection) { ... }
```

```
...
```

```
// The call below passes an IEnumerable<String> to Count
```

```
Int32 c = Count(new[] { "Grant" });
```

مهم گاهی اوقات برنامه‌نویسان می‌پرسند چرا باید صریحا **in** یا **out** را روی پارامترهای نوع جنریک بگذارند. آن‌ها فکر می‌کنند کامپایلر باید بتواند تعریف نماینده یا رابط را بررسی کرده و به صورت خودکار تشخیص دهد کدام پارامترهای نوع جنریک می‌توانند **contravariant** و **covariant** باشند. در حالیکه این درست است که کامپایلر می‌تواند به صورت خودکار این کار را بکند، تیم سی‌شارپ اعتقاد داشتند که شما یک قرارداد تعریف می‌کنید و باید صریحا بگویید چه چیزهایی را اجازه می‌دهید. برای نمونه، این بد است اگر کامپایلر تعیین کند یک پارامتر نوع جنریک می‌تواند **contravariant** باشد و در آینده شما عضوی به یک رابط بیافزایید که پارامتر نوع را در مکان خروجی استفاده کند. دفعه‌ی بعد که کد را کامپایل کنید، کامپایلر تعیین می‌کند نوع پارامتر باید ثابت باشد اما تمام کدهایی که به دیگر اعضا ارجاع داده اند، ممکن است خطایی تولید کنند اگر از این حقیقت که پارامتر نوع، **contravariant** بوده، استفاده کرده باشند.

به همین علت، تیم کامپایلر، شما را مجبور می‌کند هنگام تعریف یک پارامتر نوع جنریک صریح باشید. آنگاه، اگر شما سعی کنید از این پارامتر نوع در محیطی که با آنچه تعریف شده، تطابق ندارد استفاده کنید، کامپایلر خطایی اعلام می‌کند تا بدانید که سعی شما باعث نقض قرارداد می‌شود.

اگر سپس سعی کنید که قرارداد را با افزودن **in** یا **out** به پارامترهای نوع جنریک، نقض کنید، باید انتظار تغییر کدهایی که از قرارداد قبلی استفاده می‌کنند را داشته باشید.

متدهای جنریک

وقتی شما یک کلاس، ساختار یا رابط را جنریک تعریف می‌کنید، هر متدی که در این نوع‌ها تعریف شده است، می‌تواند به یک پارامتر نوع که توسط نوع تعیین شده است، ارجاع کند. یک پارامتر نوع می‌تواند به عنوان پارامتر یک متد، مقدار برگشتی یک متد یا متغیر محلی تعریف شده درون متد استفاده شود. هرچند CLR از قابلیت پشتیبانی می‌کند که یک متد بتواند پارامترهای نوع خودش را تعیین کند. و این پارامترهای نوع نیز می‌توانند برای پارامترها، نوع برگشتی یا متغیرهای محلی استفاده شوند. مثال زیر نمونه‌ای است که یک نوع، یک پارامتر نوع و یک متد که پارامتر نوع خودش را دارد، تعریف می‌کند:

```
internal sealed class GenericType<T> {
    private T m_value;

    public GenericType(T value) { m_value = value; }

    public TOutput Converter<TOutput>() {
        TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
        return result;
    }
}
```

در این مثال، همانگونه که می‌بینید، کلاس **GenericType** پارامتر نوع خودش (**T**) را تعریف کرده و متد **Converter** پارامتر نوع خودش (**TOutput**) را تعریف کرده است. این کار اجازه می‌دهد یک **GenericType** برای کار با هر نوع ساخته شود. متد **Converter** می‌تواند شی ارجاع شده توسط فیلد **m_value** را به نوع‌های مختلف بسته به آنچه آرگومان نوع هنگام فراخوانی ارسال شده باشد، تبدیل کند. قابلیت داشتن پارامترهای نوع و پارامترهای متد، انعطاف پذیری زیادی را به همراه دارد.

یک مثال خوب از یک متد جنریک، متد **Swap** است:

```
private static void Swap<T>(ref T o1, ref T o2) {
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
```


کد می‌تواند اکنون **Swap** را اینگونه فراخوانی کند:

```
private static void CallingSwap() {
    Int32 n1 = 1, n2 = 2;
    Console.WriteLine("n1={0}, n2={1}", n1, n2);
    Swap<Int32>(ref n1, ref n2);
    Console.WriteLine("n1={0}, n2={1}", n1, n2);

    String s1 = "Aidan", s2 = "Grant";
    Console.WriteLine("s1={0}, s2={1}", s1, s2);
    Swap<String>(ref s1, ref s2);
    Console.WriteLine("s1={0}, s2={1}", s1, s2);
}
```

استفاده از نوع‌های جنریک مخصوصا با متدهایی که پارامترهای **out** و **ref** می‌گیرند، جالب است چون متغیری که شما به عنوان آرگومان **out/ref** ارسال می‌کنید باید از همان نوع پارامتر متد باشد تا امنیت نوع از دست نرود. این مسئله مرتبط با پارامترهای **out/ref** در اواخر بخش "ارسال پارامترها با ارجاع به یک متد" در فصل ۹ "پارامترها" بحث شده است. در حقیقت، متدهای **Exchange** و **CompareExchange** از کلاس **Interlocked** سربارگذاری جنریک را دقیقا به همین منظور ارائه می‌کند^{۵۱}:

```
public static class Interlocked {
    public static T Exchange<T>(ref T location1, T value) where T: class;
    public static T CompareExchange<T>(
        ref T location1, T value, T comparand) where T: class;
}
```

متدهای جنریک و استنتاج نوع

برای بسیاری از برنامه‌نویسان، نحو جنریک سی‌شارپ با علامت‌های کوچکتر از و بزرگتر از، گیج‌کننده است. برای بهبود ساخت کد، خوانایی و نگهداری آن، کامپایلر سی‌شارپ هنگام فراخوانی یک متد جنریک، استنتاج نوع **inference type** را ارائه می‌کند. استنتاج نوع یعنی کامپایلر سعی می‌کند نوع را به صورت خودکار هنگام فراخوانی یک متد، تعیین (یا استنتاج) کند. کد زیر استنتاج را نشان می‌دهد:

```
private static void CallingSwapUsingInference() {
    Int32 n1 = 1, n2 = 2;
    Swap(ref n1, ref n2); // Calls Swap<Int32>

    String s1 = "Aidan";
    Object s2 = "Grant";
    Swap(ref s1, ref s2); // Error, type can't be inferred
}
```

در این کد، دقت کنید در فراخوانی **Swap** آرگومان‌های نوع را در بین علامت‌های کوچکتر از و بزرگتر از تعیین نمی‌کند. در اولین فراخوانی به **Swap**، کامپایلر سی‌شارپ قادر است استنتاج کند که **n1** و **n2** از نوع **Int32** هستند و بنابراین باید **Swap** را با یک آرگومان نوع **Int32** فراخوانی کند.

هنگام انجام استنتاج نوع، سی‌شارپ از نوع داده‌ای متغیر و نه نوع واقعی شی که توسط متغیر ارجاع می‌شود، استفاده می‌کند. در دومین فراخوانی به **Swap**، سی‌شارپ می‌بیند که **s1** یک رشته و **s2** یک **Object** است (اگرچه به یک **String** اشاره دارد). چون **s1** و **s2** متغیرهایی از نوع‌های داده‌ای مختلف هستند، کامپایلر نمی‌تواند دقیقا نوع را برای آرگومان نوع **Swap**، استنتاج کند و پیام زیر را اعلام می‌نماید.

“error CS0411: The type arguments for method ‘Program.Swap<T>(ref T, ref T)’ cannot be inferred from the usage. Try specifying the type arguments explicitly.”

یک نوع می‌تواند چندین متد تعریف کند که یکی، یک نوع داده‌ای خاص را بگیرد و دیگری یک پارامتر نوع جنریک، همانند مثال زیر:

^{۵۱} عبارت **where** در بخش "قابلیت بازبینی و محدودیت‌ها" در اواخر فصل بحث می‌شود.

```
private static void Display(String s) {
    Console.WriteLine(s);
}

private static void Display<T>(T o) {
    Display(o.ToString()); // Calls Display(String)
}
```

برخی روش‌های فراخوانی متد **Display** اینگونه است:

```
Display("Jeff");           // Calls Display(String)
Display(123);              // Calls Display<T>(T)
Display<String>("Aidan"); // Calls Display<T>(T)
```

در اولین فراخوانی، کامپایلر می‌تواند متد **Display** که یک **String** یا یک متد جنریک **Display** (با جایگزینی **T** با **String**) را فراخوانی کند. هرچند، کامپایلر سی‌شارپ همواره تطبیق صریح‌تر را بر یک تطبیق جنریک ترجیح می‌دهد و بنابراین، یک فراخوانی به متد غیرجنریک **Display** که یک **String** می‌گیرد را تولید می‌نماید. برای فراخوانی دوم، کامپایلر نمی‌تواند متد غیرجنریک **Display** را که یک **String** می‌گیرد فراخوانی کند، پس باید متد جنریک **Display** را فراخوانی کند. ضمناً، از خوش شانس است که کامپایلر همیشه تطبیق صریح‌تر را ترجیح می‌دهد، اگر کامپایلر متد جنریک را ترجیح می‌داد، چون متد جنریک **Display** مجدداً **Display** را فراخوانی می‌کند (اما با یک **String** برگشتی از **ToString**)، بازگشتی بی‌نهایت رخ خواهد می‌داد. سومین فراخوانی به **Display** یک آرگومان نوع جنریک، **String**، را تعیین می‌کند. این به کامپایلر می‌گوید که سعی نکند آرگومان‌های نوع را استخراج کند و به جای آن از آرگومان‌های نوعی که صریحاً تعیین شده است استفاده کند. در این حالت کامپایلر، فرض می‌کند واقعا می‌خواهم متد جنریک **Display** را فراخوانی کنم. پس متد جنریک **Display** فراخوانی خواهد شد. در درون، متد جنریک **Display**، **ToString** را روی رشته‌ی ارسالی فراخوانی می‌کند، که نتیجه‌ی آن، یک رشته، به متد غیرجنریک **Display** ارسال می‌گردد.

جنریک‌ها و دیگر اعضا

در سی‌شارپ، ویژگی‌ها، ایندکسرها، رویدادها، متدهای عملگر، سازنده‌ها و مخرب‌ها خودشان نمی‌توانند پارامترهای نوع داشته باشند. هرچند، آن‌ها می‌توانند درون یک نوع جنریک تعریف شوند و کد درون این اعضا می‌تواند از پارامترهای نوع متعلق به نوع استفاده کند. سی‌شارپ اجازه نمی‌دهد این اعضا پارامترهای نوع جنریک خودشان را تعریف کنند چون تیم سی‌شارپ مایکروسافت اعتقاد دارد که برنامه‌نویسان به ندرت نیاز به استفاده از این اعضا به عنوان جنریک دارند. علاوه بر این، هزینه‌ی افزودن پشتیبانی جنریک به این اعضا جهت طراحی نحو مناسب در زبان، بسیار بالا خواهد بود. برای نمونه، وقتی شما از یک عملگر **+** در کدتان استفاده می‌کنید، کامپایلر می‌تواند یک متد سربارگذاری شده‌ی عملگر را فراخوانی کند. هیچ راهی برای تعیین یک آرگومان نوع همراه با عملگر **+** در کد شما وجود ندارد.

قابلیت بازبینی و محدودیت‌ها

هنگام کامپایل کد جنریک، کامپایلر سی‌شارپ آن را تحلیل می‌کند که کد برای هر نوعی که اکنون وجود دارد و یا در آینده ممکن است تعریف شود، کار بکند. به متد زیر نگاه کنید:

```
private static Boolean MethodTakingAnyType<T>(T o) {
    T temp = o;
    Console.WriteLine(o.ToString());
    Boolean b = temp.Equals(o);
    return b;
}
```

این متد یک متغیر محلی (**temp**) از نوع **T** تعریف می‌کند، و سپس متد، تعدادی انتساب متغیر و فراخوانی متغیر انجام می‌دهد. این متد برای هر نوعی کار می‌کند. اگر **T** یک نوع ارجاعی باشد، آن کار می‌کند. اگر **T** یک نوع مقداری یا شمارشی باشد، آن کار می‌کند. اگر **T** یک نوع رابط یا نماینده باشد، آن کار می‌کند. این متد برای تمام نوع‌هایی که اکنون وجود دارند و در آینده تعریف می‌شوند کار می‌کند چون هر نوعی از انتساب و فراخوانی به متدهای تعریف شده توسط **Object** (مثل **ToString** و **Equals**) پشتیبانی می‌کند. حال به متد زیر نگاه کنید:

```
private static T Min<T>(T o1, T o2) {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

متد **Min** سعی می‌کند از متغیر **o1** برای فراخوانی متد **CompareTo** استفاده کند. اما نوع‌های زیادی وجود دارند که یک متد **CompareTo** ارائه نمی‌کنند. و بنابراین، کامپایلر سی‌شارپ نمی‌تواند این کد را کامپایل کند و تضمین نماید که متد، برای تمام نوع‌ها کار می‌کند. اگر شما سعی کنید کد فوق را کامپایل کنید، کامپایلر خطای زیر را اعلام می‌کند:

“error CS0117: ‘T’ does not contain a definition for ‘CompareTo’.”

پس به نظر می‌رسد که هنگام استفاده از جنریک‌ها، شما می‌توانید متغیرهایی از نوع جنریک تعریف کنید، چند انتساب متغیر انجام دهید، متدهای تعریف شده توسط **Object** را فراخوانی کنید و فقط همین. این کار، جنریک‌ها را عملاً بی‌استفاده می‌کند. خوشبختانه، کامپایلرها و CLR از مکانیزمی به نام محدودیت‌ها **constraints** پشتیبانی می‌کنند که شما می‌توانید از آن‌ها استفاده کنید تا جنریک‌ها را مجدد سودمند کنید.

یک محدودیت، روشی برای محدود کردن تعداد نوع‌هایی است که می‌تواند برای یک آرگومان جنریک تعیین گردد. محدود کردن تعداد نوع‌ها اجازه می‌دهد که با آن‌ها کار بیشتری انجام دهید. نسخه‌ی جدید متد **Min** که یک محدودیت (به صورت **Bold** در آمده) تعریف می‌کند را در زیر می‌بینید:

```
public static T Min<T>(T o1, T o2) where T : IComparable<T> {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

نشانه‌ی **where** سی‌شارپ به کامپایلر می‌گوید که هر نوع تعیین شده برای **T** باید رابط جنریک **IComparable** را از همان نوع (**T**)، پیاده‌سازی کند. به خاطر این محدودیت، کامپایلر اکنون به متد اجازه می‌دهد که متد **CompareTo** را فراخوانی کند چون این متد توسط رابط **IComparable<T>** تعریف شده است.

حال، وقتی کد به یک نوع یا متد جنریک اشاره کند، کامپایلر مسئول است مطمئن شود که یک آرگومان نوع که محدودیت‌ها را داشته باشد، تعیین شده است. برای نمونه، کد زیر باعث می‌شود کامپایلر خطای زیر را اعلام کند:

“error CS0311: The type ‘object’ cannot be used as type parameter ‘T’ in the generic type or method ‘SomeType.Min<T>(T, T)’. There is no implicit reference conversion from ‘object’ to ‘System.IComparable<object>’.”

```
private static void CallMin() {
    Object o1 = "Jeff", o2 = "Richter";
    Object oMin = Min<Object>(o1, o2); // Error CS0311
}
```

کامپایلر خطا را به این خاطر اعلام می‌کند که **System.Object** رابط **IComparable<Object>** را پیاده‌سازی نمی‌کند. در واقع، **System.Object** اصلاً هیچ رابطی را پیاده‌سازی نمی‌کند. حال که شما از محدودیت‌ها و نحوه‌ی کار آن‌ها کمی می‌دانید، کمی عمیق‌تر به آن‌ها نگاه می‌کنیم. محدودیت‌ها می‌توانند بر پارامترهای نوع متعلق به یک نوع جنریک و پارامترهای نوع متعلق به یک متد جنریک (همانگونه که در متد **Min** نشان داده شد) اعمال شوند. CLR اجازه سربارگذاری بر اساس نام‌های پارامتر نوع و محدودیت‌ها را نمی‌دهد و شما می‌توانید نوع‌ها یا متدها را فقط بر اساس **arity** (تعداد آرگومان‌ها) سربارگذاری کنید. مثال زیر این را نشان می‌دهد:

```
// It is OK to define the following types:
internal sealed class AType {}
internal sealed class AType<T> {}
internal sealed class AType<T1, T2> {}

// Error: conflicts with AType<T> that has no constraints
internal sealed class AType<T> where T : IComparable<T> {}
```

```
// Error: conflicts with AType<T1, T2>
internal sealed class AType<T3, T4> {}

internal sealed class AnotherType {
    // It is OK to define the following methods:
    private static void M() {}
    private static void M<T>() {}
    private static void M<T1, T2>() {}

    // Error: conflicts with M<T> that has no constraints
    private static void M<T>() where T : IComparable<T> {}

    // Error: conflicts with M<T1, T2>
    private static void M<T3, T4>() {}
}
```

هنگام بازنویسی یک متد جنریک مجازی، متد در حال بازنویسی باید همان تعداد از پارامترهای نوع را تعیین کند و این پارامترهای نوع، محدودیت‌های تعیین شده توسط متد کلاس پایه را به ارث می‌برند. مثال زیر این قانون را با استفاده از متدهای مجازی نشان می‌دهد:

```
internal class Base {
    public virtual void M<T1, T2>()
        where T1 : struct
        where T2 : class {
    }
}

internal sealed class Derived : Base {
    public override void M<T3, T4>()
        where T3 : EventArgs // Error
        where T4 : class // Error
    { }
}
```

سعی در کامپایل کد فوق باعث می‌شود کامپایلر خطای زیر را اعلام کند:

“error CS0460: Constraints for override and explicit interface implementation methods are inherited from the base method so cannot be specified directly.”

اگر ما دو خط **where** از متد **M<T3, T4>** از کلاس **Derived** را حذف کنیم، کد به خوبی کامپایل می‌شود. توجه کنید که شما می‌توانید نام پارامترهای نوع را تغییر دهید (همانند مثال: از **T1** به **T3** و از **T2** به **T4**) اما نمی‌توانید محدودیت‌ها را تغییر (یا حتی تعیین) کنید.

حال بگذارید درباره‌ی سه گونه‌ی متفاوت از محدودیت‌ها که کامپایلر CLR اجازه‌ی اعمال آن‌ها بر یک پارامتر نوع را می‌دهد، صحبت کنیم. یک پارامتر نوع می‌تواند با یک محدودیت اولیه (اصلی) **primary constraint**، یک محدودیت ثانویه **secondary constraint** و/یا یک محدودیت سازنده **constructor constraint**، محدود شود. من درباره‌ی این سه گونه از محدودیت‌ها در سه بخش آتی صحبت می‌کنم.

محدودیت‌های اصلی

یک پارامتر نوع می‌تواند صفر محدودیت اصلی یا یک محدودیت اصلی تعیین کند. یک محدودیت اصلی می‌تواند یک نوع ارجاعی باشد که یک کلاس مهر (sealed) نشده را نشان می‌دهد. شما نمی‌توانید یکی از نوع‌های ارجاعی خاص زیر را تعیین کنید: **System.Array**، **System.Object**، **System.Delegate**، **System.MulticastDelegate**، **System.ValueType**، **System.Enum** یا **System.Void**.

هنگام تعیین یک محدودیت نوع ارجاعی، شما به کامپایلر قول می‌دهید که آرگومان نوع تعیین شده یا از همان نوع یا یکی از نوع‌های مشتق شده از نوع محدودیت باشد. برای مثال، کلاس جنریک زیر را نگاه کنید:

```
internal sealed class PrimaryConstraintOfStream<T> where T : Stream {
    public void M(T stream) {
        stream.Close();// OK
    }
}
```

در تعریف این کلاس، پارامتر نوع **T** دارای یک محدودیت اصلی از **Stream** (تعریف شده در فضای نام **System.IO**) است. این به کامپایلر می‌گوید که کدی که از **PrimaryConstraintOfStream** استفاده می‌کند باید یک آرگومان نوع از **Stream** یا یک نوع مشتق شده از **Stream** (مثل **FileStream**) را تعیین کند. اگر یک پارامتر نوع، یک محدودیت اصلی تعیین نکند، **System.Object** فرض می‌گردد. هرچند، کامپایلر سی‌شارپ یک خطا ("**error CS0702: Constraint cannot be special class 'object'**") اعلام می‌کند اگر شما صریحاً **System.Object** را در کدتان تعیین کنید.

دو محدودیت اصلی خاص وجود دارد: **class** و **struct**. محدودیت **class** به کامپایلر قول می‌دهد که آرگومان نوع تعیین شده، یک نوع ارجاعی باشد. هر نوع کلاس، نوع رابط، نوع نماینده یا نوع آرایه این محدودیت را ارضا می‌کند. برای نمونه، کلاس جنریک زیر را ببینید:

```
internal sealed class PrimaryConstraintOfClass<T> where T : class {
    public void M() {
        T temp = null;// Allowed because T must be a reference type
    }
}
```

در این مثال، **null** کردن **temp** مجاز است چون **T** مطمئناً یک نوع ارجاعی است و تمام متغیرهای نوع ارجاعی می‌توانند **null** شوند. اگر **T** بدون محدودیت بود، کد فوق کامپایل نمی‌شد چون **T** می‌توانست یک نوع مقداری باشد و متغیرهای نوع مقداری نمی‌توانند **null** شوند.

محدودیت **struct** به کامپایلر قول می‌دهد که آرگومان نوع تعیین شده، یک نوع مقداری باشد. هر نوع مقداری، شامل شمارشی‌ها، این محدودیت را ارضا می‌کنند. هرچند، کامپایلر و CLR هر نوع مقداری **<T> System.Nullable** را یک نوع خاص لحاظ کرده و نوع‌های تهی پذیر، این محدودیت را ارضا نمی‌کنند. علت اینست که نوع **<T> Nullable** پارامتر نوعش را به **struct** محدود می‌کند و CLR می‌خواهد از یک نوع بازگشتی مثل **<<Nullable<Nullable<T>>> Nullable** خودداری کند. نوع‌های تهی پذیر در فصل ۱۹ "نوع‌های مقداری تهی پذیر" بحث می‌شوند.

کلاس زیر یک مثال است که پارامتر نوعش را با محدودیت **struct** محدود می‌کند:

```
internal sealed class PrimaryConstraintOfStruct<T> where T : struct {
    public static T Factory() {
        // Allowed because all value types implicitly
        // have a public, parameterless constructor
        return new T();
    }
}
```

در این مثال، **new** کردن یک **T** مجاز است چون **T** مطمئناً یک نوع مقداری است و تمام نوع‌های مقداری به صورت ضمنی یک سازنده‌ی عمومی بدون پارامتر دارند. اگر **T** بدون محدودیت می‌بود، به یک نوع ارجاعی محدود می‌شد، یا به **class** محدود می‌شد، آنگاه کد فوق کامپایل نمی‌شد چون بعضی نوع‌های ارجاعی دارای سازنده‌ی عمومی بدون پارامتر نیستند.

محدودیت‌های ثانویه

یک پارامتر نوع می‌تواند صفر یا بیشتر محدودیت ثانویه تعیین کند که یک محدودیت ثانویه یک نوع رابط را نشان می‌دهد. هنگام تعیین یک محدودیت نوع رابط، شما به کامپایلر قول می‌دهید که پارامتر نوع تعیین شده نوعی است که رابط را پیاده‌سازی کرده است. و چون شما نمی‌توانید چندین محدودیت رابط تعیین کنید، آرگومان نوع باید نوعی را تعیین کند که تمام محدودیت‌های رابط (و تمام محدودیت‌های اصلی اگر تعیین شده باشد) را پیاده‌سازی کند. فصل ۱۳ محدودیت‌های رابط را با جزئیات بحث می‌کند.

گونه‌ی دیگری از محدودیت ثانویه وجود دارد که یک محدودیت پارامتر نوع **type parameter constraint** نامیده می‌شود (گاهی به آن **naked type constraint** نیز اطلاق می‌شود). این گونه از محدودیت بسیار کمتر از یک محدودیت رابط، استفاده می‌شود. آن اجازه می‌دهد یک نوع یا متد جنریک تعیین کند که باید رابطه‌ای بین آرگومان‌های نوع تعیین شده وجود داشته باشد. یک پارامتر نوع می‌تواند صفر یا بیشتر محدودیت نوع بر آن اعمال گردد. متد جنریک زیر استفاده از یک محدودیت پارامتر نوع را نشان می‌دهد:

```
private static List<TBase> ConvertIList<T, TBase>(IList<T> list)
    where T : TBase {
    List<TBase> baseList = new List<TBase>(list.Count);
    for (Int32 index = 0; index < list.Count; index++) {
        baseList.Add(list[index]);
    }
    return baseList;
}
```

متد **ConvertIList** دو پارامتر نوع تعیین می‌کند که پارامتر **T** با پارامتر نوع **TBase** محدود شده است. این یعنی هر آرگومان نوعی که برای **T** تعیین شود، آرگومان نوع باید با هر آرگومان نوعی که برای **TBase** تعیین می‌شود سازگار باشد. متد زیر تعدادی فراخوانی مجاز و غیر مجاز به **ConvertIList** را نشان می‌دهد:

```
private static void CallingConvertIList() {
    // Construct and initialize a List<String> (which implements IList<String>)
    IList<String> ls = new List<String>();
    ls.Add("A String");

    // Convert the IList<String> to an IList<Object>
    IList<Object> lo = ConvertIList<String, Object>(ls);

    // Convert the IList<String> to an IList<IComparable>
    IList<IComparable> lc = ConvertIList<String, IComparable>(ls);

    // Convert the IList<String> to an IList<IComparable<String>>
    IList<IComparable<String>> lcs =
    ConvertIList<String, IComparable<String>>(ls);

    // Convert the IList<String> to an IList<String>
    IList<String> ls2 = ConvertIList<String, String>(ls);

    // Convert the IList<String> to an IList<Exception>
    IList<Exception> le = ConvertIList<String, Exception>(ls); // Error
}
```

در اولین فراخوانی به **ConvertIList**، کامپایلر مطمئن می‌شود که **String** با **Object** سازگار باشد. چون **String** از **Object** مشتق شده است اولین فراخوانی محدودیت پارامتر نوع را رعایت می‌کند. در دومین فراخوانی به **ConvertIList**، کامپایلر مطمئن می‌شود که **String** با **IComparable** سازگار باشد. چون **String** رابط **IComparable** را پیاده‌سازی می‌کند، دومین فراخوانی، محدودیت پارامتر نوع را رعایت می‌کند. در سومین فراخوانی به **ConvertIList** کامپایلر مطمئن می‌شود که **String** با **IComparable<String>** سازگار باشد. چون **String** رابط **IComparable<String>** را پیاده‌سازی می‌کند، سومین فراخوانی، محدودیت پارامتر نوع را رعایت می‌کند. در چهارمین فراخوانی به **ConvertIList**، کامپایلر می‌داند که **String** یا خودش سازگار باشد. در پنجمین فراخوانی، کامپایلر مطمئن می‌شود که **String** با **Exception** سازگار باشد. چون **String** با **Exception** سازگار نیست، پنجمین فراخوانی محدودیت پارامتر نوع را رعایت نکرده و کامپایلر خطای زیر را اعلام می‌کند:

"error CS0311: The type 'string' cannot be used as type parameter 'T' in the generic type or method 'Program.ConvertToList<T,TBase>(System.Collections.Generic.IList<T>}'. There is no implicit reference conversion from 'string' to 'System.Exception'."

محدودیت های سازنده

یک پارامتر نوع می تواند صفر محدودیت سازنده یا یک محدودیت سازنده تعیین کند. هنگام تعیین یک محدودیت سازنده، شما به کامپایلر قول می دهید که آرگومان نوع تعیین شده از یک نوع غیر خلاصه (non-abstract) که یک سازنده عمومی بدون پارامتر را پیاده سازی می کند است. دقت کنید که کامپایلر سی شارپ تعیین یک محدودیت سازنده با محدودیت struct را خطا تشخیص می دهد چون اضافی است، تمام نوع های مقداری به صورت ضمنی، یک سازنده عمومی بدون پارامتر ارائه می کنند. کلاس مثال زیر پارامتر نوعش را با استفاده از محدودیت سازنده محدود کرده است:

```
internal sealed class ConstructorConstraint<T> where T : new() {
    public static T Factory() {
        // Allowed because all value types implicitly
        // have a public, parameterless constructor and because
        // the constraint requires that any specified reference
        // type also have a public, parameterless constructor
        return new T();
    }
}
```

در این مثال، **new** کردن یک **T** مجاز است چون **T** به عنوان نوعی شناخته می شود که یک سازنده عمومی بدون پارامتر دارد. این مطمئناً برای تمام نوع-های مقداری صحیح است و محدودیت سازنده نیاز دارد که آن برای هر نوع ارجاعی به عنوان پارامتر نوع نیز درست باشد.

گاهی برنامه نویسان دوست دارند یک پارامتر نوع را با استفاده از یک محدودیت سازنده تعریف کنند که در آن خود سازنده، پارامترهای مختلفی بگیرد. در حال حاضر، CLR (و بنابراین کامپایلر سی شارپ) تنها از سازنده های بدون پارامتر پشتیبانی می کنند. میکروسافت حس می کند این برای تقریباً اکثر سناریوها خوب است و من نیز موافقم.

دیگر مسائل قابلیت بازبینی

در مابقی بخش، من می خواهم به چند کد دیگر اشاره کنم که رفتار غیر منتظره ای دارند وقتی با جنریک ها استفاده می شوند که به خاطر مسائل بازبینی است و اینکه چگونه محدودیت ها می توانند استفاده شوند تا کد مجدداً قابلیت بازبینی پیدا کند.

تبدیل یک متغیر نوع جنریک

تبدیل یک متغیر نوع جنریک به نوع دیگر غیر مجاز است مگر آنکه شما به نوعی تبدیل کنید که با یک محدودیت سازگار باشد:

```
private static void CastingAGenericTypeVariable1<T>(T obj) {
    Int32 x = (Int32) obj; // Error
    String s = (String) obj; // Error
}
```

کامپایلر در هر دو خط، خطا اعلام می کند چون **T** می تواند هر نوعی باشد، و هیچ تضمینی وجود ندارد که تبدیل موفقیت آمیز باشد. شما می توانید این کد را برای کامپایل شدن اینگونه تغییر دهید که ابتدا آن را به یک **Object** تبدیل کنید:

```
private static void CastingAGenericTypeVariable2<T>(T obj) {
    Int32 x = (Int32) (Object) obj; // No error
    String s = (String) (Object) obj; // No error
}
```

اگرچه این کد اکنون اجرا می شود، این ممکن است که CLR در زمان اجرا یک **InvalidCastException** تولید کند.

اگر شما سعی دارید به یک نوع ارجاعی تبدیل کنید، می توانید از عملگر **as** سی شارپ استفاده کنید. کد تغییر یافته که از عملگر **as** با **String** (چون **Int32** یک نوع مقداری است) استفاده می کند را در زیر می بینید:

```
private static void CastingAGenericTypeVariable3<T>(T obj) {
    String s = obj as String;    // No error
}
```

تنظیم یک نوع جنریک به یک مقدار پیش فرض

تنظیم یک متغیر نوع جنریک به **null** غیر مجاز است مگر اینکه نوع جنریک به یک نوع ارجاعی محدود شده باشد.

```
private static void SettingAGenericTypeVariableToNull<T>() {
    T temp = null;    // CS0403 - Cannot convert null to type parameter 'T' because it could
                    // be a non-nullable value type. Consider using 'default(T)' instead
}
```

چون **T** بدون محدودیت است، آن می‌تواند یک نوع مقداری باشد و تنظیم یک متغیر از یک نوع مقداری به **null** غیرممکن است. اگر **T** به یک نوع ارجاعی محدود شده بود، تنظیم **temp** به **null** به خوبی کامپایل و اجرا می‌شد.

تیم سی‌شارپ مایکروسافت احساس کردند این کاربردی است که به برنامه‌نویسان قابلیت تنظیم یک متغیر به یک مقدار پیش فرض را بدهند. بنابراین کامپایلر سی‌شارپ به شما اجازه می‌دهد برای این کار از کلمه کلیدی **default** استفاده کنید:

```
private static void SettingAGenericTypeVariableToDefaultValue<T>() {
    T temp = default(T);    // OK
}
```

استفاده از کلمه کلیدی **default** به کامپایلر سی‌شارپ و کامپایلر JIT از CLR می‌گوید کدی تولید کند که اگر **T** یک نوع ارجاعی است آن را **null** کند و اگر یک نوع مقداری است، تمام بیت‌هایش را صفر کند.

مقایسه یک متغیر نوع جنریک با **null**

مقایسه یک متغیر نوع جنریک با **null** با استفاده از عملگر **==** یا **!=** مجاز است بدون توجه به اینکه آیا نوع جنریک محدود شده است یا نه:

```
private static void ComparingAGenericTypeVariablewithNull<T>(T obj) {
    if (obj == null) { /* Never executes for a value type */ }
}
```

چون **T** محدود نشده است، آن می‌تواند یک نوع ارجاعی یا یک نوع مقداری باشد. اگر **T** یک نوع مقداری باشد، **obj** هرگز **null** نمی‌تواند باشد. به صورت عادی، شما انتظار دارید که کامپایلر سی‌شارپ به این خاطر اعلام خطا کند. هرچند، کامپایلر سی‌شارپ اعلام خطا نمی‌کند، به جای آن، کد را به خوبی کامپایل می‌کند. وقتی که این متد با استفاده از یک آرگومان نوع که یک نوع مقداری است فراخوانی می‌شود، کامپایلر JIT می‌بیند که عبارت **if** هرگز برقرار نمی‌شود و کامپایلر JIT کد اصلی را برای کد درون آکولات‌های عبارت **if** و خود شرط آن تولید نمی‌کند. اگر من از عملگر **!=** استفاده کرده بودم، کامپایلر JIT کد را برای شرط **if** (چون همیشه برقرار است) تولید نمی‌کرد و کد درون آکولات‌های **if** را تولید می‌نمود.

ضمناً، اگر **T** به یک **struct** محدود شده بود، کامپایلر سی‌شارپ یک خطا اعلام می‌کرد چون شما نباید کدی بنویسید که یک متغیر نوع مقداری را با **null** مقایسه کند چون نتیجه همیشه یکسان است.

مقایسه‌ی دو متغیر نوع جنریک با همدیگر

مقایسه دو متغیر از نوع ارجاعی یکسان غیرمجاز است اگر پارامتر نوع جنریک به عنوان یک نوع ارجاعی شناخته نشود:

```
private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {
    if (o1 == o2) { }    // Error
}
```

در این مثال، **T** محدود نشده است و در حالیکه مقایسه دو متغیر نوع ارجاعی با همدیگر مجاز است، مقایسه دو نوع مقداری با همدیگر مجاز نیست مگر آنکه نوع مقداری عملگر **==** را سربارگذاری کرده باشد. اگر **T** به **class** محدود شده بود، این کد کامپایل می‌شد و عملگر **==** در صورتی که متغیرها به شی یکسانی اشاره می‌کردند، **true** برمی‌گرداند و در واقع هویت را بررسی می‌کرد و توجه داشته باشد که اگر **T** به یک نوع ارجاعی که متد عملگر **==** را سربارگذاری کرده، محدود شده باشد، کامپایلر هر جا عملگر **==** را می‌دیدید، فراخوانی به این متد را تولید می‌کرد.

بدیهی است که تمام این مبحث به استفاده از عملگر $=$ نیز اعمال می‌شود. وقتی شما کدی می‌نویسید که نوع‌های مقداری اصلی `Int32`، `Byte`، `Decimal`، `Single` و غیره را مقایسه کند، کامپایلر سی‌شارپ می‌داند که چگونه کد صحیحی را تولید نماید. هرچند، برای نوع‌های مقداری غیر اصلی، کامپایلر سی‌شارپ نمی‌داند چگونه کدی تولید کند که مقایسه را انجام دهد.

پس اگر `T` در متد `ComparingTwoGenericTypeVariables` به `struct` محدود شده بود، کامپایلر یک خطا اعلام می‌کرد. و شما مجاز نیستید که یک پارامتر نوع را به یک نوع مقداری خاص محدود کنید چون به صورت ضمنی مهر شده است و بنابراین هیچ نوع دیگری وجود ندارد که از نوع مقداری مشتق شده باشد. مجاز کردن این، متد جنریک را به یک نوع خاص محدود می‌کند و کامپایلر سی‌شارپ اجازه‌ی انجام چنین کاری را نمی‌دهد چون ساخت یک متد غیرجنریک کاراثر است.

استفاده از متغیرهای نوع جنریک به عنوان عملوند

در پایان، این باید اشاره شود که مسائل فراوانی در استفاده از عملگرها با عملوندهای نوع جنریک وجود دارد. در فصل ۵ من درباره‌ی سی‌شارپ و اینکه آن چگونه نوع‌های اصلی اش: `Byte`، `Int16`، `Int32`، `Int64`، `Decimal` و غیره را مدیریت می‌کند، صحبت کردم. به خصوص، اشاره کردم که سی‌شارپ می‌داند چگونه عملگرها (مثل `+`، `-`، `*` و `/`) را وقتی بر نوع‌های اصلی اعمال می‌شوند تفسیر کند. خوب، این عملگرها نمی‌توانند به متغیرهایی از یک نوع جنریک اعمال شوند چون کامپایلر، نوع را در زمان کامپایل نمی‌داند. این یعنی شما نمی‌توانید این عملگرها را با متغیرهایی از یک نوع جنریک استفاده کنید. پس نوشتن یک الگوریتم ریاضی که روی یک نوع عددی دلخواه کار کند غیر ممکن است. در اینجا مثالی از یک متد جنریک که من دوست دارم بنویسم را می‌بینید:

```
private static T Sum<T>(T num) where T : struct {
    T sum = default(T);
    for (T n = default(T); n < num; n++)
        sum += n;
    return sum;
}
```

من هر آنچه ممکن بود برای کامپایل شدن این کد انجام دادم. من `T` را به `struct` محدود کردم و از `default(T)` برای مقداردهی اولیه `sum` و `n` به `0` استفاده کردم. اما وقتی این کد را کامپایل می‌کنم، من سه خطای زیر را دریافت می‌کنم:

- **error CS0019: Operator '<' cannot be applied to operands of type 'T' and 'T'**
- **error CS0023: Operator '++' cannot be applied to operand of type 'T'**
- **error CS0019: Operator '+=' cannot be applied to operands of type 'T' and 'T'**

این محدودیت بزرگی در پشتیبانی جنریک CLR است و بسیاری از برنامه‌نویسان (مخصوصاً در زمینه‌های علمی، اقتصادی و ریاضی) با این محدودیت بسیار ناامید شده‌اند. بسیاری افراد با استفاده از رفلکشن (فصل ۲۳ "بارگذاری اسمبلی و رفلکشن" را نگاه کنید)، سربارگذاری عملگرها و ... به طریقی بر این محدودیت غلبه کرده‌اند. اما تمام این‌ها باعث ضربه شدید به کارایی یا خوانایی کد می‌شود. با امیدواری، این بحثی است که مایکروسافت در نسخه‌های آتی از CLR و کامپایلرها به آن خواهد پرداخت.

فصل ۱۳: رابط‌ها

بسیاری از برنامه‌نویسان، با مفهوم وراثت چندگانه آشنا هستند. قابلیت تعریف یک کلاس که از دو یا بیشتر کلاس پایه، مشتق شده باشد. برای نمونه، کلاسی به نام **TransmitData** را تصور کنید که کارش ارسال داده است و کلاس دیگری به نام **ReceiveData** که کارش دریافت داده است. حال تصور کنید شما می‌خواهید کلاسی به نام **SocketPort** بسازید که کارش ارسال و دریافت داده است. برای انجام این کار، شما می‌خواهید که **SocketPort** از هر دوی **ReceiveData** و **TransmitData** مشتق شود.

برخی زبان‌های برنامه‌نویسی اجازه‌ی وراثت چندگانه را می‌دهند و این را ممکن می‌سازند که کلاس **SocketPort** از دو کلاس پایه، **TransmitData** و **ReceiveData** مشتق شود. هرچند، CLR - و بنابراین تمام زبان‌های برنامه‌نویسی مدیریت‌شده - وراثت چندگانه را پشتیبانی نمی‌کنند. به جای ارائه نکردن هیچ گونه وراثت چندگانه، CLR وراثت چندگانه را در مقایسه کوچکتر، از طریق رابط‌ها **interfaces** ارائه می‌کند. این فصل بحث می‌کند چگونه رابط‌ها را تعریف و استفاده کنید و راهنمایی‌هایی برای تصمیم‌گیری درباره‌ی استفاده از یک رابط به جای یک کلاس پایه را نیز بیان می‌کند.

وراثت کلاس و رابط

در دات‌نت فریم‌ورک میکروسافت، کلاسی به نام **System.Object** وجود دارد که چهار متد نمونه عمومی تعریف می‌کند: **Equals**، **Tostring**، **GetType** و **GetHashCode**. این کلاس، ریشه و کلاس پایه‌ی نهایی تمام دیگر کلاس‌هاست - تمام کلاس‌ها چهار متد نمونه‌ی **Object** را به ارث می‌برند. این همچنین یعنی کدی که بر روی یک نمونه از کلاس **Object** عمل می‌کند در حقیقت روی یک نمونه از هر کلاسی نیز عمل می‌کند. چون فردی در میکروسافت متدهای **Object** را پیاده‌سازی کرده است، هر کلاسی که از **Object** مشتق می‌شود، در حقیقت موارد زیر را به ارث می‌برد:

- **امضای متدها** این به کد اجازه می‌دهد که فکر کند بر روی یک نمونه از کلاس **Object** عمل می‌کند، در حالیکه در حقیقت می‌تواند بر روی یک نمونه از کلاسی دیگر عمل کند.
- **پیاده‌سازی این متدها** این به برنامه‌نویسی که یک کلاس مشتق شده از **Object** را تعریف می‌کند اجازه می‌دهد دیگر مجبور نباشد به صورت دستی متدهای **Object** را پیاده‌سازی کند.

در CLR، یک کلاس همیشه از یک و فقط یک کلاس مشتق می‌شود (که سرانجام باید از **Object** مشتق گردد). این کلاس پایه، تعدادی امضای متدها و پیاده‌سازی این متدها را فراهم می‌کند. و یک چیز خوب پیرامون تعریف یک کلاس جدید اینست که آن می‌تواند کلاس پایه برای کلاس دیگری که در آینده توسط برنامه‌نویس دیگری تعریف می‌شود باشد - تمام امضای متدها و پیاده‌سازی آن‌ها توسط کلاس مشتق شده‌ی جدید به ارث برده می‌شود.

CLR به برنامه‌نویسان اجازه می‌دهد یک رابط **interface** تعریف کنند که در حقیقت روشی برای نامگذاری یک مجموعه از امضای متدهاست. این متدها اصلاً با پیاده‌سازی همراه نیستند. یک کلاس، یک رابط را با تعیین نام رابط به ارث می‌برد و قبل از آنکه CLR تعریف نوع را معتبر در نظر بگیرد، کلاس باید صریحاً پیاده‌سازی‌های متدها رابط را فراهم کند. البته، پیاده‌سازی متدهای یک رابط شاید خسته کننده باشد که بهمین خاطر است که من به وراثت رابط به عنوان روشی با مقیاس کوچکتر جهت دستیابی به وراثت چندگانه اشاره کردم. کامپایلر سی‌شارپ و CLR به یک کلاس اجازه می‌دهند چند رابط را به ارث ببرد و البته، کلاس باید پیاده‌سازی را برای تمام متدهای به ارث برده شده‌ی رابط، فراهم کند.

یکی از ویژگی‌های عالی وراثت اینست که اجازه می‌دهد نمونه‌هایی از یک نوع مشتق شده در تمام زمینه‌هایی که یک نمونه از نوع پایه مورد انتظار است، جایگزین شوند. به طریق مشابه، وراثت رابط به نمونه‌هایی از یک نوع که رابط را پیاده‌سازی کرده، اجازه می‌دهد در تمام زمینه‌هایی که یک نوع رابط نام برده، مورد انتظار است، جایگزین شود. حال به چگونگی تعریف رابط‌ها نگاه می‌کنیم تا بچثمان را مستحکم تر کنیم.

تعریف یک رابط

همانگونه که در بخش قبلی اشاره کردم، یک رابط یک مجموعه‌ی نامگذاری شده از امضای متدهاست. توجه کنید که رابط‌ها می‌توانند رویدادها، ویژگی‌های بدون پارامتر و ویژگی‌های پارامتردار (ایندکسرها در سی‌شارپ) را تعریف کنند چون تمام این‌ها فقط نحو ساده شده‌ای از متدها هستند. همانطور که در فصل - های قبلی نشان داده شد، یک رابط نمی‌تواند هیچ متد سازنده‌ای تعریف کند. به علاوه، یک رابط مجاز نیست هیچ فیلد نمونه‌ای تعریف کند.

اگرچه CLR به یک رابط اجازه می‌دهد متدهای استاتیک، فیلدهای استاتیک، ثابت‌ها و سازنده‌های استاتیک را تعریف کند، یک رابط منطبق بر زیرساختار مشترک زبان (CLI) نباید هیچ عضو استاتیکی تعریف کند چون برخی زبان‌های برنامه‌نویسی قادر به تعریف یا دسترسی به آن‌ها نیستند. در حقیقت، سی-شارپ مانع می‌شود که یک رابط هر کدام از این اعضای استاتیک را تعریف کند.

در سی‌شارپ، شما از کلمه کلیدی `interface` برای تعریف یک رابط استفاده می‌کنید، به آن یک نام می‌دهید و امضای متدهای نمونه آن را تنظیم می‌کنید. تعریف چند رابط که در کتابخانه کلاس فریمورک (FCL) تعریف شده اند، را در زیر می‌بینیم:

```
public interface IDisposable {
    void Dispose();
}

public interface IEnumerable {
    IEnumerator GetEnumerator();
}

public interface IEnumerable<out T> : IEnumerable {
    new IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    void Add(T item);
    void Clear();
    Boolean Contains(T item);
    void CopyTo(T[] array, Int32 arrayIndex);
    Boolean Remove(T item);
    Int32 Count { get; } // Read-only property
    Boolean IsReadOnly { get; } // Read-only property
}
```

برای CLR، تعریف یک رابط دقیقاً شبیه یک نوع است. یعنی، CLR یک ساختار داده‌ای درونی برای شی نوع رابط تعریف می‌کند و برای دسترسی به خصوصیات نوع رابط می‌توان از رفلکشن استفاده نمود. همانند نوع‌ها، یک رابط می‌تواند در میدان فایل یا تودرتو درون نوعی دیگر تعریف شود. هنگام تعریف نوع رابط، شما هر پدیداری/دسترس پذیری را که بخواهید می‌توانید تعیین کنید (**internal**, **protected**, **public** و غیره).

طبق قرارداد، نام نوع‌های رابط با یک پیشوند حرف **I** بزرگ همراه است، تا شناسایی آن به عنوان یک رابط در کد آسانتر باشد. CLR از رابط‌های جنریک (می‌توانید در مثال‌های قبلی ببینید) و متدهای جنریک در یک رابط پشتیبانی می‌کند.

من برخی از ویژگی‌های ارائه شده توسط رابط‌های جنریک را در این فصل خواهیم گفت و در فصل ۱۲ "جنریک‌ها" بحث جنریک‌ها را پوشش دادم. تعریف یک رابط می‌تواند دیگر رابط‌ها را "به ارث ببرد". هر چند من عبارت "به ارث بردن" را ضعیف تر به کار بردم چون وراثت رابط دقیقاً شبیه وراثت کلاس، کار نمی‌کند. من ترجیح می‌دهم به وراثت رابط به عنوان شامل شدن قرارداد دیگر رابط‌ها، فکر کنم. برای نمونه، تعریف رابط `ICollection<T>` قراردادهای رابط‌های `IEnumerable` و `IEnumerable<T>` را شامل می‌شود. این یعنی:

- هر کلاسی که رابط `ICollection<T>` را به ارث ببرد، باید تمام متدهای تعریف شده توسط رابط‌های `ICollection<T>`، `IEnumerable` و `IEnumerable<T>` را پیاده‌سازی کند.
- هر کدی که یک شی، که نوعش رابط `ICollection<T>` را پیاده‌سازی می‌کند را انتظار دارد، می‌تواند فرض کند که نوع شی، متدهای رابط‌های `IEnumerable` و `IEnumerable<T>` را نیز پیاده‌سازی می‌کند.

به ارث بردن یک رابط

در این بخش، من نشان خواهم داد چگونه یک نوع تعریف کنید که یک رابط را پیاده‌سازی کند و سپس نشان می‌دهم چگونه یک نمونه از این نوع بسازید و از شی برای فراخوانی متدهای رابط استفاده کنید. سی‌شارپ این کار را بسیار ساده کرده است، اما آنچه پشت پرده رخ می‌دهد کمی پیچیده است. من در ادامه فصل توضیح می‌دهم در پشت پرده چه رخ می‌دهد.

رابط `System.IComparable<T>` اینگونه (در `mscorlib.dll`) تعریف شده است:

```
public interface IComparable<in T> {
    Int32 CompareTo(T other);
}
```

کد زیر نشان می‌دهد چگونه یک نوع که این رابط را پیاده‌سازی می‌کند تعریف کنید و همچنین کدی را نشان می‌دهد که دو شی `Point` را با هم مقایسه می‌کند:

```
using System;
```

```
// Point is derived from System.Object and implements IComparable<T> for Point.
```

```
public sealed class Point : IComparable<Point> {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    // This method implements IComparable<T>.CompareTo() for Point
    public Int32 CompareTo(Point other) {
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
            - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}
```

```
public static class Program {
    public static void Main() {
        Point[] points = new Point[] {
            new Point(3, 3),
            new Point(1, 2),
        };

        // Here is a call to Point's IComparable<T> CompareTo method
        if (points[0].CompareTo(points[1]) > 0) {
            Point tempPoint = points[0];
            points[0] = points[1];
            points[1] = tempPoint;
        }
        Console.WriteLine("Points from closest to (0, 0) to farthest:");
    }
}
```

```

        foreach (Point p in points)
            Console.WriteLine(p);
    }
}

```

کامپایلر سی شارپ نیاز دارد که یک متد که یک رابط را پیاده‌سازی می‌کند **public** علامت زده شود. CLR نیاز دارد که متدهای رابط **virtual** علامت زده شوند. اگر شما صریحاً در سورس کدتان، متد را مجازی علامت نزنید، کامپایلر متد را مجازی علامت می‌زند (و آن را مهر نشده رها می‌کند). این به یک کلاس مشتق شده اجازه می‌دهد متد رابط را بازنویسی کند.

اگر یک متد رابط مهر شده باشد، یک کلاس پایه نمی‌تواند متد را بازنویسی کند، هر چند یک کلاس پایه می‌تواند همان رابط را مجدداً به ارث ببرد و پیاده‌سازی خودش را برای متدهای رابط فراهم کند. هنگام فرخوانی متد یک رابط روی یک شی، پیاده‌سازی‌ای که همراه با نوع شی است فراخوانی می‌گردد. مثالی که این مطلب را نشان می‌دهد در اینجا آمده است:

```
using System;
```

```

public static class Program {
    public static void Main() {
        /***** First Example *****/
        Base b = new Base();

        // Calls Dispose by using b's type: "Base's Dispose"
        b.Dispose();

        // Calls Dispose by using b's object's type: "Base's Dispose"
        ((IDisposable)b).Dispose();

        /***** Second Example *****/
        Derived d = new Derived();

        // Calls Dispose by using d's type: "Derived's Dispose"
        d.Dispose();

        // Calls Dispose by using d's object's type: "Derived's Dispose"
        ((IDisposable)d).Dispose();

        /***** Third Example *****/
        b = new Derived();

        // Calls Dispose by using b's type: "Base's Dispose"
        b.Dispose();

        // Calls Dispose by using b's object's type: "Derived's Dispose"
        ((IDisposable)b).Dispose();
    }
}

```

```
// This class is derived from Object and it implements IDisposable
```

```

internal class Base : IDisposable {
    // This method is implicitly sealed and cannot be overridden
    public void Dispose() {

```

```

        Console.WriteLine("Base's Dispose");
    }
}

// This class is derived from Base and it re-implements IDisposable
internal class Derived : Base, IDisposable {
    // This method cannot override Base's Dispose. 'new' is used to indicate
    // that this method re-implements IDisposable's Dispose method
    new public void Dispose() {
        Console.WriteLine("Derived's Dispose");

        // NOTE: The next line shows how to call a base class's implementation (if desired)
        // base.Dispose();
    }
}

```

اطلاعات بیشتر درباره فراخوانی متدهای رابط

نوع **System.String** امضاء و پیاده‌سازی متدهای **System.Object** را به ارث می‌برد. به علاوه، نوع **String**، چندین رابط را پیاده‌سازی می‌کند: **IEnumerable<Char>**، **IComparable<String>**، **IEnumerable**، **IConvertible**، **ICloneable**، **IComparable** و **IComparable<String>**. این یعنی نوع **String** نیاز ندارد متدهای نوع پایه اش، **Object**، را پیاده‌سازی (یا بازنویسی) کند. هر چند، نوع **String** باید متدهای تعریف شده در تمام این رابط‌ها را پیاده‌سازی کند.

CLR به شما اجازه می‌دهد فیلد، پارامتر یا متغیرهای محلی تعریف کنید که از نوع یک رابط باشند. استفاده از یک متغیر از یک نوع رابط به شما اجازه می‌دهد متدهای تعریف شده توسط آن رابط را فراخوانی کنید. به علاوه، CLR به شما اجازه می‌دهد که متدهای تعریف شده توسط **Object** را نیز فراخوانی کنید چون تمام کلاس‌ها، متدهای **Object** را به ارث می‌برند. کد زیر این مطلب را نشان می‌دهد:

```

// The s variable refers to a String object.
String s = "Jeffrey";
// Using s, I can call any method defined in
// String, Object, IComparable, ICloneable, IConvertible, IEnumerable, etc.

// The cloneable variable refers to the same String object
ICloneable cloneable = s;
// Using cloneable, I can call any method declared by the
// ICloneable interface (or any method defined by Object) only.

// The comparable variable refers to the same String object
IComparable comparable = s;
// Using comparable, I can call any method declared by the
// IComparable interface (or any method defined by Object) only.

// The enumerable variable refers to the same String object
// At run time, you can cast a variable from one interface to another as
// long as the object's type implements both interfaces.
IEnumerable enumerable = (IEnumerable) comparable;
// Using enumerable, I can call any method declared by the
// IEnumerable interface (or any method defined by Object) only.

```

در این کد تمام متغیرها به همان شیء **String**، **Jeffrey** اشاره دارند که در هیپ مدیریت شده است و بنابراین، هر متدی را که با استفاده از این متغیرها فراخوانی کنیم بر شیء **String**، **Jeffrey**، اثر می‌گذارد. هر چند، نوع متغیر، عملی که می‌توانیم روی شیء انجام دهیم را مشخص می‌کند. متغیر **s** از نوع **String** است و بنابراین، من می‌توانم از **s** برای فراخوانی هر عضو تعریف شده توسط نوع **String** (مثل ویژگی **Length**) استفاده کنم. من همچنین می‌توانم از متغیری برای فراخوانی هر متد به ارث برده شده از **Object** (مثل **GetType**) استفاده کنم.

متغیر **cloneable** از نوع رابط **ICloneable** است و بنابراین، با استفاده از متغیر **cloneable** می‌توانم متد **Clone** که توسط این رابط تعریف شده است را فراخوانی کنم. به علاوه، من می‌توانم هر متد تعریف شده توسط **Object** (مثل **GetType**) را فراخوانی کنم چون **CLR** می‌داند تمام نوع‌ها از **Object** مشتق می‌شوند. هر چند، با استفاده از متغیر **cloneable** من نمی‌توانم متدهای عمومی تعریف شده توسط خود **String** یا متدهای تعریف شده توسط هر رابط دیگری که **String** پیاده‌سازی می‌کند را فراخوانی کنم. به طریق مشابه، با استفاده از متغیر **comparable**، من می‌توانم **CompareTo** یا هر متد تعریف شده توسط **Object** را تعریف کنم اما هیچ متد دیگری با این متغیر قابل فراخوانی نیست.

مهم همانند یک نوع ارجاعی، یک نوع مقداری می‌تواند صفر یا بیشتر رابط را پیاده‌سازی کند. هر چند، وقتی شما یک نمونه از یک نوع مقداری را به قالب یک نوع رابط درمی‌آورید (تبدیل می‌کنید)، نمونه نوع مقداری باید بسته بندی شود. این بدین خاطر است که یک متغیر رابط یک اشاره‌گر است که باید به یک شیء در هیپ اشاره کند تا **CLR** بتواند اشاره‌گر شیء نوع متعلق به شیء را برای تعیین نوع دقیق شیء، بررسی کند. سپس، هنگام فراخوانی یک متد رابط با یک نوع مقداری بسته بندی شده، **CLR** اشاره‌گر شیء نوع متعلق به شیء را پیگیری می‌کند تا جدول متد شیء نوع را برای فراخوانی متد صحیح، بیابد.

پیاده‌سازی صریح و ضمنی متد رابط (در پشت صحنه چه رخ می‌دهد)

وقتی یک نوع در **CLR** بارگذاری می‌شود، یک جدول متد برای نوع ایجاد و مقداردهی اولیه می‌شود (همانگونه که در فصل ۱ "مدل اجرایی CLR" بحث شد). این جدول متد حاوی یک ورودی برای هر متد معرفی شده توسط نوع و همچنین ورودیهایی برای هر متد مجازی به ارث برده شده توسط نوع می‌باشد. متدهای مجازی به ارث برده شده شامل متدهای تعریف شده توسط نوع‌های پایه در سلسله مراتب وراثت و همچنین هر متد تعریف شده توسط نوع‌های رابط است. پس اگر شما یک نوع ساده مثل این تعریف کنید:

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("Dispose"); }
}
```

جدول متد نوع حاوی ورودیهایی برای موارد زیر است:

- تمام متدهای نمونه مجازی تعریف شده توسط **Object**، کلاس پایه‌ای که به صورت ضمنی از آن ارث برده می‌شود.
- تمام متدهای تعریف شده توسط **IDisposable**، رابط به ارث برده شده. در این مثال، تنها یک متد، **Dispose**، وجود دارد چون رابط **IDisposable** تنها یک متد تعریف می‌کند.
- متد جدید، **Dispose**، که توسط **SimpleType** تعریف شده است.

برای آنکه کار برنامه‌نویس ساده تر شود، کامپایلر سی‌شارپ فرض می‌کند که متد **Dispose** تعریف شده توسط **SimpleType**، پیاده‌سازی متد **Dispose** از **IDisposable** است. کامپایلر سی‌شارپ این فرض را برای این کار لحاظ می‌کند که متد، **public** است و امضای متد رابط و متد جدید معرفی شده، یکسان است. یعنی، متدها پارامترها و نوع برگشتی یکسانی دارند. ضمناً، اگر متد جدید **Dispose** به عنوان **virtual** علامت زده شده بود، کامپایلر سی‌شارپ هنوز هم این متد را مطابق با متد رابط در نظر می‌گرفت.

وقتی کامپایلر سی‌شارپ یک متد جدید را با یک متد رابط تطابق می‌دهد، متادیتایی تولید می‌کند که بیان می‌کند هر دو ورودی در جدول متد **SimpleType** باید به پیاده‌سازی یکسانی اشاره کنند. برای واضح تر کردن مطلب، کد زیر نشان می‌دهد که چگونه متد عمومی **Dispose** متعلق به کلاس را فراخوانی کرده و همچنین چگونه پیاده‌سازی کلاس برای متد **Dispose** از **IDisposable** را فراخوانی کنید:

```
public sealed class Program {
    public static void Main() {
        SimpleType st = new SimpleType();
    }
}
```

```
// This calls the public Dispose method implementation
st.Dispose();

// This calls IDisposable's Dispose method implementation
IDisposable d = st;
d.Dispose();
}
}
```

در اولین فراخوانی به **Dispose**، متد **Dispose** تعریف شده توسط **SimpleType** فراخوانی می شود. سپس من یک متغیر تعریف می کنم، **d**، که از نوع رابط **IDisposable** است. من متغیر **d** را با اشاره به شی **SimpleType** مقداردهی اولیه می کنم. حال من، وقتی **d.Dispose()** را فراخوانی می کنم، من دارم متد **Dispose** از رابط **IDisposable** را فراخوانی می کنم. چون سی شارپ نیاز دارد که متد عمومی **Dispose**، پیاده سازی برای متد **Dispose** از **IDisposable** نیز باشد، همان کد اجرا خواهد شد و در این مثال، شما تفاوت قابل مشاهده ای نمی بینید. خروجی اینگونه است:

```
Dispose
Dispose
```

حال بگذارید **SimpleType** فوق را به گونه ای تغییر دهیم که تفاوت قابل مشاهده باشد:

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("public Dispose"); }
    void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

بدون تغییر متد **Main** که قبلا نشان دادم، اگر فقط برنامه را مجدداً کامپایل و اجرا کنیم، خروجی این خواهد بود:

```
public Dispose
IDisposable Dispose
```

در سی شارپ، وقتی شما نام متد را با پیشوند نام رابطی که متد را تعریف کرده، همراه می کنید (در این مثال **IDisposable.Dispose**)، شما یک پیاده سازی صریح متد رابط (**EIMI**) **explicit interface method implementation** می سازید. دقت کنید وقتی شما یک متد رابط صریح در سی شارپ تعریف می کنید، اجازه ندارید هیچ دسترس پذیری (مثل **public** یا **private**) برای آن تعیین کنید. هرچند، وقتی کامپایلر، متادیتا را برای متد تولید می کند، دسترس پذیری آن را **private** می کند و مانع از این می شود که هر کدی به سادگی با استفاده از یک نمونه از کلاس، متد رابط را فراخوانی کند. تنها راه فراخوانی متد رابط از طریق یک متغیر از نوع رابط است.

همچنین دقت کنید که یک متد **EIMI** نمی تواند **virtual** علامت زده شود و بنابراین غیر قابل بازنویسی است. این بدین خاطر است که متد **EIMI** در واقع بخشی از مدل شی متعلق به نوع نیست، آن راهی برای ضمیمه کردن یک رابط (مجموعه ای از رفتارها یا متدها) به یک نوع بدون ظاهر کردن رفتارها/متدهاست.

اگر تمام این ها برای شما ناچور به نظر می رسد، شما دارید آن را درست درک می کنید. تمام این ها کمی ناچور است. در ادامه این فصل، دلایل معتبری بر استفاده از **EIMI** نشان خواهیم داد.

رابط های جنریک

پشتیبانی سی شارپ و CLR از رابط های جنریک مزایای بسیار عالی برای برنامه نویسان به همراه دارد. در این بخش، من می خواهم فواید استفاده از رابط های جنریک را بحث کنم.

نخست اینکه رابط های جنریک امنیت نوع بسیار عالی در زمان کامپایل ارائه می کنند. برخی رابط ها (مثل رابط غیر جنریک **IComparable**) متدهایی تعریف می کنند که دارای پارامترها یا نوع برگشتی **Object** هستند. وقتی کدی این متدهای رابط را فراخوانی می کند، یک اشاره گر به یک نمونه از هر نوع را می توان ارسال کرد. این معمولاً مطلوب نیست. کد زیر این را نشان می دهد:

```
private void SomeMethod1() {
    Int32 x = 1, y = 2;
    IComparable c = x;
```



```
// CompareTo expects an Object; passing y (an Int32) is OK
c.CompareTo(y); // y is boxed here

// CompareTo expects an Object; passing "2" (a String) compiles
// but an ArgumentException is thrown at runtime
c.CompareTo("2");
}
```

بدیهی است که ترجیح داده می‌شود که متد رابط، نوع قوی باشد و این علتی است که FCL شامل یک رابط جنریک **IComparable<T>** است. نسخه‌ی جدید کد که با رابط جنریک کار می‌کند:

```
private void SomeMethod2() {
    Int32 x = 1, y = 2;
    IComparable<Int32> c = x;

    // CompareTo expects an Int32; passing y (an Int32) is OK
    c.CompareTo(y); // y is not boxed here

    // CompareTo expects an Int32; passing "2" (a String) results
    // in a compiler error indicating that String cannot be cast to an Int32
    c.CompareTo("2"); // Error
}
```

مزیت دوم رابط‌های جنریک این است که بسته‌بندی بسیار کمتری در هنگام کار با نوع‌های مقداری رخ می‌دهد. دقت کنید در **SomeMethod1** متد **CompareTo** از رابط غیرجنریک **IComparable** انتظار یک **Object** دارد، **y** (یک نوع مقداری **Int32**) بدان ارسال می‌شود که باعث بسته‌بندی **y** می‌گردد. ولی، در **SomeMethod2**، متد **CompareTo** از رابط جنریک **IComparable<in T>** انتظار یک **Int32** را دارد، **y** با مقدار، بدان ارسال می‌شود و هیچ بسته‌بندی‌ای نیاز نخواهد بود.

نکته FCL نسخه‌های غیرجنریک و جنریک از رابط‌های **IComparable**، **ICollection**، **IList** و **IDictionary** و برخی دیگر را تعریف می‌کند. اگر شما یک نوع تعریف می‌کنید و می‌خواهید یکی از این رابط‌ها را پیاده‌سازی کنید، شما باید نوعاً نسخه‌ی جنریک این رابط‌ها را پیاده‌سازی کنید. نسخه‌های غیرجنریک برای سازگاری با نسخه‌های قبلی در FCL هستند تا با کد نوشته شده در قبل از پشتیبانی دات‌نت فریم‌ورک از جنریک‌ها، کار کنند. نسخه‌های غیرجنریک همچنین روشی برای دستکاری داده به گونه‌ای عمومی تر و کمتر نوع-امن، فراهم می‌کنند. برخی رابط‌های جنریک، نسخه‌های غیرجنریک را به ارث می‌برند، پس کلاس شما مجبور خواهد بود هر دو نسخه‌ی جنریک و غیرجنریک رابط‌ها را پیاده‌سازی کند. برای نمونه، رابط جنریک **IEnumerable<out T>**، رابط غیرجنریک **IEnumerable** را به ارث می‌برد. پس اگر کلاس شما **IEnumerable<out T>** را پیاده‌سازی می‌کند، کلاس شما باید **IEnumerable** را نیز پیاده‌سازی کند. گاهی هنگام کار با دیگر کدها، شاید شما بخواهید یک رابط غیرجنریک را به این خاطر که یک نسخه‌ی جنریک از رابط وجود ندارد، پیاده‌سازی کنید. در این حالت اگر هر یک از متدهای رابط، **Object** بگیرند یا برگردانند، شما امنیت نوع در زمان کامپایل را از دست می‌دهید و شما با نوع‌های مقداری، بسته‌بندی را خواهید داشت. شما می‌توانید این وضعیت را تا حدی با تکنیکی که در بخش "بهبود امنیت نوع در زمان کامپایل در پیاده‌سازی‌های صریح متد رابط" در انتهای فصل می‌گوییم، بهبود بخشید.

فایده‌ی سوم رابط‌های جنریک اینست که یک کلاس می‌تواند همان رابط را چندین بار پیاده‌سازی کند تا زمانی که پارامترهای نوع متفاوتی را تعیین کند. کد زیر مثالی را نشان می‌دهد که این مزیت چقدر در آن سودمند است:

```
using System;
```

```
// This class implements the generic IComparable<T> interface twice
public sealed class Number: IComparable<Int32>, IComparable<String> {
    private Int32 m_val = 5;
```

```

// This method implements IComparable<Int32>'s CompareTo
public Int32 CompareTo(Int32 n) {
    return m_val.CompareTo(n);
}

// This method implements IComparable<String>'s CompareTo
public Int32 CompareTo(String s) {
    return m_val.CompareTo(Int32.Parse(s));
}
}

public static class Program {
    public static void Main() {
        Number n = new Number();

        // Here, I compare the value in n with an Int32 (5)
        IComparable<Int32> cInt32 = n;
        Int32 result = cInt32.CompareTo(5);

        // Here, I compare the value in n with a String ("5")
        IComparable<String> cString = n;
        result = cString.CompareTo("5");
    }
}

```

پارامترهای نوع جنریک یک رابط نیز می‌توانند **contravariant** یا **covariant** علامت زده شوند که اجازه‌ی انعطاف‌پذیری بیشتری برای استفاده از رابط‌های جنریک را می‌دهد. برای اطلاعات بیشتر درباره‌ی **contravariance** و **covariance**، بخش "آرگومان‌ها نوع جنریک **Contravariant** و **Covariant** از نماینده‌ها و رابط‌ها" در فصل ۱۲ را ببینید.

جنریک‌ها و محدودیت‌های رابط

در بخش قبلی، مزایای استفاده از رابط‌های جنریک را بحث کردیم. در این بخش، من مزایای محدود کردن پارامترهای نوع جنریک به رابط‌ها را بحث می‌کنم.

اولین فایده اینست که شما می‌توانید یک پارامتر نوع جنریک را به چندین رابط محدود کنید. وقتی این کار را می‌کنید، نوع پارامتری که ارسال می‌کنید باید تمام این محدودیت‌های رابط را پیاده‌سازی کند. نمونه‌ای در اینجا آمده است:

```

public static class SomeType {
    private static void Test() {
        Int32 x = 5;
        Guid g = new Guid();

        // This call to M compiles fine because
        // Int32 implements IComparable AND IConvertible
        M(x);

        // This call to M causes a compiler error because
        // Guid implements IComparable but it does not implement IConvertible
        M(g);
    }
}

```

```

}

// M's type parameter, T, is constrained to work only with types that
// implement both the IComparable AND IConvertible interfaces
private static Int32 M<T>(T t) where T : IComparable, IConvertible {
    ...
}
}

```

این واقعا عالیست، وقتی شما پارامترهای یک متد را تعریف می‌کنید، نوع هر پارامتر بیان می‌کند که آرگومان ارسالی باید از نوع پارامتر یا یک نوع مشتق شده از آن باشد، اگر نوع پارامتر یک رابط است، این بیان می‌کند که آرگومان می‌تواند از هر نوعی باشد تا زمانی که کلاس، رابط را پیاده‌سازی می‌کند. استفاده از محدودیت‌های رابط‌های چندگانه اجازه می‌دهد متد بیان کند که آرگومان ارسالی باید چندین رابط را پیاده‌سازی کند.

در حقیقت، اگر ما **T** را به یک کلاس و دو رابط محدود کنیم، ما می‌گوییم که نوع آرگومان ارسالی باید از نوع کلاس پایه تعیین شده (یا مشتق شده از آن) باشد و آن همچنین باید دو رابط را پیاده‌سازی کند. این انعطاف پذیری به متد اجازه می‌دهد که واقعا دیکته کند فراخوانی کننده‌ها چه چیزهایی می‌توانند ارسال کنند و اگر فراخوانی کننده‌ها این سه محدودیت را نداشته باشند، کامپایلر اعلام خطا می‌کند.

فایده دوم از محدودیت‌های رابط، کاهش بسته‌بندی هنگام ارسال نمونه‌هایی از نوع مقداری است. در تکه کد قبلی، **x** (یک نمونه از یک **Int32** که یک نوع مقداری است) به متد **M** ارسال شد. وقتی **x** به **M** ارسال شد هیچ بسته‌بندی رخ نداد. اگر کد درون **M**، **t.CompareTo(...)** را فراخوانی کند، هنوز هم هیچ بسته‌بندی برای فراخوانی رخ نمی‌دهد (بسته‌بندی برای آرگومان‌های ارسالی به **CompareTo** ممکن است رخ بدهد). در سوی دیگر، اگر **M** اینگونه تعریف شده بود:

```

private static Int32 M(IComparable t) {
    ...
}

```

آنگاه برای ارسال **x** به **M**، **x** می‌بایست بسته‌بندی می‌شد.

برای محدودیت‌های رابط، کامپایلر سی‌شارپ دستورات زبان میانی (IL) خاصی تولید می‌کند که منجر به فراخوانی متد رابط، مستقیما روی نوع مقداری بدون بسته‌بندی می‌شود. به جز استفاده از محدودیت‌های رابط، راه دیگری که کامپایلر سی‌شارپ این دستورات IL را تولید کند وجود ندارد و بنابراین، فراخوانی یک متد رابط روی یک نوع مقداری همیشه باعث بسته‌بندی می‌گردد.

پیاده‌سازی چندین رابط که نام و امضای متد یکسانی دارند

ندرتا، شما می‌خواهید یک نوع تعریف کنید که چندین رابط که متدهایی با نام و امضای یکسان تعریف می‌کنند را پیاده‌سازی کند. برای مثال، فرض کنید دو رابط طبق زیر تعریف شده اند:

```

public interface IWindow {
    Object GetMenu();
}

```

```

public interface IRestaurant {
    Object GetMenu();
}

```

بگوییم که شما می‌خواهید یک نوع تعریف کنید که هر دوی این رابط‌ها را پیاده‌سازی کند شما مجبورید اعضای نوع را با استفاده از پیاده‌سازی صریح متد رابط طبق زیر پیاده کنید:

```

// This type is derived from System.Object and
// implements the IWindow and IRestaurant interfaces.
public sealed class MarioPizzeria : IWindow, IRestaurant {

    // This is the implementation for IWindow's GetMenu method.

```

```

Object IWindow.GetMenu() { ... }

// This is the implementation for IRestaurant's GetMenu method.
Object IRestaurant.GetMenu() { ... }

// This (optional method) is a GetMenu method that has nothing
// to do with an interface.
public Object GetMenu() { ... }
}

```

چون این نوع باید چندین متد جداگانه‌ی **GetMenu** را پیاده‌سازی کند، شما نیاز دارید به کامپایلر سی‌شارپ بگویید کدام متد **GetMenu** حاوی پیاده‌سازی برای یک رابط خاص است.

کدی که از یک شی **MarioPizzeria** استفاده می‌کند، باید در قالب یک رابط خاص قرار بگیرد (تبدیل شود) تا بتواند متد مورد نظر را فراخوانی کند. کد زیر این را نشان می‌دهد:

```

MarioPizzeria mp = new MarioPizzeria();

// This line calls MarioPizzeria's public GetMenu method
mp.GetMenu();

// These lines call MarioPizzeria's IWindow.GetMenu method
IWindow window = mp;
window.GetMenu();

// These lines call MarioPizzeria's IRestaurant.GetMenu method
IRestaurant restaurant = mp;
restaurant.GetMenu();

```

بهبود امنیت نوع در زمان کامپایل در پیاده‌سازی های صریح متد رابط

رابط‌ها عالی هستند چون یک روش استاندارد برای نوع‌هایی که می‌خواهند با همدیگر ارتباط برقرار کنند، تعریف می‌کنند. قبلاً، من درباره‌ی رابط‌های جنریک و اینکه آن‌ها چگونه امنیت نوع در زمان کامپایل را بهبود بخشیده و بسته‌بندی را کاهش می‌دهند، صحبت کردم. متأسفانه، مواقع زیادی هست که شما نیاز به پیاده‌سازی یک رابط غیرجنریک فقط به این دلیل دارید که یک نسخه‌ی جنریک از آن وجود ندارد. اگر هر یک از متدهای رابط، پارامترهایی از نوع **System.Object** بپذیرند یا یک مقدار که نوعش **System.Object** است، برگردانند، شما امنیت نوع در زمان کامپایل را از دست می‌دهید و بسته‌بندی را نیز خواهید داشت. در این بخش، من نشان خواهم داد چگونه از **EIMI** برای بهبود این وضعیت استفاده کنید:

به رابط بسیار رایج **IComparable** نگاه کنید:

```

public interface IComparable {
    Int32 CompareTo(Object other);
}

این رابط یک متد که یک پارامتر از نوع System.Object دریافت می‌کند را تعریف می‌نماید. اگر من نوع خودم که این رابط را پیاده‌سازی می‌کند تعریف کنم، تعریف نوع شبیه به این است:

internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }
    public Int32 CompareTo(Object other) {
        return(m_x - ((SomeValueType) other).m_x);
    }
}

```

با استفاده از **SomeValueType**، اکنون من می‌توانم کد زیر را بنویسم:

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v);    // Undesired boxing
    n = v.CompareTo(o);        // InvalidCastException
}
```

این کد دو ویژگی دارد که ایده آل نیست:

- **بسته‌بندی ناخواسته** وقتی **v** به عنوان یک آرگومان به متد **CompareTo** ارسال می‌شود، باید بسته‌بندی گردد چون **CompareTo** انتظار یک **Object** را دارد.
- **نبود امنیت نوع** این کد کامپایل می‌شود، اما وقتی که سعی دارد **o** را به **SomeValueType** تبدیل کند، یک **InvalidCastException** درون متد **CompareTo** تولید می‌شود.

هر دوی این مشکلات می‌توانند با استفاده از EIMI حل شوند. نسخه‌ی تغییریافته‌ی **SomeValueType** که یک EIMI بدان اضافه شده است:

```
internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }

    public Int32 CompareTo(SomeValueType other) {
        return(m_x - other.m_x);
    }

    // NOTE: No public/private used on the next line
    Int32 IComparable.CompareTo(Object other) {
        return CompareTo((SomeValueType) other);
    }
}
```

به چندین تغییر در این نسخه‌ی جدید دقت کنید. نخست اینکه دو متد **CompareTo** دارد. اولین متد **CompareTo**، دیگر یک **Object** را به عنوان پارامتر نمی‌گیرد، اکنون به جای آن، یک **SomeValueType** می‌گیرد. چون این پارامتر تغییر یافته است، کدی که **other** را به **SomeValueType** تبدیل می‌کند، دیگر نیاز نیست و حذف شده است. دوم اینکه تغییریافته‌ی اولین متد **CompareTo** برای نوع-امن کردن آن به این معنی است که **SomeValueType** دیگر به قراردادی که با پیاده‌سازی رابط **IComparable** داشت، پایبند نیست. کلاس **SomeValueType** باید یک متد **CompareTo** که قرارداد **IComparable** را رعایت کند، پیاده‌سازی نماید. این وظیفه‌ی دومین متد **IComparable.CompareTo** است، می‌باشد.

با اعمال این دو تغییر، اکنون امنیت نوع در زمان کامپایل را داشته و دیگر بسته‌بندی نیاز نیست:

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v);    // No boxing
    n = v.CompareTo(o);        // compile-time error
}
```

هرچند، ما یک متغیر از نوع رابط تعریف می‌کنیم، مجدداً امنیت نوع در زمان کامپایل را از دست داده و با بسته‌بندی ناخواسته مواجه می‌شویم:

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    IComparable c = v; // Boxing!
```

```

Object o = new Object();
Int32 n = c.CompareTo(v); // Undesired boxing
n = c.CompareTo(o); // InvalidCastException
}

```

در حقیقت، همانگونه که قبلا در این فصل گفتیم، هنگام تبدیل یک نمونه نوع مقداری به یک نوع رابط، CLR باید نمونه نوع مقداری را بسته‌بندی کند. به خاطر این، در متد **Main** قبلی، دو بسته‌بندی رخ خواهد داد. EIMI ها غالبا هنگام پیاده‌سازی رابط‌هایی مثل **IList**، **ICollection**، **IConvertible** و **IDictionary** استفاده می‌شوند، آن‌ها به شما اجازه می‌دهند نسخه‌های نوع-امن از متدهای این رابط‌ها را بسازید و شما را قادر می‌سازند عملیات بسته-بندی روی نوع‌های مقداری را کاهش دهید.

هنگام پیاده‌سازی های صریح متد رابط مراقب باشید

این بسیار مهم است که شما نکات ریزی که هنگام استفاده از EIMI ها وجود دارد را درک کنید و به خاطر این نکات، باید تا حد ممکن از EIMI ها خودداری کنید. خوشبختانه، رابط‌های جنریک به شما کمک می‌کنند تا حد زیادی از EIMI ها خودداری نمایید. اما هنوز ممکن است مواقعی باشد که شما نیاز به استفاده از آن‌ها دارید (مثل پیاده‌سازی متدهای دو رابط با نام و امضای یکسان). مشکلات بزرگی که با EIMI ها همراه است عبارتند از:

- هیچ مستنداتی که توضیح دهد یک نوع چگونه یک متد EIMI را پیاده‌سازی می‌کند وجود ندارد و هیچ گونه پشتیبانی **IntelliSense** و **ویژوال استودیو** نیز وجود ندارد.
- نمونه‌های نوع مقداری هنگام تبدیل به یک رابط بسته‌بندی می‌شوند.
- یک EIMI نمی‌تواند توسط یک نوع مشتق شده فراخوانی شود.

بگذارید دقیق‌تر به این مشکلات نگاه کنیم:

هنگام بررسی متدهای یک نوع در مستندات مرجع دات‌نت فریمورک، پیاده‌سازی‌های صریح متد رابط لیست شده اند، اما هیچ راهنمایی مخصوص به یک نوع، وجود ندارد، شما فقط می‌توانید راهنمایی عمومی درباره‌ی متدهای رابط را بخوانید. برای نمونه، مستندات برای نوع **Int32** نشان می‌دهد که این نوع، تمام متدهای رابط **IConvertible** را پیاده‌سازی کرده است. این خوب است چون برنامه‌نویسان می‌دانند که این متدها وجود دارند، هرچند برای برنامه-نویس بسیار گیج‌کننده است چون شما نمی‌توانید یک متد **IConvertible** را مستقیما روی یک **Int32** فراخوانی کنید. برای نمونه، متد زیر کامپایل نمی‌شود:

```

public static void Main() {
    Int32 x = 5;
    Single s = x.ToSingle(null); // Trying to call an IConvertible method
}

```

هنگام کامپایل این متد، کامپایلر سی‌شارپ خطای زیر را اعلام می‌کند:

“message117: ‘int’ does not contain a definition for ‘ToSingle’.”

این پیام خطا، برنامه‌نویس را گیج می‌کند چون به وضوح می‌گوید نوع **Int32** یک متد **ToSingle** را تعریف نکرده است، در حالیکه در حقیقت تعریف کرده است.

برای فراخوانی **ToSingle** روی یک **Int32**، شما باید ابتدا **Int32** را به یک **IConvertible** تبدیل کنید، همانگونه که در متد زیر نشان داده شده است:

```

public static void Main() {
    Int32 x = 5;
    Single s = ((IConvertible) x).ToSingle(null);
}

```

نیاز به این تبدیل اصلا معلوم نیست و بسیاری از برنامه‌نویسان خودشان به این راه حل نمی‌رسند. اما یک مسئله باز هم بزرگتر وجود دارد. تبدیل نوع مقداری **Int32** به یک **IConvertible** باعث بسته‌بندی نوع می‌شود، حافظه را هدر داده و عملکرد را ضعیف می‌کند. این دومین مشکل بزرگی است که من در ابتدای فصل به آن اشاره کردم.

سومین و احتمالا بزرگترین مشکلات با EIMI ها اینست که آن‌ها توسط یک کلاس مشتق شده قابل فراخوانی نیستند. یک مثال در این زمینه:

```

internal class Base : IComparable {

    // Explicit Interface Method Implementation
    Int32 IComparable.CompareTo(Object o) {
        Console.WriteLine("Base's CompareTo");
        return 0;
    }
}

internal sealed class Derived : Base, IComparable {

    // A public method that is also the interface implementation
    public Int32 CompareTo(Object o) {
        Console.WriteLine("Derived's CompareTo");

        // This attempt to call the base class's EIMI causes a compiler error:
        // error CS0117: 'Base' does not contain a definition for 'CompareTo'
        base.CompareTo(o);
        return 0;
    }
}

```

در متد **CompareTo** از **Derived**، من سعی می‌کنم **base.CompareTo** را فراخوانی کنم، اما این باعث می‌شود کامپایلر سی‌شارپ یک خطا اعلام کند. مشکل اینجاست که کلاس **Base** یک متد عمومی یا محافظت شده **CompareTo** ارائه نمی‌کند تا بتواند فراخوانی شود، آن، یک متد **CompareTo** ارائه می‌کند که تنها توسط یک متغیر از نوع **IComparable** قابل فراخوانی است. من می‌توانستم متد **CompareTo** از **Derived** را تغییر دهم تا شبیه به این شود:

```

// A public method that is also the interface implementation
public Int32 CompareTo(Object o) {
    Console.WriteLine("Derived's CompareTo");

    // This attempt to call the base class's EIMI causes infinite recursion
    IComparable c = this;
    c.CompareTo(o);
    return 0;
}

```

در این نسخه، من **this** را به یک متغیر **IComparable** تبدیل می‌کنم، **c**. و سپس از **c** برای فراخوانی **CompareTo** استفاده می‌کنم. هر چند، متد عمومی **CompareTo** از **Derived** به عنوان پیاده‌سازی متد **CompareTo** از **IComparable** برای **Derived**، عمل می‌کند و بنابراین، بازگشتی بی‌نهایت رخ می‌دهد. این مشکل با تعریف کلاس **Derived** بدون رابط **IComparable** حل می‌شود، همانند این:

```

internal sealed class Derived : Base /*, IComparable */ { ... }

```

حال، متد **CompareTo**، متد **CompareTo** در **Base** را فراخوانی می‌کند. اما گاهی شما به راحتی نمی‌توانید رابط را از نوع حذف کنید چون شما می‌خواهید نوع مشتق شده، یک رابط را پیاده‌سازی کند. بهترین راه حل این مشکل، اینست که کلاس پایه، یک متد مجازی علاوه بر متد رابط که صریحاً پیاده‌سازی کرده است، فراهم کند. آنگاه، کلاس **Derived** می‌تواند متد مجازی را بازنویسی کند. روش صحیح تعریف کلاس‌های **Base** و **Derived** اینگونه است:

```

internal class Base : IComparable {

    // Explicit Interface Method Implementation
    Int32 IComparable.CompareTo(Object o) {

```

```

    Console.WriteLine("Base's IComparable CompareTo");
    return CompareTo(o); // This now calls the virtual method
}

// virtual method for derived classes (this method could have any name)
public virtual Int32 CompareTo(Object o) {
    Console.WriteLine("Base's virtual CompareTo");
    return 0;
}

internal sealed class Derived : Base, IComparable {

    // A public method that is also the interface implementation
    public override Int32 CompareTo(Object o) {
        Console.WriteLine("Derived's CompareTo");

        // Now, we can call Base's virtual method
        return base.CompareTo(o);
    }
}

```

توجه کنید که من متد مجازی فوق را به عنوان یک متد عمومی تعریف کردم، اما در بعضی موارد، شما ترجیح می‌دهید متد مرا محافظت شده کنید. این خوب است که به جای عمومی، متد را محافظت شده کنید. اما این کار تغییرات کوچکی را ناگزیر می‌سازد. این بحث به وضوح نشان داد که EIMI ها باید با دقت زیادی استفاده شوند. وقتی بسیاری از برنامه‌نویسان در ابتدا درباره‌ی EIMI ها یاد می‌گیرند، فکر می‌کنند آن‌ها خوب هستند و هر جا بتوانند از آن‌ها استفاده می‌کنند. این کار را نکنید. EIMI ها در بعضی شرایط مفید هستند اما شما باید هر جا ممکن است از آن‌ها خودداری کنید چون استفاده از یک نوع را بسیار مشکل می‌کنند.

طراحی: کلاس پایه یا رابط ؟

من اغلب این سوال را می‌شنوم "من باید یک کلاس پایه طراحی کنم یا یک رابط؟" جواب سوال همیشه واضح نیست. راهنمایی‌های زیر به شما کمک می‌کند:

- **رابطه‌ی IS-A در مقابل CAN-DO** یک نوع فقط می‌تواند یک پیاده‌سازی را به ارث ببرد. اگر نوع مشتق شده نتواند یک رابطه IS-A ("هست یک") با نوع پایه برقرار کند، از یک نوع پایه استفاده نکنید. از یک رابط استفاده کنید. رابط‌ها یک رابطه‌ی CAN-DO ("می‌تواند انجام دهد") را بیان می‌کنند. اگر کاربرد CAN-DO به نظر می‌رسد متعلق به نوع‌های شی متنوع باشد، از یک رابط استفاده کنید. برای نمونه، یک نوع می‌تواند نمونه‌های خودش را به نوع دیگر (IConvertible) تبدیل کند، یک نوع می‌تواند یک نمونه از خودش را سرپالی کند (ISerializable) و غیره. دقت کنید که نوع‌های مقداری باید از System.ValueType مشتق شوند و بنابراین آن‌ها نمی‌توانند از یک کلاس پایه دلخواه مشتق شوند. در این حالت، شما باید از یک رابطه‌ی CAN-DO استفاده کنید و یک رابط تعریف کنید.
- **استفاده آسان** معمولاً برای شما به عنوان برنامه‌نویس آسانتر است یک نوع مشتق شده از یک نوع پایه تعریف کنید تا اینکه تمام متدهای یک رابط را پیاده‌سازی نمایید. نوع پایه می‌تواند کاربردهای زیادی را فراهم کند، پس نوع مشتق شده احتمالاً تنها به بخشی از رفتار تغییر یافته آن نیازمند است. اگر شما یک رابط تعریف کنید، نوع جدید باید تمام اعضا را پیاده‌سازی کند.
- **پیاده‌سازی استوار** بدون توجه به اینکه چقدر خوب، یک رابط مستندسازی شده باشد، خیلی بعید است که هر فردی، قرارداد را ۱۰۰ درصد صحیح پیاده‌سازی کند. در واقع، COM از این مشکل رنج می‌برد که بهمین خاطر است که بعضی اشیاء COM فقط با Internet Explorer و Microsoft Office Word به درستی کار می‌کنند. با فراهم کردن یک نوع پایه با یک پیاده‌سازی پیش فرض خوب، شما از نوعی استفاده می‌کنید که کار می‌کند و به خوبی تست شده است. آنگاه می‌توانید بخش‌هایی که نیاز به تغییر دارند را تغییر دهید.

▪ **نسخه‌بندی** اگر شما یک متد به نوع پایه اضافه کنید، نوع مشتق شده، متد جدید را به ارث می‌برد، شما از نوعی استفاده می‌کنید که کار می‌کند و سورس‌کد کاربر، حتی نیاز به کامپایل مجدد ندارد. افزودن یک عضو جدید به رابط، اجبار می‌کند ارث برنده‌ی رابط، سورس‌کدش را تغییر داده و آن را مجدداً کامپایل کند.

در FCL، کلاس‌ها مربوط به استریم داده‌ها از یک پیاده‌سازی وراثت استفاده می‌کند. کلاس **System.IO.Stream** کلاس پایه خلاصه است. آن، تعدادی متد مثل **Read** و **Write** فراهم می‌کند. دیگر کلاس‌ها، **System.IO.FileStream**، **System.IO.MemoryStream** و **System.Net.Sockets.NetworkStream** از **Stream** مشتق شده‌اند. مایکروسافت یک رابطه IS-A را بین هر یک از این سه کلاس و کلاس **Stream** انتخاب کرده است چون پیاده‌سازی کلاس‌های بهم مرتبط را آسان می‌کند. برای نمونه، کلاس‌های مشتق شده، تنها نیاز به پیاده‌سازی عملیات همزمان ورودی خروجی دارند، آن‌ها قابلیت انجام عملیات‌های غیرهمزمان ورودی خروجی را از کلاس پایه **Stream** به ارث می‌برند.

اقرار می‌کنم انتخاب وراثت برای کلاس‌های استریم، کاملاً روشن و واضح نیست؛ کلاس پایه **Stream**، پیاده‌سازی کوچکی را فراهم می‌کند. هر چند، اگر شما به کلاس‌های کنترل **Windows Forms** نگاه کنید که **Button**، **CheckBox**، **ListBox** و تمام دیگر کنترل‌ها از **System.Windows.Forms.Control** مشتق شده‌اند، ساده است که تمام کدی که تمام کدی که **Control** پیاده‌سازی کرده است را تصور کنید که کلاس‌های کنترل مختلف، به سادگی برای کارکرد صحیح خود، آن را به ارث می‌برند.

بر خلاف آن، مایکروسافت مجموعه‌های FCL را بر اساس رابط طراحی کرده است. فضای نام **System.Collections.Generic** چندین رابط مربوط به مجموعه‌ها را تعریف می‌کند. **ICollection<T>**، **IEnumerable<out T>**، **IDictionary<TKey, TValue>**، مایکروسافت تعدادی کلاس مثل **List<T>**، **Dictionary<TKey, TValue>**، **Queue<T>**، **Stack<T>** و غیره را فراهم کرده است که ترکیب‌هایی از این رابط‌ها را پیاده‌سازی می‌کنند. در اینجا طراحان، یک رابطه CAN-DO را بین کلاس‌ها و رابط‌ها انتخاب کرده‌اند چون پیاده‌سازی کلاس‌های مجموعه مختلف اساساً با یکدیگر متفاوتند. به بیان دیگر، کد اشتراکی زیادی بین یک **List<T>**، یک **Dictionary<TKey, TValue>** و یک **Queue<T>** وجود ندارد.

با این وجود، عملیات‌هایی که این کلاس‌ها ارائه می‌کنند بسیار یکنواخت است. برای نمونه، همه‌ی آن‌ها یک مجموعه از عناصر را نگهداری می‌کنند که می‌توانند شمرده شوند، و تمام آن‌ها اجازه‌ی افزودن و حذف عناصر را می‌دهند. اگر شما یک اشاره‌گر به یک شی که نوعش رابط **IList<T>** را پیاده‌سازی می‌کند، دارید، شما می‌توانید کدی بنویسید که عناصر را اضافه کند، عناصر را حذف کند، و به دنبال یک عنصر بگردد و تمام این‌ها را بدون دانستن نوع دقیق مجموعه‌ای که روی آن کار می‌کنید، انجام دهید. این یک مکانیزم بسیار قدرتمند است.

سرانجام، باید اشاره شود که شما واقعاً می‌توانید هر دو کار را انجام دهید، یک رابط تعریف کنید و یک نوع پایه که رابط را پیاده‌سازی می‌کند، فراهم کنید. برای نمونه، رابط **FCL** رابط **IComparer<in T>** را تعریف می‌کند و هر نوعی می‌تواند انتخاب کند که این رابط را پیاده‌سازی کند. به علاوه، FCL یک کلاس پایه خلاصه، **Comparer<T>** که این رابط را پیاده‌سازی می‌کند فراهم می‌نماید. داشتن هر دوی یک رابط و یک کلاس پایه، انعطاف‌پذیری بالایی را فراهم می‌کند چون برنامه‌نویسان می‌توانند انتخاب کنند که کدام را ترجیح می‌دهند.

فصل ۱۴: کاراکترها، رشته‌ها و کار با متن

در این فصل، من مکانیزم کار با تک‌تک کاراکترها و رشته‌ها در دات‌نت فریمورک مایکروسافت را توضیح می‌دهم. با صحبت درباره‌ی ساختار **System.Char** شروع می‌کنم و راه‌های مختلف دستکاری یک کاراکتر را می‌گویم. سپس به سراغ کلاس پر کاربردتر **System.String** می‌روم، که به شما اجازه می‌دهد با رشته‌های تغییرناپذیر کار کنید. (وقتی ساخته می‌شوند، رشته‌ها به هیچ روشی قابل تغییر نیستند.) پس از بررسی رشته‌ها من به شما نشان می‌دهم چگونه عملیات‌های متنوع را به صورت کارا انجام دهید تا یک رشته را به صورت پویا از طریق کلاس **System.Text.StringBuilder** بسازید. پس از گفتن مبانی رشته‌ها، سپس روش فرمت‌شی‌ها به رشته‌ها و روش‌های نگهداری و ارسال رشته‌ها با اینکدینگ‌های مختلف را خواهیم گفت. در پایان، من کلاس **System.Security.SecureString** را بحث می‌کنم که می‌تواند برای محافظت رشته‌های حساس مثل رمز عبور و اطلاعات کارت اعتباری استفاده شود.

کاراکترها

در دات‌نت فریمورک، کاراکترها همیشه با مقادیر ۱۶ بیتی یونیکد نمایش داده می‌شوند، که ساخت برنامه‌های جهانی را آسان می‌کند. یک کاراکتر با یک نمونه از ساختار **System.Char** (یک نوع مقداری) نمایش داده می‌شوند. نوع **System.Char** بسیار ساده است. دو فیلد ثابت عمومی فقط-خواندنی ارائه می‌کند، **MinValue** که به عنوان ' ' تعریف شده و **MaxValue** که بعنوان '\uffff' تعریف شده است.

با یک نمونه از یک **Char**، شما می‌توانید متد **GetUnicodeCategory** را فراخوانی کنید که یک مقدار از نوع شمارشی **System.Globalization.UnicodeCategory** را برمی‌گرداند. این مقدار بیان می‌کند که کاراکتر، یک کاراکتر کنترلی، یک نماد ارز (نرخ پول)، یک حرف کوچک، یک حرف بزرگ، یک کاراکتر علامت گذاری، یک نماد ریاضی یا یک کاراکتر دیگر (که توسط استاندارد یونیکد تعریف شده) می‌باشد.

برای ساده کردن برنامه‌نویسی، نوع **Char** چندین متد استاتیک نیز ارائه می‌کند، مثل **IsUpper**، **IsWhiteSpace**، **IsLetter**، **IsDigit**، **IsLowerSurrogate**، **IsSurrogate**، **IsSeperator**، **IsNumber**، **IsControl**، **IsLetterDigit**، **IsPunctuation**، **IsHighSurrogate** و **IsSymbol**. اکثر این متدها در درون خود، **GetUnicodeCategory** را فراخوانی کرده و بر طبق آن، **true** یا **false** برمی‌گرداند. توجه کنید تمام این متدها یا یک تک کاراکتر برای پارامتر می‌گیرند یا یک **String** به‌همراه اندیسی از یک کاراکتر درون **String**.

به علاوه، شما می‌توانید یک کاراکتر را به معادل حرف کوچک یا حرف بزرگش به روشی مستقل از فرهنگ با فراخوانی متد استاتیک **ToLowerInvariant** یا **ToUpperInvariant** تبدیل کنید. متدهای **ToLower** و **ToUpper**، کاراکتر را با استفاده از اطلاعات فرهنگ مربوط به ترد فراخوانی کننده (که متدها، آن را در درون از طریق ویژگی **CurrentCulture** از کلاس **System.Threading.Thread** دریافت می‌کنند) تبدیل می‌نمایند. شما همچنین می‌توانید یک فرهنگ خاص را با ارسال یک نمونه از کلاس **CultureInfo** به این متدها تعیین کنید. **ToLower** و **ToUpper** نیاز به اطلاعات فرهنگ دارند چون بزرگی و کوچکی حروف، وابسته به فرهنگ است. برای نمونه ترک‌ها حرف بزرگ برای U+0069 (حرف کوچک لاتین ا) را U+0130 (حرف بزرگ لاتین ا با نقطه بالای آن) در نظر می‌گیرند، در حالیکه دیگر فرهنگ‌ها آن را U+0049 (حرف بزرگ لاتین ا) در نظر می‌گیرند.

گذشته از این متدهای استاتیک، نوع **Char** تعدادی متد نمونه از خودش نیز ارائه می‌کند. متد **Equals** در صورتی که دو نمونه **Char**، کد ۱۶ بیتی یونیکد یکسانی را نمایش دهند، **true** برمی‌گرداند. متدهای **CompareTo** (تعریف شده توسط رابط‌های **IComparable/Comparable<Char>**) مقایسه‌ای از دو نمونه **Char** را بر می‌گرداند، این مقایسه حساس به فرهنگ نیست.

متد **ConvertFromUtf32** یک رشته شامل دو کاراکتر UTF-16 از یک تک کاراکتر UTF-32 تولید می‌کند. **ConvertToUtf32** یک کاراکتر UTF-16 از یک جفت‌جانشین^{۵۲} بالا/پایین یا از یک رشته تولید می‌کند. متد **ToString** یک **String** حاوی یک تک کاراکتر را برمی‌گرداند. بر عکس **ToString**، **Parse/TryParse** قرار دارد که یک **String** تک کاراکتر را گرفته و کد UTF-16 آنرا برمی‌گرداند. آخرین متد، **GetNumericValue**، معادل عددی یک کاراکتر را برمی‌گرداند، من این متد را در کد زیر نشان داده‌ام:

```
using System;
```

^{۵۲} یک **Surrogate Pair**، ترکیب دو کاراکتر یونیکد است برای کاراکترهایی که نمایش معتبری در یونیکد ندارند.

```
public static class Program {
    public static void Main() {
        Double d; // '\u0033' is the "digit 3"
        d = Char.GetNumericValue('\u0033'); // '3' would work too
        Console.WriteLine(d.ToString()); // Displays "3"

        // '\u00bc' is the "vulgar fraction one quarter ('¼')"
        d = Char.GetNumericValue('\u00bc');
        Console.WriteLine(d.ToString()); // Displays "0.25"

        // 'A' is the "Latin capital letter A"
        d = Char.GetNumericValue('A');
        Console.WriteLine(d.ToString()); // Displays "-1"
    }
}
```

سرانجام، سه تکنیک به شما اجازه‌ی تبدیل بین نوع‌های عددی مختلف و نمونه‌های **Char** و بالعکس را می‌دهد. تکنیک‌ها به ترتیب ارجحیت لیست شده‌اند:

- **تبدیل (Casting)** راحت‌ترین راه تبدیل یک **Char** به یک مقدار عددی مثل یک **Int32**. تبدیل (Casting) می‌باشد. از این سه تکنیک، این مورد کاراترین است چون کامپایلر دستورات زبان میانی (IL) ای تولید می‌کند که تبدیل را انجام می‌دهد، هیچ متدی نیاز نیست فراخوانی شود. به علاوه، بعضی زبان‌ها (مثل سی‌شارپ) به شما اجازه می‌دهند که تبدیل با کد بررسی شده یا بررسی نشده، انجام پذیرد (که در فصل ۵ "نوع‌های اصلی، ارجاعی و مقداری" بحث شد).
- **استفاده از نوع Convert** نوع **System.Convert**، چندین متد عمومی ارائه می‌کند که قادرند یک **Char** را به یک نوع عددی و بالعکس تبدیل کنند. تمام این متدها تبدیل را به عنوان یک عمل بررسی شده، انجام می‌دهند، که باعث می‌شود در صورتی که تبدیل منجر به از دست رفتن داده شود، یک **OverflowException** تولید گردد.
- **استفاده از رابط IConvertible** نوع **Char** و تمام نوع‌های عددی در کتابخانه کلاس فریمورک (FCL) رابط **IConvertible** را پیاده‌سازی می‌کنند. این رابط متدهایی مثل **ToUInt16** و **ToChar** تعریف می‌کند. این تکنیک در بین سه روش، کمترین کارایی را دارد چون فراخوانی یک متد رابط روی یک نوع مقداری نیاز دارد که نمونه بسته‌بندی شود — **Char** و تمام نوع‌های عددی، نوع مقداری هستند. متدهای **IConvertible** در صورتی که نوع نتواند تبدیل شود یا اگر تبدیل منجر به از دست دادن داده شود (مثل تبدیل یک **Char** به یک **Boolean**) یک **System.InvalidCastException** تولید می‌کنند. توجه کنید نوع‌های زیادی (شامل نوع **Char** و نوع‌های عددی) متدهای **IConvertible** را با پیاده‌سازی صریح متد رابط (که در فصل ۱۳ "رابط‌ها" بحث شد) پیاده‌سازی می‌کنند. این یعنی شما قبل از آنکه بتوانید هر یک از متدهای رابط را فراخوانی کنید، باید صریحاً نمونه را به یک **IConvertible** تبدیل (cast) کنید. تمام متدهای **IConvertible** به جز **GetTypeCode** یک اشاره‌گر به یک شی که رابط **IFormatProvider** را پیاده‌سازی می‌کند، دریافت می‌کنند. این پارامتر سودمند است اگر به دلایلی، تبدیل، نیاز به اطلاعات فرهنگ داشته باشد. برای اغلب تبدیل‌ها، شما می‌توانید **null** را برای این پارامتر ارسال کنید چون در هر صورت، نادیده گرفته می‌شود.

کد زیر استفاده از این سه تکنیک را نشان می‌دهد:

```
using System;

public static class Program {
    public static void Main() {
        Char c;
        Int32 n;

        // Convert number <-> character using C# casting
```

```

c = (Char) 65;
Console.WriteLine(c);           // Displays "A"

n = (Int32) c;
Console.WriteLine(n);           // Displays "65"

c = unchecked((Char) (65536 + 65));
Console.WriteLine(c);           // Displays "A"

// Convert number <-> character using Convert
c = Convert.ToChar(65);
Console.WriteLine(c);           // Displays "A"

n = Convert.ToInt32(c);
Console.WriteLine(n);           // Displays "65"

// This demonstrates Convert's range checking
try {
    c = Convert.ToChar(70000);    // Too big for 16 bits
    Console.WriteLine(c);        // Doesn't execute
}
catch (OverflowException) {
    Console.WriteLine("Can't convert 70000 to a Char.");
}

// Convert number <-> character using IConvertible
c = ((IConvertible) 65).ToChar(null);
Console.WriteLine(c);           // Displays "A"

n = ((IConvertible) c).ToInt32(null);
Console.WriteLine(n);           // Displays "65"
}
}

```

نوع System.String

یکی از نوع‌های پرکاربرد در هر برنامه‌ای، **System.String** است. یک **String**، یک مجموعه تغییر ناپذیر از کاراکترها را نمایش می‌دهد. نوع **String** مستقیماً از **Object** مشتق شده است، که آن را یک نوع ارجاعی می‌کند و بنابراین اشیاء **String** (آرایه کاراکترهایش) همیشه در هیپ زندگی می‌کنند و هرگز در پشته‌ی ترد قرار ندارند. نوع **String** چند رابط را نیز پیاده‌سازی می‌کند (**String**، **ICloneable**، **IComparable/IComparable<String>**، **IConvertible**، **IEnumerable/IEnumerable<Char>** و **IComparable<String>**).

ساختن رشته‌ها

بسیاری از زبان‌های برنامه‌نویسی (شامل سی‌شارپ)، **String** را یک نوع اصلی در نظر می‌گیرند – یعنی، کامپایلر به شما اجازه می‌دهد رشته‌های تحت-لفظی را مستقیماً در سورس‌کدتان بیان کنید. کامپایلر این رشته‌ها را در متادیتای ماژول قرار می‌دهد و آن‌ها در زمان اجرا بارگذاری و ارجاع می‌شوند.

در سی‌شارپ شما نمی‌توانید از عملگر **new** برای ساخت یک شیء **String** از روی یک رشته تحت‌لفظی استفاده کنید:

```
using System;
```

```
public static class Program {
    public static void Main() {
        String s = new String("Hi there.");    // <-- Error
        Console.WriteLine(s);
    }
}
```

به جای آن، شما باید از نحو ساده شده‌ی زیر استفاده کنید:

```
using System;
```

```
public static class Program {
    public static void Main() {
        String s = "Hi there.";
        Console.WriteLine(s);
    }
}
```

اگر شما این کد را کامپایل کرده و IL آن را (با ILDasm.exe) بررسی کنید، محتویات زیر را می‌بینید:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      13 (0xd)
    .maxstack 1
    .locals init (string v_0)
    IL_0000: ldstr  "Hi there."
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: call  void [mscorlib]System.Console::writeLine(string)
    IL_000c: ret
} // end of method Program::Main
```

دستور **IL**، **newobj**، یک نمونه از یک شی می‌سازد. اما هیچ دستور **newobj** در کد این مثال وجود ندارد. به جای آن، شما دستور خاص **ldstr** (بارگذاری رشته **load string**) را می‌بینید، که یک شی **String** را با استفاده از رشته تحت‌اللفظی که از متادیتا بدست می‌آید، می‌سازد. این نشان می‌دهد که CLR در حقیقت روش ویژه‌ای برای ساخت اشیاء **String** تحت‌اللفظی دارد.

اگر شما از کد ناامن استفاده می‌کنید، می‌توانید یک شی **String** را از روی یک **Char*** یا **SByte*** بسازید. برای انجام این کار، شما از عملگر **new** سی‌شارپ استفاده کرده و یکی از سازنده‌هایی که نوع **String** فراهم می‌کند و پارامترهای **Char*** یا **SByte*** می‌گیرد را فراخوانی کنید. این سازنده‌ها یک شی **String** می‌سازند و رشته را از روی یک آرایه از نمونه‌های **Char** یا بایت‌های علامت دار، مقداردهی اولیه می‌کنند. دیگر سازنده‌ها پارامتر اشاره‌گری ندارد و تنها توسط کد امن (قابل بازبینی) نوشته شده در هر زبان برنامه‌نویسی مدیریت‌شده، قابل فراخوانی هستند.

سی‌شارپ نحو ویژه‌ای برای کمک به شما در وارد کردن رشته‌های تحت‌اللفظی در سورس‌کدتان، ارائه می‌کند. برای کاراکترهای خاص مثل خط جدید، اینتر و فاصله، سی‌شارپ از مکانیزم **escape** که برنامه‌نویسان **C/C++** با آن آشنا هستند استفاده می‌کند:

```
// String containing carriage-return and newline characters
String s = "Hi\r\nthere.";
```

مهم اگرچه مثال قبل، کاراکترهای خط جدید و اینتر (carriage-return) را در رشته قرار می دهد، اما این روش را توصیه نمی کنیم. به جای آن، نوع **System.Environment.NewLine**، یک ویژگی فقط-خواندنی **NewLine** تعریف می کند که یک رشته شامل این کاراکترها وقتی برنامه شما در ویندوز مایکروسافت در حال اجراست، بر می گرداند. هر چند، ویژگی **NewLine** حساس به پلتفرم است و رشته‌ی مناسب را با بدست آوردن کاراکتر خط جدید در آن پلتفرم، بر می گرداند. پس برای نمونه، اگر زیرساختار مشترک زبان (CLI) به یک سیستم **UNIX** انتقال داده شود، ویژگی **NewLine**، یک رشته شامل یک تک کاراکتر **\n** را بر می گرداند. روش صحیح تعریف رشته قبلی که در هر پلتفرمی به درستی کار کند اینگونه است:

```
String s = "Hi" + Environment.NewLine + "there.";
```

شما می توانید چندین رشته را به هم اتصال داده و یک رشته‌ی جدید با استفاده از عملگر + سی شارپ بسازید:

```
// Three literal strings concatenated to form a single literal string
String s = "Hi" + " " + "there.";
```

در این کد، چون تمام رشته‌ها تحت‌اللفظی هستند، کامپایلر سی شارپ آن‌ها را در زمان کامپایل به هم متصل کرده و تنها یک رشته – **“Hi there.”** – را در یک متادیتای ماژول قرار می دهد. برای اتصال چندین رشته به همدیگر در زمان اجرا، از استفاده از عملگر + خودداری کنید چون آن، چندین شی رشته در هیپ می سازد که باید جمع آوری گردند. به جای آن از نوع **System.Text.StringBuilder** استفاده کنید (که بعداً در این فصل، آن را توضیح خواهیم داد). سرانجام، سی شارپ یک روش ویژه برای تعریف یک رشته ارائه می کند که در آن تمام کاراکترهای بین دو نقل قول به عنوان بخشی از رشته در نظر گرفته می شوند. این تعاریف ویژه، رشته‌های تحت‌اللفظی **verbatim strings** نامیده می شوند. (معمولاً هنگام تعیین مسیر یک فایل یا دایرکتوری یا هنگام کار با عبارت‌های منظم، استفاده می شوند. کد زیر نحوه‌ی تعریف یک رشته یکسان با و بدون استفاده از کاراکتر رشته تحت‌اللفظی (@) را نشان می دهد:

```
// Specifying the pathname of an application
String file = "C:\\windows\\System32\\Notepad.exe";
```

```
// Specifying the pathname of an application by using a verbatim string
String file = @"C:\windows\System32\Notepad.exe";
```

شما می توانید هر کدام از این دو خط را در یک برنامه استفاده کنید چون آن‌ها رشته‌های یکسانی در متادیتای ماژول تولید می کنند. به هر حال، نماد @ قبل از رشته در دومین خط به کامپایلر می گوید که رشته، یک رشته‌ی تحت‌اللفظی است. در عمل، این به کامپایلر می گوید با کاراکترهای **backslash**، به جای کاراکترهای **escape** به عنوان کاراکترهای **backslash** رفتار کند، که مسیر تعیین شده در کد شما را بسیار خواناتر می کند. حال که نحوه‌ی ساخت یک رشته را دیدید، بگذارید کمی درباره‌ی عملیاتی که می توانید روی اشیاء **String** انجام دهید، صحبت کنیم.

رشته‌ها تغییر ناپذیرند

مهمترین چیزی که باید درباره‌ی یک شی **String** بدانید اینست که آن، تغییر ناپذیر است. یعنی، وقتی ساخته می شود، یک رشته هرگز طولانی تر نمی شود، کوتاه تر نمی شود، یا هیچ یک از کاراکترهایش تغییر نمی کند. تغییر ناپذیر بودن رشته‌ها، چندین مزیت دارد. اول اینکه، به شما اجازه می دهد بدون تغییر دادن رشته، روی آن عملیات انجام دهید:

```
if (s.ToUpperInvariant().Substring(10, 21).EndsWith("EXE")) {
    ...
}
```

در اینجا، **ToUpperInvariant**، یک رشته‌ی جدید بر می گرداند؛ کاراکترهای رشته‌ی **s** را تغییر نمی دهد. **Substring** بر روی رشته‌ی برگشتی توسط **ToUpperInvariant** عمل می کند و یک رشته بر می گرداند که توسط **EndsWith** بررسی می شود. دو رشته‌ی موقتی تولید شده توسط **ToUpperInvariant** و **Substring** برای مدت زمان طولانی توسط کد برنامه ارجاع نمی شوند و جمع آوری کننده‌ی زباله، حافظه‌ی آن‌ها را در جمع آوری بعدی پس می گیرد. اگر شما تعداد زیادی دستکاری روی رشته انجام دهید، شما تعداد زیادی شی **String** در هیپ ساخته اید که باعث جمع آوری‌های بیشتر می گردد و عملکرد برنامه شما را تضعیف می کند. برای انجام تعداد زیادی دستکاری روی رشته به صورت کارا، از کلاس **StringBuilder** استفاده کنید. داشتن رشته‌های تغییر ناپذیر همچنین یعنی هیچ مشکل همزمانی ترد هنگام دستکاری یا دسترسی به یک رشته وجود ندارد. به علاوه، برای CLR این

امکان وجود دارد که محتویات چندین **String** یکسان را از طریق یک تک شی **String** به اشتراک بگذارد. این کار تعداد رشته‌های سیستم را کاهش داده – حافظه کمتری مصرف می‌شود – و این چیزی است که وارد کردن رشته (که بعداً در این فصل بحث می‌شود)، تماماً درباره‌ی آن است.

به دلایل عملکردی، نوع **String** بسیار با CLR هماهنگ است. به خصوص CLR قالب بندی دقیق فیلدهای تعریف شده در نوع **String** را می‌داند و CLR مستقیماً به این فیلدها دسترسی پیدا می‌کند. این دسترسی مستقیم و عملکرد بهتر همراه با کمی هزینه‌ی برنامه‌نویسی است: کلاس **String** مهر شده (sealed) است که یعنی شما نمی‌توانید از آن به عنوان کلاس پایه برای نوع خودتان استفاده کنید. اگر شما قادر به تعریف نوع خودتان با نوع **String** به عنوان کلاس پایه بودید، شما می‌توانستید برخی فرضیاتی که تیم CLR درباره‌ی تغییر ناپذیری رشته‌های **String** دارند را بر هم بزنید.

مقایسه رشته‌ها

مقایسه، احتمالاً رایج‌ترین عملی است که روی رشته‌ها انجام می‌شود. دو دلیل برای مقایسه رشته با همدیگر وجود دارد. ما رشته‌ها را برای تعیین برابری یا مرتب کردن آن‌ها (معمولاً برای ارائه به یک کاربر) مقایسه می‌کنیم. در تعیین برابری رشته یا هنگام مقایسه برای مرتب سازی، قویاً توصیه می‌شود شما یکی از این متدها (تعریف شده توسط کلاس **String**) را فراخوانی کنید:

```
Boolean Equals(String value, StringComparison comparisonType)
static Boolean Equals(String a, String b, StringComparison comparisonType)

static Int32 Compare(String strA, String strB, StringComparison comparisonType)
static Int32 Compare(string strA, string strB, Boolean ignoreCase, CultureInfo culture)
static Int32 Compare(String strA, String strB, CultureInfo culture, CompareOptions options)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
    StringComparison comparisonType)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
    CultureInfo culture, CompareOptions options)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
    Boolean ignoreCase, CultureInfo culture)
Boolean StartsWith(String value, StringComparison comparisonType)
Boolean StartsWith(String value,
    Boolean ignoreCase, CultureInfo culture)
```

```
Boolean EndsWith(String value, StringComparison comparisonType)
Boolean EndsWith(String value, Boolean ignoreCase, CultureInfo culture)
```

هنگام مرتب سازی، شما همیشه باید مقایسه‌های حساس به بزرگی و کوچکی حروف را انجام دهید. علت اینست که اگر دو رشته که فقط از لحاظ بزرگی و کوچکی حروف با هم تفاوت دارند، یکسان در نظر گرفته شوند، هربار که آن‌ها را مرتب می‌کنید، ممکن است متفاوت مرتب شوند، که این، کاربر را گیج می‌کند. آرگومان **ComparisonType** (در اغلب متدهای نشان داده شده در بالا) یکی از مقادیر تعریف شده توسط نوع شمارشی **StringComparison** است که طبق زیر تعریف شده است:

```
public enum StringComparison {
    CurrentCulture = 0,
    CurrentCultureIgnoreCase = 1,
    InvariantCulture = 2,
    InvariantCultureIgnoreCase = 3,
    Ordinal = 4,
    OrdinalIgnoreCase = 5
}
```

آرگومان **options** (در دو مورد از متدهای فوق) یکی از مقادیر تعریف شده توسط نوع شمارشی **CompareOptions** است:

[Flags]

```
public enum CompareOptions {
    None = 0,
    IgnoreCase = 1,
    IgnoreNonSpace = 2,
    IgnoreSymbols = 4,
    IgnoreKanaType = 8,
    IgnoreWidth = 0x00000010,
    Ordinal = 0x40000000,
    OrdinalIgnoreCase = 0x10000000,
    StringSort = 0x20000000
}
```

متدهایی که یک آرگومان **CompareOptions** می‌پذیرند، همچنین شما را مجبور می‌کنند صریحا یک فرهنگ را ارسال کنید. هنگام ارسال پرچم **Ordinal** یا **OrdinalIgnoreCase**، این متدهای **Compare**، فرهنگ تعیین شده را نادیده می‌گیرند.

بسیاری از برنامه‌ها از رشته برای هدف‌های برنامه‌نویسی مثل نام مسیرها، نام فایل‌ها، URL ها، کلیدها و مقادیر رجیستری، متغیرهای محیطی، رفلکشن، برچسب‌های XML، صفت‌های XML و غیره استفاده می‌کنند. اغلب این رشته‌ها به کاربر نشان داده نمی‌شوند و فقط درون برنامه استفاده می‌شوند. هنگام مقایسه چنین رشته‌هایی شما باید همیشه از **StringComparison.Ordinal** یا **StringComparison.OrdinalIgnoreCase** استفاده کنید. چون اطلاعات فرهنگ هنگام مقایسه در نظر گرفته نشده اند، این سریعترین راه برای انجام مقایسه‌ای است که نباید تحت تاثیر زبان قرار بگیرد. در سوی دیگر، وقتی شما می‌خواهید رشته‌ها را به روش صحیح زبانی (معمولا برای نمایش به یک کاربر نهایی) مقایسه کنید، شما باید از **StringComparison.CurrentCulture** یا **StringComparison.CurrentCultureIgnoreCase** استفاده کنید.

مهم برای اغلب مواقع، **StringComparison.InvariantCulture** و

StringComparison.InvariantCultureIgnoreCase نباید استفاده شود. اگرچه این مقادیر باعث می‌شوند مقایسه از لحاظ زبانی صحیح باشد، استفاده از آن‌ها برای مقایسه رشته‌های مربوط به برنامه‌نویسی، زمان بیشتری از یک مقایسه وصفی (**ordinal**) می‌گیرد. علاوه بر این، فرهنگ غیر متنوع (**invariant culture**)، خنثی از فرهنگ است که یعنی برای کار با رشته‌هایی که می‌خواهید به کاربر نشان دهید انتخاب اشتباهی می‌باشد.

مهم اگر شما می‌خواهید بزرگی و کوچکی کاراکترهای یک رشته را قبل از انجام یک مقایسه وصفی (**ordinal**) تغییر دهید، شما باید از متدهای **ToUpperInvariant** و **ToLowerInvariant** متعلق به **String**، استفاده کنید. هنگام نرمال کردن رشته‌ها، قویا توصیه می‌شود به جای **ToLowerInvariant** از **ToUpperInvariant** استفاده کنید چون مایکروسافت کد انجام مقایسه برای حروف بزرگ را بهینه کرده است. در واقع، FCL در درون خود، قبل از انجام مقایسه‌هایی که به بزرگی و کوچکی حروف حساس نیستند، رشته‌ها را تبدیل به حروف بزرگ می‌کند. ما از متدهای **ToUpperInvariant** و **ToLowerInvariant** استفاده می‌کنیم چون کلاس **String** متدهای **ToUpperOrdinal** و **ToLowerOrdinal** را ارائه نمی‌کند. ما از متدهای **ToUpper** و **ToLower** استفاده نمی‌کنیم چون این متدها به فرهنگ حساس هستند.

گاهی اوقات، وقتی شما رشته‌ها را به روش زبانی صحیح، مقایسه می‌کنید، می‌خواهید یک فرهنگ خاص را به جای فرهنگی که با ترد فراخوانی کننده همراه است، استفاده کنید. در این حالت، شما می‌توانید از سربارگذاری متدهای **StartsWith** و **EndsWith** و **Compare** که تمام آن‌ها آرگومان‌های **Boolean** و **CultureInfo** می‌گیرند و قبلا نشان داده شد، استفاده کنید.

مهم نوع **String** چندین سربارگذاری دیگر علاوه بر نسخه‌های نشان داده شده از متدهای **EndsWith**، **StartsWith**، **Equals** و **Compare** ارائه می‌کند. مایکروسافت توصیه می‌کند از استفاده از این دیگر نسخه‌ها (که در کتاب نشان داده نشده‌اند) خودداری شود. علاوه بر این، دیگر متدهای مقایسه متعلق به **String - CompareTo** (که توسط رابط **IComparable** نیاز است)، **CompareOrdinal** و عملگرهای **==** و **!=** -- نیز باید خودداری شوند. دلیل خودداری از این متدها و عملگرها اینست که فراخوانی کننده صریحا بیان نمی‌کند مقایسه رشته چگونه باید انجام پذیرد و شما از روی نام متد نمی‌توانید تعیین کنید مقایسه پیش فرض چیست. برای نمونه، طبق پیش فرض، **CompareTo** یک مقایسه حساس به فرهنگ انجام می‌دهد در حالیکه **Equals** یک مقایسه وصفی انجام می‌دهد. اگر شما صریحا بیان کنید می‌خواهید چه مقایسه رشته‌ای انجام دهید، کد شما خواناتر شده و نگهداری آن آسانتر است.

حال، بگذارید درباره‌ی مقایسه به روش صحیح زبانی صحبت کنیم. دات‌نت فریمورک از نوع **System.Globalization.CultureInfo** برای بیان یک جفت زبان/کشور (که طبق استاندارد RFC1766 توصیف شده) استفاده می‌کند. برای نمونه "en-US"، انگلیسی که در ایالات متحده نوشته می‌شود را شناسایی می‌کند، "en-AU" انگلیسی که در استرالیا نوشته می‌شود را شناسایی می‌کند و "de-DE" آلمانی که در آلمان نوشته می‌شود را نشان می‌دهد. در CLR، هر ترد دو ویژگی همراه با خودش دارد. هر یک از این ویژگی‌ها به یک شی **CultureInfo** اشاره دارد: این دو ویژگی عبارتند از:

- **CurrentUICulture** این ویژگی برای بدست آوردن منابعی که به کاربر نهایی نشان داده می‌شود به کار می‌رود. آن اغلب برای برنامه‌هایی با رابط گرافیکی یا فرم‌های وب کاربرد دارد چون زبانی که هنگام نمایش عناصر رابط گرافیکی مثل برچسب‌ها و دکمه‌ها استفاده می‌شود را تعیین می‌کند. طبق پیش فرض وقتی شما یک ترد می‌سازید، این ویژگی ترد با یک شی **CultureInfo** تنظیم می‌شود که با استفاده از تابع **GetUserDefaultUILanguage Win32**، زبان نسخه‌ی ویندوزی که برنامه بر روی آن در حال اجراست را شناسایی می‌کند. اگر شما نسخه‌ی رابط کاربری چند زبانه^{۵۳} (MUI) ویندوز را در حال اجرا دارید، شما می‌توانید این را از طریق تنظیمات پنجره‌ی "Regional and Language Options" در **Control Panel** تنظیم کنید. در یک نسخه غیر MUI، زبان توسط نسخه‌ی سیستم عامل نصب شده (پک زبان نصب شده) تعیین می‌شود و زبان قابل تغییر نیست.
- **CurrentCulture** این ویژگی برای هر چیزی که **CurrentUICulture** برایش استفاده نشود، استفاده می‌شود شامل فرمت اعداد و تاریخ، بزرگی و کوچکی رشته‌ها و مقایسه رشته‌ها. هنگام فرمت کردن، هر دو بخش زبان و کشور از شی **CultureInfo** استفاده می‌شوند. طبق پیش فرض، وقتی شما یک ترد می‌سازید، این ویژگی ترد به یک شی **CultureInfo** تنظیم می‌شود که مقدارش با فراخوانی متد **GetUserDefaultLCID Win32** که مقدار آن نیز در اپلت "Regional and Language" در **Control Panel** تنظیم می‌شود، تعیین می‌گردد.

در بسیاری از کامپیوترها، ویژگی‌های **CurrentUICulture** و **CurrentCulture** از یک ترد، به یک شی یکسان **CultureInfo** تنظیم می‌شوند که یعنی هر دوی آن‌ها از اطلاعات زبان/کشور یکسانی استفاده می‌کنند. هر چند، می‌توانند متفاوت از هم تنظیم شوند. برای نمونه، یک برنامه که در ایالات متحده اجرا می‌شود می‌تواند از زبان اسپانیایی برای تمام آیتم‌های منو و عناصر گرافیکی خود بهره‌بردار در حالیکه به درستی تمام نرخ‌های پول و فرمت تاریخ‌ها را برای ایالات متحده نمایش می‌دهد. برای این کار، ویژگی **CurrentUICulture** از ترد، باید به یک شی **CultureInfo** که زبانش "es" (برای اسپانیایی) است تنظیم گردد در حالیکه ویژگی **CurrentCulture** از ترد باید به یک شی **CultureInfo** که با یک جفت زبان/کشور "en-US" مقداردهی شده، تنظیم گردد. در درون، یک شی **CultureInfo** دارای یک فیلد است که به یک شی **System.Globalization.CompareInfo** اشاره دارد، که جدول اطلاعات مرتب سازی کاراکتر، متعلق به فرهنگ که توسط استاندارد یونیکد تعریف شده است را کپسوله می‌کند. کد زیر تفاوت بین یک مقایسه وصفی (ordinal) و یک مقایسه که از فرهنگ مطلع است را نشان می‌دهد:

```
using System;
using System.Globalization;

public static class Program {
    public static void Main() {
        String s1 = "Strasse";
        String s2 = "Straße";
```

⁵³ Multilingual User Interface

```

Boolean eq;

// CompareOrdinal returns nonzero.
eq = String.Compare(s1, s2, StringComparison.Ordinal) == 0;
Console.WriteLine("Ordinal comparison: '{0}' {2} '{1}'", s1, s2,
eq ? "==" : "!=");

// Compare Strings appropriately for people
// who speak German (de) in Germany (DE)
CultureInfo ci = new CultureInfo("de-DE");

// Compare returns zero.
eq = String.Compare(s1, s2, true, ci) == 0;
Console.WriteLine("Cultural comparison: '{0}' {2} '{1}'", s1, s2,
eq ? "==" : "!=");
}
}

```

ساختن و اجرای این کد، خروجی زیر را به دنبال دارد:

```

Ordinal comparison: 'Strasse' != 'Straße'
Cultural comparison: 'Strasse' == 'Straße'

```

نکته وقتی متد **Compare** یک مقایسه وصفی (ordinal) انجام نمی‌دهد، آن، بسط کاراکتر character expansion را انجام می‌دهد. بسط کاراکتر، هنگامی است که یک کاراکتر بدون توجه به فرهنگ به چندین کاراکتر بسط داده می‌شود. در مورد بالا، کاراکتر Eszet آلمانی 'ß' همیشه به 'ss' بسط داده می‌شود. به طریق مشابه، کاراکتر تحت الفظی 'Æ' همیشه به 'AE' بسط داده می‌شود. بنابراین در کد مثال، دومین فراخوانی به **Compare** بدون توجه به فرهنگی که به آن ارسال کردم، 0 برمی‌گرداند.

در شرایطی نادر، شاید شما نیاز به کنترل بیشتری هنگام مقایسه رشته‌ها برای برابری یا مرتب سازی داشته باشید. این ممکن است هنگام مقایسه رشته‌هایی که از کاراکترهای ژاپنی تشکیل شده‌اند نیاز شود. این کنترل اضافی از طریق ویژگی **CompareInfo** از شی **CultureInfo** قابل دسترسی است. همانگونه که قبلا اشاره کردم، یک شی **CompareInfo**، جدول مقایسه کاراکترهای یک فرهنگ را کپسوله می‌کند و تنها یک شی **CompareInfo** به ازای هر فرهنگ وجود دارد.

وقتی شما متد **Compare** از **String** را فراخوانی می‌کنید، اگر فراخوانی کننده، یک فرهنگ تعیین کند، فرهنگ تعیین شده استفاده می‌گردد و اگر فرهنگی تعیین نشود، مقدار درون **CurrentCulture** از ترد فراخوانی کننده استفاده می‌شود. در درون، متد **Compare**، ارجاع به شی **CompareInfo** را برای فرهنگ مناسب بدست آورده و متد **Compare** از شی **CompareInfo** را با ارسال انتخاب‌های مناسب (مثل بزرگی و کوچکی حروف) فراخوانی می‌کند. طبیعتا، شما می‌توانستید متد **Compare** از یک شی خاص **CompareInfo** را خودتان فراخوانی کنید اگر نیاز به کنترل بیشتری داشتید.

متد **Compare** از نوع **CompareInfo** به عنوان یک پارامتر، یک مقدار از نوع شمارشی **CompareOptions** (که قبلا نشان داده شد) را می‌گیرد. شما می‌توانید این پرچم‌های بیتی را با هم OR کنید تا هنگام مقایسه‌ی رشته‌ها به کنترل بسیار بیشتری دست یابید. برای توضیح کامل این نمادها به مستندات دات‌نت فریمورک مراجعه کنید.

کد زیر نشان می‌دهد که فرهنگ چقدر در مرتب سازی رشته‌ها اهمیت دارد و راه‌های متنوع مقایسه رشته‌ها را بیان می‌کند:

```

using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;

```

```

using System.Threading;

public sealed class Program {
    public static void Main() {
        String output = String.Empty;
        String[] symbol = new String[] { "<", "=", ">" };
        Int32 x;
        CultureInfo ci;

        // The code below demonstrates how strings compare
        // differently for different cultures.
        String s1 = "coté";
        String s2 = "côte";

        // Sorting strings for French in France.
        ci = new CultureInfo("fr-FR");
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;

        // Sorting strings for Japanese in Japan.
        ci = new CultureInfo("ja-JP");
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;

        // Sorting strings for the thread's culture
        ci = Thread.CurrentThread.CurrentCulture;
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine + Environment.NewLine;

        // The code below demonstrates how to use CompareInfo.Compare's
        // advanced options with 2 Japanese strings. One string represents
        // the word "shinkansen" (the name for the Japanese high-speed
        // train) in hiragana (one subtype of Japanese writing), and the
        // other represents the same word in katakana (another subtype of
        // Japanese writing).
        s1 = "しんかんせん"; // ("\u3057\u3093\u304B\u3093\u305b\u3093")
        s2 = "シンカンセン"; // ("\u30b7\u30f3\u30ab\u30f3\u30bb\u30f3")

        // Here is the result of a default comparison
        ci = new CultureInfo("ja-JP");
        x = Math.Sign(String.Compare(s1, s2, true, ci));
    }
}

```

```

output += String.Format("Simple {0} Compare: {1} {3} {2}",
ci.Name, s1, s2, symbol[x + 1]);
output += Environment.NewLine;

// Here is the result of a comparison that ignores
// kana type (a type of Japanese writing)
CompareInfo compareInfo = CompareInfo.GetCompareInfo("ja-JP");
x = Math.Sign(compareInfo.Compare(s1, s2, CompareOptions.IgnoreKanaType));
output += String.Format("Advanced {0} Compare: {1} {3} {2}",
ci.Name, s1, s2, symbol[x + 1]);
MessageBox.Show(output, "Comparing Strings For Sorting");
}
}

```

نکته این فایل سورس کد نمی‌تواند در ANSI ذخیره شود مگر اینکه کاراکترهای ژاپنی آن از بین روند. برای ذخیره این فایل در ویژوال استودیو، پنجره‌ی **Save File As** را باز کرده، فلش سمت پایین که بخشی از دکمه **Save** است را کلیک کرده و **Save With Encoding** را انتخاب کنید. من **"Unicode (UTF-8 with signature) – Codepage 65001"** را انتخاب کردم. کامپایلر سی‌شارپ مایکروسافت می‌تواند سورس کد فایل را با این صفحه کد، با موفقیت تجزیه و ذخیره کند.

ساخت و اجرای این برنامه، خروجی نمایش داده شده در شکل ۱-۱۴ را تولید می‌کند.



شکل ۱-۱۴ نتایج مرتب سازی رشته‌ها

علاوه بر **Compare**، کلاس **CompareInfo** متدهای **IndexOf**، **LastIndexOf** و **IsPrefix** را ارائه می‌کند. چون تمام این متدها، سربرگ‌گذاری‌هایی ارائه می‌کنند که یک مقدار شمارشی **CompareOptions** به عنوان یک پارامتر می‌گیرند، آن‌ها کنترل بیشتری نسبت به متدهای **Compare**، **IndexOf**، **LastIndexOf**، **StartsWith** و **EndsWith** تعریف شده توسط کلاس **String**، به شما می‌دهند. همچنین، شما باید بدانید که FCL یک کلاس **System.StringComparer** دارد که شما می‌توانید برای انجام مقایسه رشته‌ها از آن استفاده کنید. این کلاس هنگامی که می‌خواهید یک گونه مقایسه را چندین بار برای رشته‌های مختلف تکرار کنید کاربرد دارد.

وارد کردن رشته (String Interning)

همانگونه که در بخش قبل گفتم، بررسی رشته‌ها برای برابری، یک عمل رایج در بسیاری از برنامه‌هاست – این عمل می‌تواند به عملکرد برنامه بسیار صدمه بزند. هنگام انجام یک بررسی برابری وصفی (ordinal)، CLR سریعاً تست می‌کند که آیا دو رشته تعداد یکسانی کاراکتر دارند. اگر ندارند، رشته‌ها قطعاً برابر نیستند، و اگر دارند، رشته‌ها ممکن است برابر باشند و CLR باید هر کاراکتر را برای اطمینان حاصل کردن، مقایسه کند. هنگام انجام یک مقایسه که از فرهنگ مطلع است، CLR باید همیشه همه‌ی تک‌تک کاراکترها را مقایسه کند چون رشته‌هایی با طول متفاوت ممکن است یکسان در نظر گرفته شوند. به علاوه، اگر شما چندین نمونه از یک رشته در حافظه دارید، شما حافظه را هدر داده‌اید چون رشته‌ها تغییر ناپذیرند. اگر تنها از یک رشته در حافظه استفاده کنید و تمام متغیرهایی که نیاز دارند به رشته اشاره کنند، فقط به این تک شی رشته اشاره کنند، آنگاه شما از حافظه بسیار کارا تر استفاده نموده‌اید.

اگر برنامه شما مکرراً رشته‌ها را برای برابری با مقایسه‌های وصفی حساس به بزرگی و کوچکی حروف، مقایسه می‌کند یا اگر شما انتظار دارید تعداد زیادی شی رشته با مقدار یکسان داشته باشید شما می‌توانید با بهره بردن از مکانیزم واردن کردن رشته `string interning` در CLR، به صورت قابل ملاحظه‌ای عملکرد را بهبود بخشید. وقتی CLR مقداردهی اولیه می‌شود، یک جدول هش داخلی می‌سازد که در آن کلیدها رشته‌ها هستند و مقادیر، اشاره‌گرهایی به اشیاء `String` در هیپ مدیریت شده می‌باشند. در ابتدا جدول (البته) خالی است. کلاس `String` دو متد ارائه می‌کند که به شما اجازه می‌دهد به این جدول هش داخلی دسترسی پیدا کنید:

```
public static String Intern(String str);
public static String IsInterned(String str);
```

متد اول، `Intern`، یک `String` می‌گیرد، یک کد هش برای آن بدست می‌آورد، و جدول هش داخلی را برای یافتن تطابق جستجو می‌کند. اگر یک رشته-ی معادل، از قبل وجود داشت، یک اشاره‌گر به شی `String` موجود برگردانده می‌شود. اگر یک رشته-ی معادل وجود نداشت، یک کپی از رشته ساخته می‌شود، کپی به جدول هش داخلی اضافه می‌گردد و اشاره‌گر به این کپی برگردانده می‌شود. اگر برنامه، دیگر اشاره‌گری به شی `String` اصلی نداشته باشد، جمع آوری کننده زباله قادر است حافظه آن رشته را آزاد کند. دقت کنید که جمع آوری کننده زباله نمی‌تواند رشته‌هایی که جدول هش داخلی به آن‌ها اشاره دارد را آزاد کند، چون جدول هش، ارجاعی به آن اشیاء `String` را نگهداری می‌کند. اشیاء `String` که توسط جدول هش داخلی ارجاع می‌شوند تا زمانی که `AppDomain` تخلیه نشده یا پردازش از بین نرفته باشد، نمی‌توانند آزاد شوند.

همانگونه که متد `Intern` عمل می‌کند، متد `IsInterned` یک `String` می‌گیرد و آن را در جدول هش داخلی جستجو می‌کند. اگر یک رشته معادل در جدول یافت شد، `IsInterned` یک اشاره‌گر به شی رشته وارد شده (`interned`) را برمی‌گرداند. اگر یک رشته-ی معادل یافت نشد، `IsInterned` `null` برمی‌گرداند؛ رشته را به جدول هش اضافه نمی‌کند.

طبق پیش فرض، وقتی یک اسمبلی بارگذاری می‌شود، CLR تمام رشته‌های تحت‌الفظی توصیف شده در متادیتای اسمبلی را وارد می‌کند (`intern`). مایکروسافت یاد گرفته است که این کار به خاطر جستجوی اضافی جدول هش به شدت عملکرد را پایین می‌آورد، بنابراین، اکنون می‌توان این "ویژگی" را غیرفعال کرد. اگر یک اسمبلی با یک `System.Runtime.CompilerServices.CompilationRelaxationsAttribute` و تعیین مقدار پرچم `System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning` علامت زده شود، CLR ممکن است بر طبق مشخصات ECMA تصمیم بگیرد رشته‌های تعریف شده در متادیتای اسمبلی را وارد نکند. توجه کنید که، در جهت بهبود عملکرد برنامه شما، کامپایلر سی‌شارپ همیشه این صفت/پرچم را هر بار که یک اسمبلی را کامپایل می‌کند، تعیین می‌کند.

حتی اگر یک اسمبلی این صفت/پرچم را تعیین کند، CLR ممکن است انتخاب کند که رشته‌ها را وارد کند، اما شما روی این نباید حساب باز کنید. در حقیقت، شما هرگز نباید کدی بنویسید که بر رشته‌های وارد شده متکی باشد مگر اینکه خودتان کدی نوشته باشید که صریحاً متد `Intern` از `String` را فراخوانی کند. کد زیر وارد کردن رشته را نشان می‌دهد:

```
String s1 = "Hello";
String s2 = "Hello";
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // Should be 'False'
```

```
s1 = String.Intern(s1);
s2 = String.Intern(s2);
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 'True'
```

در اولین فراخوانی به متد `s1.ReferenceEquals` به یک شی رشته `"Hello"` در هیپ اشاره دارد و `s2` به شی رشته `"Hello"` متفاوتی در هیپ اشاره می‌کند. چون ارجاع‌ها متفاوتند، `False` باید نمایش داده شود. اما اگر شما این را روی نسخه 4.0 از CLR اجرا کنید، شما می‌بینید `True` نمایش داده می‌شود. علت اینست که این نسخه از CLR تصمیم گرفته است صفت/پرچم تعیین شده توسط کامپایلر سی‌شارپ را نادیده بگیرد و CLR وقتی اسمبلی درون `AppDomain` بارگذاری می‌شود رشته `"Hello"` را در هیپ را وارد می‌کند. این یعنی `s1` و `s2` به یک تک رشته `"Hello"` در هیپ اشاره می‌کنند. به هر حال، همانگونه که قبلاً گفته شد، شما هرگز نباید کدی بنویسید که به این رفتار تکیه کند چون یک نسخه آتی از CLR ممکن است به صفت/پرچم احترام گذاشته و رشته `"Hello"` را وارد نکند. در واقع، نسخه 4.0 از CLR هنگامیکه کد اسمبلی با ابزار `NGen.exe` کامپایل شود، به این صفت/پرچم احترام می‌گذارد.

قبل از دومین فراخوانی به متد **ReferenceEquals**، رشته **“Hello”** صریحا وارد شده است و **s1** اکنون به یک **“Hello”** ی وارد شده اشاره می-کند. پس با فراخوانی مجدد **s2.Intern** به همان رشته **“Hello”** که **s1** اشاره دارد، اشاره می-کند. اکنون وقتی **ReferenceEquals** برای بار دوم فراخوانی می-شود، ما مطمئنیم یک نتیجه **True** دریافت می-کنیم بدون توجه به اینکه آیا اسمبلی با صفت/پرچم کامپایل شده است یا نه.

پس اکنون، بگذارید به مثالی نگاه کنیم تا ببینیم چگونه می-توانید از وارد کردن رشته‌ها برای بهبود کارایی و کاهش حافظه مصرفی استفاده کنید. متد **NumTimesWordAppearEqual** در زیر دو آرگومان می-گیرد، یک کلمه و یک آرایه از رشته‌ها که هر عنصر آرایه به یک تک کلمه اشاره دارد. این متد سپس تعیین می-کند چند بار کلمه تعیین شده در لیست کلمات ظاهر شده است و این عدد را برمی-گرداند:

```
private static Int32 NumTimesWordAppearsEquals(String word, String[] wordlist) {
    Int32 count = 0;
    for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
        if (word.Equals(wordlist[wordnum], StringComparison.Ordinal))
            count++;
    }
    return count;
}
```

همانگونه که می-بینید، این متد، متد **Equals** از **String** را فراخوانی می-کند که در درون، تک-تک کاراکترهای رشته را مقایسه کرده و بررسی می-کند که تمام کاراکترها برابر باشند. این مقایسه می-تواند کند باشد. به علاوه، آرایه‌ی لیست کلمات ممکن است چندین عنصر داشته باشد که به چندین شی **String** حاوی مجموعه یکسانی از کاراکترها اشاره کنند. این یعنی چندین رشته‌ی یکسان ممکن است در هیپ موجود باشد که از دست جمع آوری کننده نجات یابند. حال، به یک نسخه از این متد که با بهره‌گیری از وارد کردن رشته‌ها، کار می-کند نگاه کنید:

```
private static Int32 NumTimesWordAppearsIntern(String word, String[] wordlist) {
    // This method assumes that all entries in wordlist refer to interned strings.
    word = String.Intern(word);
    Int32 count = 0;
    for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
        if (Object.ReferenceEquals(word, wordlist[wordnum]))
            count++;
    }
    return count;
}
```

این متد کلمه را وارد می-کند و فرض می-کند لیست کلمات حاوی ارجاعاتی به رشته وارد شده باشد. نخست اینکه، این نسخه در صورت ظهور چند باره‌ی یک کلمه در لیست کلمات، ممکن است حافظه را صرفه جویی کند، چون در این نسخه، **wordlist** حاوی ارجاعاتی به یک تک شی **String** در هیپ است. دوم اینکه، این نسخه سریعتر است چون تشخیص کلمه تعیین شده در آرایه فقط یک عملیات مقایسه اشاره‌گرهاست.

اگرچه متد **NumTimesWordAppearIntern** سریعتر از متد **NumTimesWordAppearEquals** است، عملکرد کلی برنامه ممکن است هنگام استفاده از متد **NumTimesWordAppearIntern** کندتر باشد چون مدت زمانی صرف وارد کردن تمام رشته‌هایی که به آرایه **wordlist** افزوده شده اند، می-گردد. (کد نشان داده نشده است). متد **NumTimesWordAppearIntern** عملکرد و بهبود حافظه‌اش را واقعا وقتی نشان می-دهد که برنامه نیاز داشته باشد چندین بار متد را با همان لیست کلمات فراخوانی کند. نکته این بحث این است که روش وارد کردن رشته‌ها مفید است اما باید با دقت و مراقبت استفاده شود. در حقیقت، این علتی است که کامپایلر سی‌شارپ بیان می-کند که نمی‌خواهد وارد کردن رشته‌ها فعال باشد.

ادغام رشته‌ها

هنگام کامپایل کدتان، کامپایلر شما باید هر رشته تحت‌اللفظی را پردازش کرده و رشته را در متادیتای ماژول قرار دهد. اگر رشته تحت‌اللفظی یکسانی، چندین بار در کد شما ظاهر شود، تولید تمام این رشته‌ها درون متادیتا، اندازه فایل تولیدی را بزرگ می-کند.

برای حذف این اثر، بسیاری از کامپایلرها (شامل کامپایلر سی‌شارپ) رشته تحت‌اللفظی را درون متادیتای ماژول فقط یکبار می‌نویسند. تمام کدهایی که رشته را ارجاع می‌کنند تغییر داده می‌شوند تا به یک رشته در متادیتا اشاره کنند، این قابلیت یک کامپایلر در ادغام چندین رخداد از یک تک رشته به یک تک نمونه،

می‌تواند به صورت قابل ملاحظه‌ای اندازه‌ی یک ماژول را کاهش دهد. چنین پردازشی، چیز جدیدی نیست – کامپایلرهای C/C++ برای سال‌ها چنین کاری را می‌کردند (کامپایلر C/C++ مایکروسافت این کار را ادغام رشته‌ها string pooling می‌نامند). بنابراین، ادغام رشته‌ها راه دیگری برای بهبود رشته‌ها و بخشی دیگر از دانشی است که باید در لیست خود داشته باشید.

بررسی کاراکترهای یک رشته و عناصر متنی

اگرچه مقایسه رشته‌ها برای مرتب سازی آن‌ها یا تعیین برابری، مفید است، گاهی شما نیاز دارید کاراکترهای درون یک رشته را بررسی کنید. نوع **String** چندین ویژگی و متد برای کمک به شما در این کار ارائه می‌کند، شامل **Length**، **Chars** (یک ایندکسر در سی شارپ)، **GetEnumerator**، **ToArray**، **Contains**، **IndexOf**، **LastIndexOf**، **IndexOfAny** و **LastIndexOfAny**.

در واقعیت، یک **System.Char** یک مقدار کد ۱۶ بیتی یونیکد را نمایش می‌دهد که لزوماً با یک کاراکتر خلاصه یونیکد برابر نیست. برای نمونه، برخی کاراکترهای خلاصه یونیکد ترکیبی از دو مقدار کد هستند. وقتی کاراکترهای **U+0625** (حرف الف عربی با همزه زیرین) و **U+0650** (کسره عربی) ترکیب می‌شوند یک تک کاراکتر خلاصه یا عنصر متنی **text element** می‌سازند.

به علاوه، برخی عناصر متنی یونیکد به بیش از یک مقدار ۱۶ بیتی برای نمایش خود نیاز دارند. این عناصر متنی با استفاده از دو مقدار کد ۱۶ بیتی نمایش داده می‌شوند. اولین مقدار کد، جانشین بالا **high surrogate** و دومین مقدار کد، جانشین پایین **low surrogate** نامیده می‌شوند. جانشینان بالا، مقداری بین **U+D800** و **U+DBFF** و جانشینان پایین مقداری بین **U+DC00** و **U+DFFF** دارند. استفاده از جانشینان به یونیکد اجازه می‌دهد بیش از یک میلیون کاراکتر مختلف را نمایش دهند.

جانشینان به ندرت در ایالات متحده و اروپا استفاده می‌شوند اما اغلب در آسیای شرقی کاربرد دارند. برای کار کردن صحیح با عناصر متنی، شما باید از نوع **System.Globalization.StringInfo** استفاده کنید. ساده ترین راه استفاده از این نوع، ساخت یک نمونه از آن و ارسال یک رشته به سازنده‌اش است. سپس شما می‌توانید بوسیله ویژگی **LengthInTextElements** از **StringInfo**، تعداد عناصر متنی را در رشته بیابید. بعد از آن می‌توانید متد **SubstringByTextElements** از **StringInfo** را برای استخراج عنصر متنی یا تعدادی از عناصر متنی پشت سر هم استفاده کنید.

به علاوه، کلاس **StringInfo** یک متد استاتیک **GetEnumerator** ارائه می‌کند که یک شیء **System.Globalization.TextElementEnumerator** بدست آورده که اجازه می‌دهد تمام کاراکترهای یونیکد خلاصه در درون متن را بشمارید. سرانجام، شما می‌توانید متد استاتیک **ParseCombiningCharacters** را از **StringInfo** فراخوانی کنید تا یک آرایه از مقادیر **Int32** بدست آورید. طول آرایه تعیین می‌کند چند عنصر متنی درون رشته وجود دارد. هر عنصر آرایه یک اندیس درون رشته که اولین مقدار کد برای یک عنصر متنی جدید در آنجا یافت می‌شود را شناسایی می‌کند. کد زیر روش‌های مختلف استفاده از کلاس **StringInfo** برای دستکاری عناصر متنی یک رشته را نشان می‌دهد:

```
using System;
using System.Text;
using System.Globalization;
using System.Windows.Forms;

public sealed class Program {
    public static void Main() {
        // The string below contains combining characters
        String s = "a\u0304\u0308bc\u0327";
        SubstringByTextElements(s);
        EnumTextElements(s);
        EnumTextElementIndexes(s);
    }

    private static void SubstringByTextElements(String s) {
        String output = String.Empty;
        StringInfo si = new StringInfo(s);
        for (Int32 element = 0; element < si.LengthInTextElements; element++) {
```

```

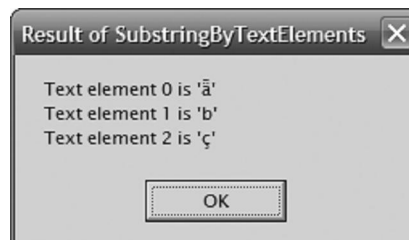
output += String.Format(
    "Text element {0} is '{1}'{2}",
    element, si.SubstringByTextElements(element, 1),
    Environment.NewLine);
}
MessageBox.Show(output, "Result of SubstringByTextElements");
}

private static void EnumTextElements(String s) {
    String output = String.Empty;
    TextElementEnumerator charEnum =
        StringInfo.GetTextElementEnumerator(s);
    while (charEnum.MoveNext()) {
        output += String.Format(
            "Character at index {0} is '{1}'{2}",
            charEnum.ElementIndex, charEnum.GetTextElement(),
            Environment.NewLine);
    }
    MessageBox.Show(output, "Result of GetTextElementEnumerator");
}

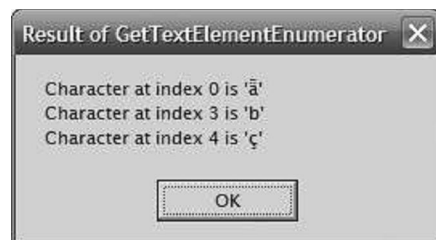
private static void EnumTextElementIndexes(String s) {
    String output = String.Empty;
    Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);
    for (Int32 i = 0; i < textElemIndex.Length; i++) {
        output += String.Format(
            "Character {0} starts at index {1}{2}",
            i, textElemIndex[i], Environment.NewLine);
    }
    MessageBox.Show(output, "Result of ParseCombiningCharacters");
}
}
}

```

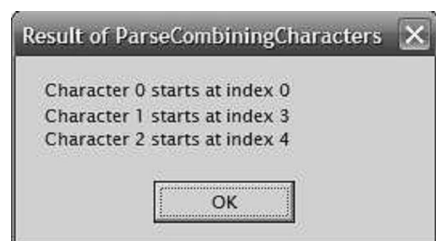
ساخت و اجرای این کد، جعبه پیام نشان داده شده در شکل‌های ۱۴-۲، ۱۴-۳، ۱۴-۴ را تولید می‌کند:



شکل ۱۴-۲ نتیجه‌ی SubstringByTextElements



شکل ۳-۱۴ نتیجه‌ی GetEnumerator



شکل ۴-۱۴ نتیجه‌ی ParseCombiningCharacters

دیگر عملیات های رشته

نوع **String** همچنین متدهایی ارائه می‌کند که به شما اجازه می‌دهند یک رشته یا بخش‌هایی از آن را کپی کنید. جدول ۱-۱۴ این متدها را لیست می‌کند. جدول ۱-۱۴ متدهای عملیات کپی برای رشته‌ها

عضو	نوع متد	توضیح
Clone	نمونه	یک اشاره‌گر به همان شی (this) برمی‌گرداند. این درست است چون اشیاء رشته، تغییر ناپذیرند. این متد، رابط ICloneable از String را پیاده‌سازی می‌کند.
Copy	استاتیک	یک کپی جدید از رشته تعیین شده را برمی‌گرداند. این متد به ندرت استفاده می‌شود و برای کمک به برنامه‌هایی که از رشته‌ها به عنوان نشانه (token) استفاده می‌کنند وجود دارد. به صورت عادی، رشته‌هایی با مجموعه کاراکترهای یکسان در یک تک رشته وارد می‌شوند. این متد یک شی رشته جدید می‌سازد پس ارجاع‌ها (اشاره-گرها) متفاوتند اگرچه رشته‌ها، کاراکترهای یکسانی دارند.
CopyTo	نمونه	بخشی از کاراکترهای رشته را در یک آرایه از کاراکترها کپی می‌کند.
Substring	نمونه	یک رشته جدید برمی‌گرداند که بخشی از رشته اصلی را نمایش می‌دهد.
ToString	نمونه	یک ارجاع به همان شی (this) برمی‌گرداند.

علاوه بر این متدها، **String** متدهای استاتیک و نمونه‌ی زیادی ارائه می‌کند که یک رشته را دستکاری می‌کنند مثل **PadLeft**، **Remove**، **Insert**، **Format**، **Concat**، **Trim**، **ToUpper**، **ToLower**، **Join**، **Split**، **Replace**، **Split**، **Join**، **ToLower**، **ToUpper**، **Trim**، **Concat**، **Format** و غیره. مجدداً، نکته مهم برای یادآوری درباره‌ی این متدها اینست که آن‌ها اشیاء رشته جدید را برمی‌گردانند، چون رشته‌ها تغییر ناپذیرند، وقتی ساخته می‌شوند، (توسط کد امن) قابل تغییر نیستند.

ساخت یک رشته به صورت کارا

چون نوع **String** یک رشته تغییر ناپذیر را نمایش می‌دهد، FCL نوع دیگری فراهم کرده است، **System.Text.StringBuilder**، که به شما اجازه می‌دهد عملیات‌های پویا روی رشته و کاراکترها را به صورت کارا انجام دهید تا یک **String** بسازید. به **StringBuilder** به عنوان یک سازنده تجملی برای ساخت یک **String** که می‌تواند توسط مابقی فریمورک استفاده شود، نگاه کنید. به طور کلی، شما باید متدهایی طراحی کنید که پارامترهای **String** بگیرند نه پارامترهای **StringBuilder**.

منطقاً، یک شی **StringBuilder** شامل یک فیلد است که به آرایه‌ای از ساختارهای **Char** اشاره دارد. اعضای **StringBuilder** به شما اجازه‌ی دستکاری این آرایه را می‌دهند، و همچنین به صورت موثر رشته را کوچک کرده یا کاراکترهای رشته را تغییر دهید. اگر شما رشته را بزرگ کنید تا از آرایه تخصیص یافته کاراکترها عبور کند، **StringBuilder** به صورت خودکار، یک آرایه جدید و بزرگتر تخصیص می‌دهد، کاراکترها را کپی می‌کند و از این به بعد از آرایه جدید استفاده می‌کند. آرایه قبلی جمع آوری می‌شود. وقتی کار شما برای ساخت رشته با شی **StringBuilder** تمام شد، آرایه کاراکترهای **StringBuilder** را به یک **String** با فراخوانی متد **ToString** از **StringBuilder** "تبدیل" کنید. این کار یک شی جدید **String** در هیپ می‌سازد که حاوی رشته‌ای است که در زمان فراخوانی متد **ToString** در **StringBuilder** وجود داشت. در این لحظه، شما می‌توانید به دستکاری رشته در **StringBuilder** ادامه دهید و بعد برای تبدیل آن به شی **String** دیگری، مجدداً **ToString** را فراخوانی کنید.

ساخت یک شی **StringBuilder**

برخلاف کلاس **String**، CLR اطلاعات خاصی درباره‌ی کلاس **StringBuilder** ندارد. به علاوه، اغلب زبان‌ها (شامل سی‌شارپ) کلاس **StringBuilder** را یک نوع اصلی تلقی نمی‌کنند. شما یک شی **StringBuilder** را همانند ساخت دیگر نوع‌های غیر اصلی، می‌سازید

```
StringBuilder sb = new StringBuilder();
```

نوع **StringBuilder** تعداد زیادی سازنده ارائه می‌کند. کار هر یک از این سازنده‌ها تخصیص و مقداردهی اولیه وضعیت است که توسط هر شی **StringBuilder** نگهداری می‌شود:

- **ماکزیمم گنجایش** یک مقدار **Int32** که ماکزیمم تعداد کاراکترهایی که می‌تواند در رشته جای بگیرد را تعیین می‌کند. پیش فرض **In32.MaxValue** (تقریباً دو میلیارد) است. تغییر این مقدار غیر معمول است. هرچند، ممکن است شما یک ماکزیمم گنجایش کوچکتر تعیین کنید تا مطمئن شوید هرگز رشته‌ای نمی‌سازید که از یک اندازه خاص بلندتر شود. وقتی ساخته می‌شود، مقدار ماکزیمم گنجایش یک **StringBuilder** قابل تغییر نیست.
- **گنجایش** یک مقدار **Int32** که اندازه‌ی آرایه کاراکترهایی که توسط **StringBuilder** نگهداری می‌شود را بیان می‌کند. پیش فرض، ۱۶ است. اگر نظری درباره‌ی اینکه چه تعداد کاراکتر در **StringBuilder** قرار خواهید داد، دارید شما باید از این عدد برای تنظیم گنجایش، هنگام ساخت شی **StringBuilder** استفاده کنید.
- وقتی کاراکترهایی به آرایه کاراکترها می‌افزایید، **StringBuilder** تشخیص می‌دهد آیا آرایه سعی دارد بیش از گنجایش آرایه بزرگ شود. اگر اینگونه است، **StringBuilder** به صورت خودکار فیلد گنجایش را دو برابر می‌کند، یک آرایه جدید (به اندازه گنجایش جدید) تخصیص می‌دهد و کاراکترها را از آرایه اصلی به آرایه جدید کپی می‌کند. آرایه اصلی در آینده جمع آوری خواهد شد. بزرگ کردن پویای آرایه به عملکرد ضربه می‌زند، با تعیین یک گنجایش اولیه مناسب از این اتفاق جلوگیری کنید.
- **آرایه کاراکترها** یک آرایه از ساختارهای **Char** که مجموعه‌ای از کاراکترهای درون "رشته" را نگهداری می‌کند. تعداد کاراکترهای آرایه همیشه کوچکتر یا مساوی مقادیر گنجایش و ماکزیمم گنجایش است. شما می‌توانید از ویژگی **Length** از **StringBuilder** استفاده کنید. تا تعداد کاراکترهای استفاده شده در آرایه را بیابید. **Length** همیشه کوچکتر یا مساوی تعداد گنجایش **StringBuilder** است، هنگام ساخت یک **StringBuilder**، شما می‌توانید یک رشته را برای مقداردهی اولیه آرایه کاراکترها ارسال کنید. اگر شما یک رشته تعیین نکنید، آرایه در ابتدا حاوی هیچ کاراکتری نیست - یعنی، ویژگی **Length**، ۰ برمی‌گرداند.

اعضای **StringBuilder**

برخلاف یک **String**، یک **StringBuilder** یک رشته تغییر ناپذیر را نمایش می‌دهد. این یعنی اغلب اعضای **StringBuilder** محتویات آرایه کاراکترها را تغییر می‌دهند و باعث نمی‌شوند اشیاء جدیدی در هیپ تخصیص یابد. یک **StringBuilder** در دو حالت، یک شی جدید تخصیص می‌دهد:

- یک رشته به صورت پویا بسازید که طولش بلندتر از گنجایشی باشد که تعیین کرده اید.
- شما متد **ToString** از **StringBuilder** را فراخوانی کنید.

جدول ۱۴-۲ اعضای **StringBuilder** را نشان می‌دهد.

جدول ۲-۱۴ اعضای **StringBuilder**

عضو	نوع عضو	توضیح
MaxCapacity	ویژگی فقط-خواندنی	بیشترین تعداد کاراکترهایی که می‌تواند در رشته جای بگیرد را برمی‌گرداند.
Capacity	ویژگی خواندنی/نوشتنی	اندازه آرایه کاراکترها را می‌خواند یا می‌نویسد. سعی در تنظیم گنجایش با مقدار کوچکتر از طول رشته یا بزرگتر از MaxCapacity یک ArgumentOutOfRangeException تولید می‌کند.
EnsureCapacity	متد	تضمین می‌کند که آرایه کاراکتر حداقل به اندازه‌ی تعیین شده است. اگر مقدار ارسالی بزرگتر از گنجایش StringBuilder باشد گنجایش کنونی افزایش می‌یابد. اگر گنجایش کنونی از قبل بزرگتر از مقدار ارسالی به این متد باشد، تغییری حاصل نمی‌شود.
Length	ویژگی خواندنی/نوشتنی	تعداد کاراکترهای "رشته" را می‌خواند یا می‌نویسد. این احتمالا کوچکتر از گنجایش کنونی آرایه کاراکترهاست. تنظیم این ویژگی به 0 ، محتویات StringBuilder را به یک رشته خالی پاک می‌کند.
ToString	متد	نسخه‌ی بدون پارامتر این متد یک String که آرایه کاراکترهای StringBuilder را نمایش می‌دهد برمی‌گرداند.
Chars	ویژگی ایندکسر خواندنی/نوشتنی	کاراکتر را در اندیس تعیین شده از آرایه‌ی کاراکترها، می‌خواند یا به آن می‌نویسد. در سی-شارپ این یک ایندکسر (ویژگی پارامتردار) است که شما به آن با نحو آرایه (I) دسترسی دارید.
Clear	متد	محتویات شی StringBuilder را پاک می‌کند، معادل تنظیم ویژگی Length به 0 .
Append	متد	یک تک شی را به انتهای آرایه کاراکترها اضافه می‌کند، در صورت نیاز آرایه را بزرگ می‌کند. شی با استفاده از فرمت کلی و فرهنگ ترد فراخوانی کننده، به یک رشته تبدیل می‌گردد.
Insert	متد	یک تک شی را در آرایه کاراکتر درج می‌کند، در صورت نیاز آرایه را بزرگ می‌کند. شی با استفاده از فرمت کلی و فرهنگ ترد فراخوانی کننده، به یک رشته تبدیل می‌شود.
AppendFormat	متد	اشیاء تعیین شده را به انتهای آرایه کاراکترها می‌افزاید، در صورت نیاز آرایه را بزرگ می‌کند. اشیا با استفاده از فرمت و فرهنگ فراهم شده توسط فراخوانی کننده، به رشته‌ها تبدیل می‌شوند. AppendFormat یکی از رایج ترین متدهای استفاده شده با اشیا StringBuilder است.
AppendLine	متد	یک خط خالی یا یک رشته همراه با یک خط خالی را به انتهای آرایه کاراکتر می‌افزاید، در صورت نیاز گنجایش آرایه را زیاد می‌کند.
Replace	متد	یک کاراکتر را با کاراکتر دیگر یا یک رشته را با رشته دیگر درون آرایه کاراکترها تعویض می‌کند.
Remove	متد	بازه‌ای از کاراکترها را از آرایه کاراکترها حذف می‌کند.
Equals	متد	در صورتی که هر دو شی StringBuilder ماکزیمم گنجایش، گنجایش و کاراکترهای درون آرایه یکسانی داشته باشند، true برمی‌گرداند.
CopyTo	متد	یک زیر مجموعه از کاراکترهای StringBuilder را در یک آرایه Char برمی‌گرداند.

یک نکته مهم درباره‌ی متدهای **StringBuilder** اینست که اغلب آن‌ها ارجاعی به همان شی **StringBuilder** برمی‌گردانند. این اجازه می‌دهد با نحوی آسان چند عمل را به هم زنجیر کنید:

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").
    Replace(' ', '-').Remove(4, 3).ToString();
```

```
Console.WriteLine(s); // "Jeff-Richter"
```

شما متوجه خواهید شد که کلاس‌های **String** و **StringBuilder** تناظر کاملی در متدها ندارند، یعنی، **String** متدهای **ToUpper**، **ToLower**، **EndsWith**، **PadLeft**، **PadRight**، **Trim** و غیره را دارد. کلاس **StringBuilder** هیچ یک از این متدها را ارائه نمی‌کند. در سوی دیگر، کلاس **StringBuilder** یک متد غنی تر **Replace** ارائه می‌کند که به شما اجازه می‌دهد کاراکترها یا رشته‌ها را در بخشی از رشته (نه کل رشته) تعویض کنید. مایه تاسف است که تناظر کاملی بین متدهای این کلاس وجود ندارد چون اکنون برای انجام عملیات‌های خاصی مجبورید بین **String** و **StringBuilder** تبدیل کنید.

برای نمونه، برای ساخت یک رشته، تبدیل تمام کاراکترهایش به حروف بزرگ و سپس درج یک رشته، به کدی شبیه به زیر نیاز دارید:

```
// Construct a StringBuilder to perform string manipulations.
StringBuilder sb = new StringBuilder();

// Perform some string manipulations by using the StringBuilder.
sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").Replace(" ", "-");

// Convert the StringBuilder to a String in
// order to uppercase all the characters.
String s = sb.ToString().ToUpper();

// Clear the StringBuilder (allocates a new Char array).
sb.Length = 0;

// Load the uppercase String into the StringBuilder,
// and perform more manipulations.
sb.Append(s).Insert(8, "Marc-");

// Convert the StringBuilder back to a String.
s = sb.ToString();

// Display the String to the user.
Console.WriteLine(s); // "JEFFREY-Marc-RICHTER"
```

ناچور و غیرکاراست که این را فقط به این خاطر بنویسیم که **StringBuilder** تمام عملیات‌هایی که **String** انجام می‌دهد را ارائه نمی‌کند. در آینده، من امیدوارم مایکروسافت متدهای عملیات رشته بیشتری به **StringBuilder** بیفزاید تا آن را به یک کلاس کامل تر تبدیل کند.

بدست آوردن نمایش رشته ای از یک شی : ToString

شما مدام نیاز دارید یک نمایش رشته‌ای از یک شی بدست آورید. معمولاً، وقتی این نیاز است که می‌خواهید یک نوع عددی (مثل **Byte**، **Int32** و **Single**) یا یک شی **DateTime** را به کاربر نشان دهید. چون دات‌نت فریمورک یک پلفرم شی گراست، هر نوع، مسئول فراهم کردن کدبست که مقدار یک نمونه را به یک رشته معادل تبدیل کند. هنگام طراحی اینکه چگونه نوع‌ها این کار را بکنند، طراحان **FCL** الگویی ابداع کردند که همواره همه جا مورد استفاده است. در این بخش، من این الگو را توصیف می‌کنم.

شما می‌توانید یک نمایش رشته‌ای از هر شی را با فراخوانی متد **ToString** بدست آورید. یک متد عمومی، مجازی و بدون پارامتر **ToString** در **System.Object** تعریف شده است و بنابراین توسط یک نمونه از هر نوعی قابل فراخوانی است. از لحاظ معنایی، **ToString** یک رشته برمی‌گرداند که مقدار کنونی شی را نمایش می‌دهد و این رشته باید برای فرهنگ کنونی ترد فراخوانی کننده، فرمت شود، یعنی نمایش رشته‌ای یک عدد باید جداکننده‌های دهد، نماد گروه بندی ارقام و دیگر عناصر همراه با فرهنگی که به ترد فراخوانی کننده نسبت داده شده است را به درستی نشان دهد. پیاده‌سازی **ToString** توسط **System.Object** تنها نام کامل نوع شی را برمی‌گرداند. این مقدار خیلی مفید نیست اما یک پیش فرض معقولانه برای بسیاری از نوع‌هاست که رشته‌ی معقولی را ارائه نمی‌کنند. برای نمونه، نمایش رشته‌ای یک **FileStream** یا یک شی **HashTable** شبیه چه باید باشد؟

تمام نوع‌هایی که می‌خواهند یک روش معقولانه برای بدست آوردن یک رشته که مقدار کنونی شی را نشان می‌دهد، ارائه کنند، باید متد **Tostring** را بازنویسی کنند. تمام نوع‌های اصلی در **FCL (Byte, Int32, UInt64, Double)** و غیره) متد **Tostring** شان را بازنویسی کرده و یک رشته با توجه به فرهنگ برمی‌گردانند. در دیباگر ویژوال استودیو، یک مستطیل کوچک اطلاعاتی (**datatip**) هنگامی که موس روی یک متغیر خاص قرار گیرد، نمایش داده می‌شود. متن نمایش داده شده در این مستطیل با فراخوانی متد **Tostring** شی بدست می‌آید. پس وقتی یک کلاس تعریف می‌کنید شما باید همیشه متد **Tostring** را بازنویسی کنید تا پشتیبانی خوبی در خطایابی داشته باشید.

فرمت های خاص و فرهنگ ها

متد بدون پارامتر **Tostring** دو مشکل دارد. نخست، فراخوانی کننده کنترلی بر فرمت کردن رشته ندارد. برای نمونه، یک برنامه شاید بخواهد یک عدد را به یک رشته ارزی، رشته دهدهی، رشته درصدی یا رشته هگز فرمت کند. دوم اینکه، فراخوانی کننده نمی‌تواند به راحتی با انتخاب یک فرهنگ خاص یک رشته را فرمت کند. این مسئله دوم، دردسر بیشتری برای کدهای سروری در مقابل کدهای کلاینتی دارد. در سناریوهایی نادر، یک برنامه نیاز دارد یک رشته را با یک فرهنگ به جز فرهنگی که با ترد فراخوانی کننده همراه است، فرمت کند. برای آنکه کنترل بیشتری بر فرمت رشته‌ها داشته باشید، شما نیاز به نسخه‌ای از متد **Tostring** دارید که به شما اجازه‌ی تعیین دقیق اطلاعات فرمت کردن و فرهنگ را بدهد.

نوع‌هایی که به فراخوانی کننده، اجازه‌ی انتخاب فرمت و فرهنگ را می‌دهند، رابط **System.IFormattable** را پیاده‌سازی می‌کنند:

```
public interface IFormattable {
    String ToString(String format, IFormatProvider formatProvider);
}
```

در **FCL**، تمام نوع‌های اصلی (**Decimal, Double, Single, Int64/UInt64, Int32/UInt32, Int16/UInt16, SByte, Byte**) و **DateTime** این رابط را پیاده‌سازی می‌کنند. به علاوه، برخی دیگر نوع‌ها مثل **Guid**، آن را پیاده‌سازی می‌کنند. سرانجام، تعریف هر نوع شمارشی به صورت خودکار رابط **IFormattable** را پیاده‌سازی می‌کند تا بتوان یک نماد رشته‌ای معنی دار از یک نمونه از نوع شمارشی بدست آورد. متد **Tostring** از **IFormattable** دو پارامتر می‌گیرد. اولین پارامتر، **format**، یک رشته است که به متد می‌گوید شی چگونه باید فرمت شود. پارامتر دوم از **formatProvider, ToString**، یک نمونه از یک نوع است که رابط **System.IFormatProvider** را پیاده‌سازی می‌کند. این نوع، اطلاعات خاص فرهنگ را برای متد **Tostring** فراهم می‌کند. من چگونه این کار را مختصر توضیح خواهم داد.

نوعی که متد **Tostring** از رابط **IFormattable** را پیاده‌سازی می‌کند تعیین می‌کند چه رشته فرمت‌هایی را قصد دارد تشخیص دهد. اگر شما یک رشته فرمت که نوع، تشخیص نمی‌دهد، ارسال کنید، تصور می‌رود که نوع، یک **System.FormatException** تولید کند.

بسیاری از نوع‌هایی که مایکروسافت در **FCL** تعریف نمود است چندین فرمت را تشخیص می‌دهند. برای نمونه، نوع **DateTime** از "d" برای تاریخ کوتاه، "D" برای تاریخ بلند، "g" برای فرمت کلی، "M" برای ماه/روز، "S" برای قابل مرتب سازی، "T" برای ساعت بلند، "u" برای ساعت جهانی در فرمت ISO 8601، "U" برای ساعت جهانی در فرمت تاریخ کامل، "Y" برای سال/ماه و غیره پشتیبانی می‌کند. تمام نوع‌های شمارشی از "G" برای فرمت کلی، "F" برای پرچم‌ها، "D" برای دهدهی و "X" برای هگز پشتیبانی می‌کنند. من قالب بندی یا فرمت نوع‌های شمارشی را با جزئیات بیشتر در فصل ۱۵ "نوع‌های شمارشی و پرچم‌های بیتی" پوشش می‌دهم.

همچنین، تمام نوع‌های عددی اصلی "C" را برای نرخ پول، "D" را برای دهدهی، "E" را برای نماد علمی، "F" را برای ممیز ثابت، "G" را برای فرمت کلی، "N" را برای عدد، "P" را برای درصد، "R" را برای گرد کردن و "X" را برای هگز پشتیبانی می‌کنند. در واقع، نوع‌های عددی در صورتی که رشته‌های فرمتی ساده، آنچه را به دنبال آن هستید، فراهم نمی‌کنند، از رشته‌های فرمتی تصویری نیز پشتیبانی می‌کنند. رشته‌های فرمتی تصویری شامل کاراکترهای خاصی هستند که به متد **Tostring** از نوع می‌گویند دقیقاً چند رقم نمایش دهد، دقیقاً ممیز دهدهی را کجا قرار دهد و چه تعداد رقم پس از ممیز قرار دهد و غیره. برای اطلاعات بیشتر درباره رشته‌های فرمت، "Formatting Types" را در SDK دات‌نت فریم‌ورک ببینید.

برای اغلب نوع‌ها، فراخوانی **Tostring** و ارسال **null** به آن برای رشته فرمت معادل فراخوانی **Tostring** و ارسال "G" برای رشته فرمت است. به بیان دیگر، اشیاء به صورت پیش فرض خودشان را با "فرمت کلی" فرمت می‌کنند. هنگام پیاده‌سازی یک نوع، فرمتی را انتخاب کنید که فکر می‌کنید اغلب استفاده می‌شود. این فرمت، "فرمت کلی" است. ضمناً، متد **Tostring** که پارامتری نمی‌گیرد فرض می‌کند فراخوانی کننده "فرمت کلی" را می‌خواهد.

پس حال که رشته‌های فرمت را فرا گرفتید، بگذارید سراغ اطلاعات فرهنگ برویم. طبق پیش فرض، رشته‌ها با استفاده از اطلاعات فرهنگی که همراه ترد فراخوانی کننده است فرمت می‌شوند. متد بدون پارامتر **Tostring** دقیقاً این کار را می‌کند و اگر برای پارامتر **formatProvider**، شما **null** ارسال کنید، متد **Tostring** از **IFormattable** نیز همین کار را می‌کند.

اینست روشی که **ToString** اطلاعات فرمت کردن اعداد را از شی (**CultureInfo**) درخواست می‌کند. نوع‌های عددی (مثل **Decimal**) تنها اطلاعات فرمت کردن اعداد را درخواست می‌کنند. اما دیگر نوع‌ها (مثل **DateTime**) می‌توانند **GetFormat** را اینگونه فراخوانی کنند:

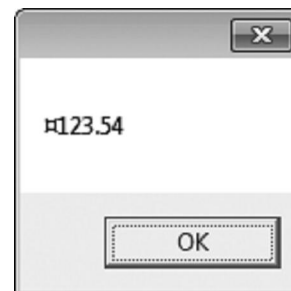
```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
    formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

در واقع، چون پارامتر **GetFormat** می‌تواند هر نوعی را شناسایی کند، متد آنقدر انعطاف پذیر است که اجازه می‌دهد هر نوعی از اطلاعات فرمت، درخواست شود. نوع‌های درون دات‌نت فریمورک، **GetFormat** را فراخوانی کرده و تنها اطلاعات عددی یا تاریخ/ساعت را درخواست می‌کنند، در آینده، دیگر گونه‌های اطلاعات فرمت کردن نیز می‌تواند درخواست شود.

ضمناً اگر شما بخواهید یک رشته برای یک شی بدست آورید که برای فرهنگ خاصی فرمت نشده باشد، شما می‌توانید ویژگی استاتیک **InvariantCulture** از **System.Globalization.CultureInfo** را فراخوانی کنید و شی برگشتی را به عنوان پارامتر **formatProvider** از **ToString** ارسال کنید:

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
MessageBox.Show(s);
```

اگر شما این کد را بسازید و اجرا کنید، جعبه‌ی پیام نشان داده شده در شکل ۶-۱۴ ظاهر می‌شود. به اولین کاراکتر در رشته خروجی دقت کنید: R. این علامت بین المللی برای نرخ پول است (U+00A4).



شکل ۶-۱۴ مقدار عددی که برای نمایش یک نرخ با فرهنگ خنثی فرمت شده است.

به صورت عادی، شما یک رشته که با یک فرهنگ غیرمتنوع (**invariant**) فرمت شده است را به کاربر نشان نمی‌دهید، معمولاً، شما این رشته را در یک فایل ذخیره می‌کنید تا بعداً از آن استفاده کنید.

در **FCL**، تنها سه نوع، رابط **IFormatProvider** را پیاده‌سازی می‌کنند. اولین آن‌ها، **CultureInfo** است که قبلاً توضیح دادم. دوتای دیگر **NumberFormatInfo** و **DateTimeFormatInfo** هستند. وقتی **GetFormat** روی یک شی **NumberFormatInfo** فراخوانی می‌شود، متد بررسی می‌کند آیا نوع درخواستی یک **NumberFormatInfo** است یا خیر. اگر هست، **this** برمی‌گردد و اگر نیست، **null** برمی‌گردد. به طریق مشابه، فراخوانی **GetFormat** روی یک شی **DateTimeFormatInfo** در صورتی که یک **DateTimeFormatInfo** درخواست شده باشد، **this** و در غیر این صورت **null** برمی‌گرداند. این دو نوع، این رابط را فقط برای راحتی برنامه‌نویسی پیاده‌سازی می‌کنند. هنگام بدست آوردن یک نمایش رشته‌ای از یک شی، فراخوانی کننده معمولاً یک فرمت را تعیین کرده و از فرهنگی که با ترد فراخوانی کننده همراه است، استفاده می‌کند. به همین دلیل، شما معمولاً **Tostring** را فراخوانی کرده، یک رشته برای پارامتر فرمت و **null** برای پارامتر **formatProvider** بدان ارسال می‌کنید. برای آنکه فراخوانی **Tostring** برای شما آسان تر باشد، اکثر نوع‌ها چندین سربارگذاری از متد **Tostring** ارائه می‌کنند. برای نمونه، نوع **Decimal** چهار متد متفاوت **Tostring** ارائه می‌کند:

```
// This version calls ToString(null, null).
// Meaning: General numeric format, thread's culture information
public override String ToString();

// This version is where the actual implementation of ToString goes.
// This version implements IFormattable's ToString method.
// Meaning: Caller-specified format and culture information
public String ToString(String format, IFormatProvider formatProvider);
```

```
// This version simply calls ToString(format, null).
// Meaning: caller-specified format, thread's culture information
public String ToString(String format);

// This version simply calls ToString(null, formatProvider).
// This version implements IConvertible's ToString method.
// Meaning: General format, caller-specified culture information
public String ToString(IFormatProvider formatProvider);
```

فرمت کردن چندین شی به یک رشته

تاکنون، من توضیح دادم یک نوع چگونه اشیاء خودش را فرمت می‌کند. گاهی، شما می‌خواهید رشته‌هایی شامل چند شی فرمت شده را بسازید. برای نمونه، رشته‌ی زیر یک تاریخ، نام یک فرد و یک سن دارد:

```
String s = String.Format("On {0}, {1} is {2} years old.",
    new DateTime(2010, 4, 22, 14, 35, 5), "Aidan", 7);
Console.WriteLine(s);
```

اگر در حالیکه فرهنگ ترد جاری "en-US" باشد، شما این کد را ساخته و اجرا کنید. خروجی زیر را می‌بینید:

```
On 4/22/2010 2:35:05 PM, Aidan is 7 years old.
```

متد استاتیک **Format** از **String**. یک رشته فرمت که پارامترهای قابل جایگزینی را با استفاده از اعداد درون آکولات‌ها شناسایی می‌کند، دریافت می‌نماید. رشته‌ی فرمت استفاده شده در این مثال به متد **Format** می‌گوید که {0} را با اولین پارامتر بعد از رشته فرمت (تاریخ/ ساعت)، {1} را با دومین پارامتر بعد از رشته فرمت ("Aidan") و {2} را با سومین پارامتر پس از رشته فرمت (7) جایگزین کند.

در درون، متد **Format**، متدهای **ToString** هر شی را برای بدست آوردن نمایش رشته‌ای شی فراخوانی می‌کند. سپس رشته‌های برگردانده شده به هم متصل شده و رشته کامل نهایی برگردانده می‌شود. همه‌ی این‌ها خوب است اما این یعنی تمام اشیاء با فرمت کلی و اطلاعات فرهنگ ترد فراخوانی کننده فرمت شده‌اند.

شما می‌توانید با تعیین اطلاعات فرمتی درون آکولات‌ها، کنترل بیشتری بر عملیات فرمت داشته باشید. برای نمونه، کد زیر معادل کد قبلی است به جز آنکه من اطلاعات فرمتی را به پارامترهای 0 و 2 افزوده‌ام:

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
    new DateTime(2010, 4, 22, 14, 35, 5), "Aidan", 7);
Console.WriteLine(s);
```

اگر شما این کد را ساخته و اجرا کنید در حالیکه "en-US" فرهنگ ترد جاری باشد، خروجی زیر را خواهید دید:

```
On Thursday, April 22, 2010, Aidan is 7.000000E+000 years old.
```

وقتی **Format**، رشته‌ی فرمت را تجزیه می‌کند، می‌بیند برای پارامتر 0 باید متد **ToString** از رابط **IFormattable** آن، با ارسال "D" و "null" برای دو پارامترش، فراخوانی گردد. همچنین، **Format**، متد **ToString** از **IFormattable** متعلق به پارامتر 2 را نیز با ارسال "E" و "null" فراخوانی می‌کند. اگر نوع، رابط **IFormattable** را پیاده‌سازی نکرده باشد، **Format**، متد بدون پارامتر **ToString** را که از **Object** به ارث برده است (و احتمالاً بازنویسی شده است) را فراخوانی می‌کند و فرمت پیش فرض به رشته خروجی اضافه می‌گردد.

کلاس **String** چندین سربارگذاری از متد استاتیک **Format** ارائه می‌کند. یک نسخه، شی‌ای می‌گیرد که رابط **IFormatProvider** را پیاده‌سازی کرده است تا شما بتوانید تمام پارامترهای قابل جایگزینی را با اطلاعات فرهنگی که فراخوانی کننده تعیین نموده، فرمت کنید. واضح است که **Format**، متد **ToString** از **IFormattable** متعلق به هر شی را با ارسال هر شی **IFormatProvider** که به **Format** ارسال شده، فراخوانی می‌کند.

اگر شما از **StringBuilder** به جای **String** برای ساخت یک رشته استفاده می‌کنید، شما می‌توانید متد **AppendFormat** از **StringBuilder** را فراخوانی کنید. این متد دقیقاً مثل متد **Format** از **String** کار می‌کند، به جز آنکه آن یک رشته را فرمت کرده و به آرایه کاراکترهای **StringBuilder** می‌افزاید. همانند **Format** از **String**، **AppendFormat** یک رشته‌ی فرمت می‌گیرد و نسخه‌ای از آن وجود دارد که یک **IFormatProvider** دریافت می‌کند.

System.Console متدهای **Write** و **WriteLine** را ارائه می‌کند که رشته‌های فرمت و پارامترهای جایگزین را دریافت می‌کنند. هرچند، هیچ سربارگذاری برای متدهای **Write** و **WriteLine** از **Console** وجود ندارد که به شما اجازه دهد یک **IFormatProvider** بدان ارسال کنید. اگر شما می‌خواهید یک رشته را برای یک فرهنگ خاص فرمت کنید، شما مجبورید متد **Format** از **String** را فراخوانی کرده، ابتدا شی **IFormatProvider** مورد نظر را بدان ارسال کرده و سپس رشته‌ی حاصل را به متد **Write** یا **WriteLine** از **Console** ارسال کنید. این نباید مشکل بزرگی باشد چون همانطور که قبلاً گفتیم، خیلی نادر است که کد سمت کلاینت، یک رشته را با فرهنگی جز فرهنگ همراه با ترد فراخوانی کننده، فرمت کند.

فراهم کردن فرمت کننده ی سفارشی خودتان

اکنون این واضح است که قابلیت‌های فرمت کردن در دات‌نت فریمورک برای آنکه به شما انعطاف پذیری و کنترل بیشتری دهند، طراحی شده اند. اما، ما هنوز بحث را تمام نکرده‌ایم. این ممکن است برای شما که یک متد تعریف کنید که متد **AppendFormat** از **StringBuilder** هر بار که بخواید یک شی را به یک رشته فرمت کند، این متد را فراخوانی کند. به بیان دیگر، به جای فراخوانی **ToString** برای هر شی، **AppendFormat** می‌تواند تابعی که شما تعریف نموده اید را فراخوانی کند و به شما اجازه دهد تمام اشیاء را به روشی که می‌خواهید فرمت کنید. آنچه می‌خواهم توضیح دهم با متد **Format** از **String** نیز کار می‌کند.

بگذارید این مکانیزم را با مثالی توضیح دهم. بگوییم که شما یک متن HTML که یک کاربر در یک مرورگر اینترنتی می‌بیند را فرمت می‌کنید. شما می‌خواهید تمام مقادیر **Int32** با فونت **Bold** نمایش داده شوند. برای انجام این کار، هر بار که یک مقدار **Int32** به یک **String** فرمت می‌شود، شما می‌خواهید در اطراف آن، برچسب (Tag)های HTML مخصوص را قرار دهید: **** و ****. کد زیر نشان می‌دهد چقدر این کار آسان است:

```
using System;
using System.Text;
using System.Threading;

public static class Program {
    public static void Main() {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}", "Jeff", 123, DateTime.Now);
        Console.WriteLine(sb);
    }
}

internal sealed class BoldInt32s : IFormatProvider, ICustomFormatter {
    public Object GetFormat(Type formatType) {
        if (formatType == typeof(ICustomFormatter)) return this;
        return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
    }

    public String Format(String format, Object arg, IFormatProvider formatProvider) {
        String s;
        IFormattable formattable = arg as IFormattable;
        if (formattable == null) s = arg.ToString();
        else s = formattable.ToString(format, formatProvider);
        if (arg.GetType() == typeof(Int32))
            return "<B>" + s + "</B>";
        return s;
    }
}
```

وقتی شما این کد را کامپایل و اجرا می‌کنید در حالیکه “en-US” فرهنگ ترد کنونی است، خروجی زیر نشان داده می‌شود (البته تاریخ ممکن است متفاوت باشد):

```
Jeff <B>123</B> January 23
```

در **Main**، من یک **StringBuilder** خالی ساخته‌ام و سپس یک رشته فرمت را بدان اضافه می‌کنم. هنگامی که من **AppendFormat** را فراخوانی می‌کنم، اولین پارامتر یک نمونه از کلاس **BoldInt32s** است. این کلاس، رابط **IFormatProvider** را که قبلاً بحث کردم پیاده‌سازی می‌کند. به علاوه، این کلاس رابط **ICustomFormatter** را نیز پیاده‌سازی می‌کند:

```
public interface ICustomFormatter {
    String Format(String format, Object arg,
        IFormatProvider formatProvider);
}
```

متد **Format** از این رابط، وقتی **AppendFormat** از **StringBuilder** نیاز دارد یک رشته برای یک شی بدست آورد، فراخوانی می‌شود. شما می‌توانید برای داشتن کنترل فراوان روی فرمت کردن رشته، درون این متد، کارهایی را که دوست دارید انجام دهید. برای آنکه ببینیم متد **AppendFormat** چگونه کار می‌کند به داخل آن نگاه می‌کنیم. شبه کد زیر نشان می‌دهد **AppendFormat** چگونه کار می‌کند:

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
    String format, params Object[] args) {

    // If an IFormatProvider was passed, find out
    // whether it offers an ICustomFormatter object.
    ICustomFormatter cf = null;

    if (formatProvider != null)
        cf = (ICustomFormatter)
            formatProvider.GetFormat(typeof(ICustomFormatter));

    // Keep appending literal characters (not shown in this pseudocode)
    // and replaceable parameters to the StringBuilder's character array.
    Boolean MoreReplaceableArgumentsToAppend = true;
    while (MoreReplaceableArgumentsToAppend) {
        // argFormat refers to the replaceable format string obtained
        // from the format parameter
        String argFormat = /* ... */;

        // argObj refers to the corresponding element
        // from the args array parameter
        Object argObj = /* ... */;

        // argStr will refer to the formatted string to be appended
        // to the final, resulting string
        String argStr = null;

        // If a custom formatter is available, let it format the argument.
        if (cf != null)
            argStr = cf.Format(argFormat, argObj, formatProvider);

        // If there is no custom formatter or if it didn't format
        // the argument, try something else.
```

```

if (argStr == null) {
    // Does the argument's type support rich formatting?
    IFormattable formattable = argObj as IFormattable;
    if (formattable != null) {
        // Yes; pass the format string and provider to
        // the type's IFormattable ToString method.
        argStr = formattable.ToString(argFormat, formatProvider);
    } else {
        // No; get the default format by using
        // the thread's culture information.
        if (argObj != null) argStr = argObj.ToString();
        else argStr = String.Empty;
    }
}
// Append argStr's characters to the character array field member.
/* ... */

// Check if any remaining parameters to format
MoreReplaceableArgumentsToAppend = /* ... */;
}
return this;
}

```

وقتی **Main**، **AppendFormat** را فراخوانی می‌کند، **AppendFormat** و **GetFormat** متد **AppendFormat** از فراهم کننده‌ی فرمت من را با ارسال نوع **IFormattable** به آن، فراخوانی می‌کند. **GetFormat** که در نوع **BoldInt32s** تعریف شده است می‌بند **IFormattable** درخواست شده است و یک ارجاع به خودش برمی‌گرداند چون این رابط را پیاده‌سازی می‌کند. اگر متد **GetFormat** من فراخوانی شده بود و هر نوع دیگری، بدان ارسال می‌شد، من متد **GetFormat** متعلق به شی **CultureInfo** که با ترد فراخوانی کننده همراه است را فراخوانی می‌کردم.

هر بار که **AppendFormat** نیاز به فرمت یک پارامتر قابل جایگزینی دارد، متد **Format** از **IFormattable** را فراخوانی می‌کند. در مثال من، **AppendFormat** متد **Format** تعریف شده در نوع **BoldInt32s** من را فراخوانی می‌کند. در متد **Format** من، من بررسی می‌کنم آیا شی‌ای که می‌خواهد فرمت شود از فرمت غنی از طریق رابط **IFormattable** پشتیبانی می‌کند یا خیر. اگر شی پشتیبانی نمی‌کند، سپس من متد ساده‌ی بدون پارامتر **ToString** را برای فرمت شی فراخوانی می‌کنم (ارث برده شده از **Object**). اگر شی، **IFormattable** را پشتیبانی می‌کند، من سپس متد غنی **ToString** را فراخوانی کرده و رشته فرمت و فراهم کننده‌ی فرمت را بدان ارسال می‌کنم.

اکنون که من رشته را فرمت کرده‌ام، من بررسی می‌کنم آیا شی یک نوع **Int32** است و اگر هست، من رشته فرمت شده را با تگ‌های HTML: **** و **** می‌پوشانم و رشته جدید را برمی‌گردانم. اگر شی یک **Int32** نیست، من فقط رشته فرمت شده را بدون هیچ گونه پردازشی برمی‌گردانم.

تجزیه یک رشته برای بدست آوردن یک شی : Parse

در بخش قبلی، من توضیح دادم چگونه یک شی را برداشته و یک نمایش رشته‌ای از آن شی بدست می‌آورید. در این بخش، من عکس آن را بحث می‌کنم: چگونه یک رشته برداشته و یک نمایش شی از آن بدست آوریم. بدست آوردن یک شی از یک رشته، عمل بسیار رایجی نیست اما در مواقعی کاربردی می‌باشد. مایکروسافت احساس کرد یک مکانیزم رسمی برای تجزیه رشته‌ها به اشیاء، نیاز است. هر نوعی که بتواند یک رشته را تجزیه کند یک متد استاتیک عمومی، به نام **Parse** تعریف می‌کند. این متد یک **String** می‌گیرد و یک نمونه از نوع را بر می‌گرداند، به گونه‌ای، **Parse** مثل یک کارخانه کار می‌کند. در **FCL**، یک متد **Parse** برای تمام نوع‌های عددی و همچنین **DateTime**، **TimeSpan** و تعداد کمی نوع‌های دیگر (مثل نوع‌های داده‌ای **SQL**) وجود دارد.

بگذارید نگاه کنیم چگونه یک رشته را باید به یک نوع عددی تجزیه کرد. تقریباً تمام نوع‌های عددی (**Byte**، **SByte**، **Int16**/**UInt16**، **Single**، **Double**، **Decimal** و **BigInteger**) حداقل یک متد **Parse** ارائه می‌کنند. من در اینجا، فقط

متد `Parse` که توسط نوع `Int32` تعریف شده است را نشان می‌دهم. (متدهای `Parse` برای دیگر نوع‌های عددی شبیه متد `Parse` از `Int32` عمل می‌کنند):

```
public static Int32 Parse(String s, NumberStyles style,
    IFormatProvider provider);
```

فقط با نگاه به فرم کلی آن، شما باید قادر باشید حدس بزنید این متد چگونه کار می‌کند. پارامتر `s` یک نمایش رشته‌ای از یک عدد که می‌خواهید آن را به یک شی `Int32` تجزیه کنید را شناسایی می‌کند. پارامتر `style` `System.Globalization.NumberStyles` مجموعه‌ای از پرچم‌های بیتی است که کاراکترهایی که `Parse` انتظار داد در رشته پیدا کند را شناسایی می‌کند. و پارامتر `provider` `IFormatProvider` یک شی را شناسایی می‌کند که متد `Parse` برای بدست آوردن اطلاعات مخصوص به فرهنگ از آن استفاده کند؛ همانطور که در ابتدای فصل گفته شد.

برای نمونه، کد زیر باعث می‌شود `Parse` یک `System.FormatException` تولید کند، چون رشته‌ی تجزیه شده حاوی یک فاصله‌ی آغازین است:

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

برای آنکه به `Parse` اجازه دهید فاصله آغازین را نادیده بگیرد، پارامتر `style` را اینگونه تعریف کنید:

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

برای توضیح کامل نمادهای بیتی و ترکیب‌های رایج که نوع شمارشی `NumberStyles` تعریف می‌کند به مستندات SDK دات‌نت فریمورک مراجعه کنید.

کد زیر نشان می‌دهد چگونه یک عدد هگز را تجزیه کنید:

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x); // Displays "26"
```

این متد `Parse` سه پارامتر می‌گیرد. برای راحتی کار، اکثر نوع‌ها چندین سربارگذاری از `Parse` تعریف می‌کنند تا شما مجبور نباشید آرگومان‌های زیادی ارسال کنید. برای نمونه، `Int32` چهار سربارگذاری از متد `Parse` تعریف می‌کند:

```
// Passes NumberStyles.Integer for style
// and thread's culture's provider information.
public static Int32 Parse(String s);

// Passes thread's culture's provider information.
public static Int32 Parse(String s, NumberStyles style);

// Passes NumberStyles.Integer for the style parameter.
public static Int32 Parse(String s, IFormatProvider provider);

// This is the method I've been talking about in this section.
public static Int32 Parse(String s, NumberStyles style,
    IFormatProvider provider);
```

نوع `DateTime` هم یک متد `Parse` تعریف می‌کند:

```
public static DateTime Parse(String s,
    IFormatProvider provider, DateTimeStyles styles);
```

این متد دقیقاً شبیه متد `Parse` که روی نوع‌های عددی تعریف شده است عمل می‌کند جز آنکه متد `Parse` از `DateTime` یک مجموعه پرچم‌های بیتی تعریف شده توسط نوع شمارشی `System.Globalization.DateTimeStyles` را به جای نوع شمارشی `NumberStyles` می‌گیرد. برای توضیح کامل نمادهای بیتی و ترکیب‌های رایج که نوع `DateTimeStyles` تعریف می‌کند به مستندات SDK دات‌نت فریمورک مراجعه کنید.

برای راحتی کار، نوع `DateTime` سه سربارگذاری از متد `Parse` تعریف می‌کند:

```
// Passes thread's culture's provider information
// and DateTimeStyles.None for the style
public static DateTime Parse(String s);
```

```
// Passes DateTimeStyles.None for the style
public static DateTime Parse(String s, IFormatProvider provider);
```

```
// This is the method I've been talking about in this section.
public static DateTime Parse(String s,
    IFormatProvider provider, DateTimeStyles styles);
```

تجزیه تاریخ‌ها و ساعت‌ها پیچیده است. بسیاری برنامه‌نویسان متد **Parse** از **DateTime** را بسیار سخاوتمند یافته‌اند که آن، گاهی رشته‌هایی که حاوی تاریخ و ساعت نیستند را هم تجزیه می‌کند. به همین دلیل، نوع **DateTime** یک متد **ParseExact** هم تعریف می‌کند که یک رشته فرمت تصویری دریافت می‌کند که بیان می‌نماید رشته تاریخ/ساعت دقیقا چگونه باید فرمت و چگونه باید تجزیه شود. برای اطلاعات بیشتر پیرامون رشته‌های فرمت تصویری کلاس **DateTimeFormatInfo** را در SDK دات‌نت فریمورک ببینید.

نکته بعضی برنامه‌نویسان مطلب زیر را به میکروسافت گزارش کرده‌اند. وقتی برنامه‌ی آن‌ها **Parse** را مکررا فراخوانی می‌کند، و **Parse** (به خاطر ورودی غیر مجاز کاربر) مکررا اکسپشن تولید می‌کند، عملکرد برنامه ضعیف می‌شود. برای استفاده‌هایی از **Parse** که عملکرد مهم است، میکروسافت متدهای **TryParse** را به تمام نوع‌های عددی، **DateTime**، **DateTimeOffset**، **TimeSpan**، و حتی **IPAddress** افزوده است. یکی از دو سربرگذاری دو متد **TryParse** از **Int32** اینگونه است:

```
public static Boolean TryParse(String s, NumberStyles style,
    IFormatProvider provider, out Int32 result);
```

همانگونه که می‌بینید، این متد **true** یا **false** برمی‌گرداند و نشان می‌دهد آیا رشته تعیین شده قابل تبدیل به یک **Int32** هست یا خیر. اگر متد **true** برگرداند متغیری که به پارامتر نتیجه با ارجاع ارسال شده است، شامل یک نوع عددی تجزیه شده خواهد بود. الگوی **TryXxx** در فصل ۲۰ "اکسپشن‌ها و مدیریت وضعیت" بحث می‌شود.

Encoding: تبدیل بین کاراکترها و بایت‌ها

در Win32، برنامه‌نویسان مکررا مجبور بودند کدی بنویسند که کاراکترهای یونیکد را به رشته و به کاراکترهای "مجموعه کاراکترهای چند بایتی" Multi-Byte Character Set (MBCS) تبدیل کنند. من خودم سهمی از نوشتن چنین کدهایی داشتم و نوشتن آن‌ها بسیار خسته کننده و استفاده از آن‌ها همیشه با بروز خطا همراه بود. در CLR، تمام کاراکترها به عنوان مقادیر کد ۱۶ بیتی یونیکد نمایش داده می‌شوند و تمام رشته‌ها از مقادیر کد ۱۶ بیتی یونیکد تشکیل می‌شوند. این، کار با کاراکترها و رشته‌ها را در زمان اجرا آسان می‌کند.

گاهی، شما می‌خواهید رشته‌ها را در یک فایل ذخیره یا در شبکه ارسال کنید. اگر رشته‌ها اکثرا از کاراکترهای قابل خواندن توسط مردم انگلیسی زبان تشکیل شده باشند، ذخیره یا ارسال یک مجموعه مقادیر ۱۶ بیتی خیلی کارآمد نیست چون نیمی از بایت‌های نوشته شده صفر خواهند بود. به جای آن، بهتر است که مقادیر ۱۶ بیتی را به یک آرایه از بایت‌ها اینکد **encode** کرده و سپس آرایه‌ی بایت‌ها را به یک آرایه از مقادیر ۱۶ بیتی دیکد **decode** کنید.

اینکدینگ **Encoding** همچنین اجازه می‌دهد یک برنامه مدیریت‌شده با رشته‌های ساخته شده توسط سیستم‌های غیر یونیکد نیز کار کند. برای نمونه، اگر شما می‌خواهید یک فایل تولید کنید که توسط یک برنامه در نسخه ژاپنی از ویندوز ۹۵ قابل خواندن باشد، شما مجبورید متن یونیکد را با استفاده از اینکدینگ **Shift-JIS** (صفحه کد ۹۳۲) ذخیره کنید. به همین طریق، شما از اینکدینگ **Shift-JIS** برای خواندن یک فایل متنی در CLR که توسط یک سیستم ویندوز ۹۵ ژاپنی تولید شده است، استفاده می‌کنید.

اینکدینگ معمولا وقتی می‌خواهید یک رشته را به یک فایل یا استریم شبکه با استفاده از نوع **System.IO.BinaryWriter** ارسال کنید انجام می‌شود. دیکدینگ معمولا وقتی می‌خواهید یک رشته را از فایل یا استریم شبکه با استفاده از نوع **System.IO.BinaryReader** یا **System.IO.StreamReader** بخوانید، انجام می‌شود. اگر شما صریحا یک اینکدینگ انتخاب نکنید، تمام این نوع‌ها، پیش فرض را بر **UTF-8** می‌گذارند (UTF مخفف "فرمت تبدیل یونیکد" Unicode Transformation Format است). هرچند گاهی شاید شما بخواهید صریحا یک رشته را اینکد یا دیکد کنید. حتی اگر شما نخواهید صریحا این کار را بکنید، این بخش، به شما دید بیشتری درباره‌ی خواندن و نوشتن رشته‌ها در استریم‌ها می‌دهد. خوشبختانه، **FCL** نوع‌هایی ارائه می‌کند که اینکدینگ و دیکدینگ را آسان می‌کند. دو اینکدینگ رایج تر **UTF-16** و **UTF-8** هستند.

- UTF-16 هر کاراکتر را به عنوان ۲ بایت اینکد می‌کند. بر کاراکترها اصلا اثر نمی‌گذارد و هیچ فشردگی صورت نمی‌گیرد – عملکردش عالیست. اینکدینگ UTF-16 با عنوان اینکدینگ Unicode نیز اطلاق می‌شود. همچنین دقت کنید که UTF-16 را می‌توان برای تبدیل از little-endian به big-endian و برعکس استفاده نمود.
 - UTF-8 بعضی کاراکترها را به عنوان ۱ بایت، بعضی کاراکترها را به عنوان ۲ بایت، بعضی کاراکترها را به عنوان ۳ بایت و بعضی کاراکترها را به عنوان ۴ بایت اینکد می‌کند. کاراکترهایی با مقدار کمتر از 0x0080 به ۱ بایت فشردگی می‌شوند، که برای کاراکترهایی که در ایالات متحده استفاده می‌شوند بسیار خوب کار می‌کند. کاراکترهای بین 0x0080 و 0x07FF به ۲ بایت تبدیل می‌شوند که برای زبان‌های اروپایی و خاورمیانه خوب کار می‌کند. کاراکترهای 0x0800 و بالاتر به ۳ بایت تبدیل می‌شوند و برای زبان‌های آسیای شرقی خوب کار می‌کند. سرانجام، جفت‌های جانشین (surrogate pair) به عنوان ۴ بایت نوشته می‌شوند. UTF-8 یک اینکدینگ بسیار معروف است اما در اینکد کردن کاراکترهای فراوان با مقادیر 0x0800 یا بیشتر، ضعیف‌تر از UTF-16 عمل می‌کند.
- اگرچه اینکدینگ‌های UTF-16 و UTF-8 رایج‌ترین هستند اما FCL اینکدینگ‌های دیگری که کمتر استفاده می‌شوند را نیز پشتیبانی می‌کند:
- UTF-32 تمام کاراکترها را به عنوان ۴ بایت اینکد می‌کند. این اینکدینگ هنگامی که شما می‌خواهید یک الگوریتم ساده برای بررسی کاراکترها بنویسید و نمی‌خواهید درگیر کاراکترهایی شوید که تعداد متغیری از بایت‌ها را دارند، مفید است. برای نمونه، با UTF-32، شما نیاز ندارید درباره‌ی جانشین‌ها فکر کنید چون هر کاراکتری ۴ بایتی است. بدیهی است که UTF-32 یک اینکدینگ کارا در بحث مصرف حافظه نیست و برای همین به ندرت در ذخیره یا ارسال یک رشته به یک فایل یا شبکه استفاده می‌شود. این اینکدینگ معمولاً درون خود برنامه استفاده می‌شود. همچنین دقت کنید که UTF-32 می‌تواند برای تبدیل از little-endian به big-endian و بر عکس استفاده شود.
 - UTF-7 معمولاً با سیستم‌های قدیمی که با کاراکترهایی که با ۷ بیت قابل بیان هستند کار می‌کنند، استفاده می‌شود. شما باید از این اینکدینگ خودداری کنید چون آن معمولاً به جای فشردگی سازی، داده را باز می‌کند. کنسرسیوم یونیکد، این اینکدینگ را نامناسب دانسته است.
 - ASCII کاراکترهای ۱۶ بیتی را به کاراکترهای ASCII اینکد می‌کند، یعنی، هر کاراکتر ۱۶ بیتی با مقداری کمتر از 0x0080 به یک تک بایت تبدیل می‌شود. هر کاراکتر با مقداری بیش از 0x007F قابل تبدیل نیست، پس مقدار کاراکتر از بین می‌رود. برای رشته‌هایی شامل کاراکترهایی در بازه‌ی ASCII (0x00 تا 0x7F)، این اینکدینگ، داده‌ها را به نصف فشردگی می‌کند و بسیار سریع است (چون فقط بایت بالا حذف می‌شود). اگر شما کاراکترهایی خارج از بازه‌ی ASCII دارید، این اینکدینگ مناسب نیست چون مقادیر کاراکتر از دست خواهد رفت.
- سرانجام، FCL به شما اجازه می‌دهد کاراکترهای ۱۶ بیتی را به یک صفحه کد دلخواه خود اینکد کنید. همانند اینکدینگ ASCII، اینکد کردن به یک صفحه کد خطرناک است چون هر کاراکتری که مقدارش در صفحه کد تعیین شده، قابل بیان نباشد، از دست خواهد رفت. شما همیشه باید از اینکدینگ‌های UTF-16 یا UTF-8 استفاده کنید. مگر آنکه بخواهید با فایل‌های قدیمی یا برنامه‌هایی که قبلاً از این اینکدینگ‌ها استفاده می‌کردند، کار کنید.
- وقتی شما نیاز به اینکد و دیکد کردن یک مجموعه از کاراکترها دارید، شما باید یک نمونه از یک کلاس مشتق شده از `System.Text.Encoding` بدست آورید که یک کلاس پایه خلاصه است و چندین ویژگی استاتیک فقط-خواندنی که هر کدام یک نمونه از یک کلاس مشتق شده از `Encoding` را برمی‌گرداند، ارائه می‌کند. مثال زیر، کاراکترها را با استفاده از UTF-8 اینکد و دیکد می‌کند:

```
using System;
using System.Text;
```

```
public static class Program {
    public static void Main() {
        // This is the string we're going to encode.
        String s = "Hi there.";

        // Obtain an Encoding-derived object that knows how
        // to encode/decode using UTF8
        Encoding encodingUTF8 = Encoding.UTF8;

        // Encode a string into an array of bytes.
        Byte[] encodedBytes = encodingUTF8.GetBytes(s);
```

```

// Show the encoded byte values.
Console.WriteLine("Encoded bytes: " +
    BitConverter.ToString(encodedBytes));

// Decode the byte array back to a string.
String decodedString = encodingUTF8.GetString(encodedBytes);

// Show the decoded string.
Console.WriteLine("Decoded string: " + decodedString);
}
}

```

این کده، خروجی زیر را تولید می‌کند:

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
```

```
Decoded string: Hi there.
```

علاوه بر ویژگی استاتیک **UTF8**، کلاس **Encoding** ویژگی‌های استاتیک زیر را نیز ارائه می‌کند: **Unicode**، **BigEndianUnicode**، **UTF32**، **UTF7**، **ASCII** و **Default**. ویژگی **Default**، یک شی که قادر است با استفاده از صفحه کد کاربر که توسط گزینه **Language for Non-Unicode Programs** از پنجره **Regional And Language Control Panel** تعیین می‌شود، اینکد و دیکد کند را برمی‌گرداند. (برای اطلاعات بیشتر تابع **GetACP Win32** را ببینید). هر چند، استفاده از ویژگی **Default** توصیه نمی‌شود چون رفتار برنامه شما وابسته به ماشین خواهد شد پس اگر شما صفحه کد پیش فرض سیستم را تغییر دهید یا اگر برنامه شما روی ماشین دیگری اجرا شود، برنامه شما متفاوت رفتار خواهد کرد. علاوه برای این ویژگی‌ها، **Encoding** همچنین یک متد استاتیک **GetEncoding** ارائه می‌کند که به شما اجازه می‌دهد یک صفحه کد (با عدد یا با رشته) تعیین کنید و یک شی که بتواند با این صفحه کد، اینکد و دیکد کند را برمی‌گرداند. برای نمونه، شما می‌توانید **GetEncoding** را فراخوانی کرده، **"Shift-JIS"** یا **932** را بدان ارسال کنید.

وقتی شما برای اولین بار یک شی اینکدینگ درخواست می‌کنید، ویژگی استاتیک یا متد **GetEncoding** از کلاس **Encoding**، یک تک شی برای اینکدینگ درخواست شده ساخته و این شی را برمی‌گرداند. اگر یک شی اینکدینگ که قبلاً درخواست شده است، در آینده درخواست گردد کلاس اینکدینگ به سادگی شی‌ای را که قبلاً ساخته، برمی‌گرداند؛ آن برای هر درخواست یک شی جدید نمی‌سازد. این کار، تعداد اشیاء موجود در سیستم را کاهش داده و از فشار جمع آوری‌های هیپ می‌کاهد.

به جای فراخوانی ویژگی‌های استاتیک **Encoding** یا متد **GetEncoding** آن، شما همچنین می‌توانستید یک نمونه از یکی از کلاس‌های زیر بسازید: **System.Text.UnicodeEncoding**، **System.Text.UTF8Encoding**، **System.Text.UTF32Encoding** یا **System.Text.ASCIIEncoding** یا **System.Text.UTF7Encoding**. هر چند، به خاطر داشته باشید ساختن هر یک از این کلاس‌ها، اشیاء جدیدی را در هیپ می‌سازد که کارایی را کاهش می‌دهد.

چهار تا از این کلاس‌ها، **UnicodeEncoding**، **UTF8Encoding**، **UTF32Encoding** و **UTF7Encoding** چندین سازنده ارائه می‌کنند که به شما کنترل بیشتری بر اینکدینگ و مقدمه می‌دهند (مقدمه **Preamble**، گاهی با عنوان یک **byte order mark** یا **BOM** اطلاق می‌شود). سه مورد اول از کلاس‌های مزبور، همچنین سازنده‌هایی ارائه می‌کنند که به شما اجازه می‌دهند به کلاس بگویند هنگام دیکدینگ یک دنباله غیر معتبر از بایت-ها، اکسپشن تولید کند؛ وقتی می‌خواهید برنامه تان در مقابل داده‌های ورودی غیر معتبر، مقاوم و ایمن باشد، شما باید از این سازنده‌های استفاده کنید.

شاید شما بخواهید هنگام کار با یک **BinaryWriter** یا یک **StreamWriter**، صریحاً نمونه‌هایی از این نوع‌های اینکدینگ بسازید. کلاس **ASCIIEncoding** تنها یک سازنده دارد و بنابراین، هیچ کنترلی بر اینکدینگ به شما ارائه نمی‌کند. اگر شما به یک شی **ASCIIEncoding** نیاز دارید، همیشه آن را از طریق ویژگی **ASCII** از **Encoding** بدست آورید، این ویژگی یک ارجاع به یک تک شی **ASCIIEncoding** برمی‌گرداند. اگر شما اشیاء **ASCIIEncoding** را خودتان بسازید، شما اشیاء بیشتری در هیپ ایجاد نموده اید که عملکرد برنامه شما را پایین می‌آورد. وقتی شما یک شی مشتق شده از **Encoding** دارید، شما می‌توانید یک رشته یا یک آرایه از کاراکترها را به یک آرایه از بایت‌ها با فراخوانی متد **GetBytes** تبدیل کنید. (چندین سربارگذاری از این متد وجود دارد) برای تبدیل یک آرایه از بایت‌ها به یک آرایه از کاراکترها، متد **GetChars** یا متد مفیدتر **GetString** را فراخوانی کنید. (چندین سربارگذاری از هر دوی این متدها وجود دارد). کد قبلی، فراخوانی به متدهای **GetBytes** و **GetString** را نشان داده است.

تمام نوع‌های مشتق شده از **Encoding** یک متد **GetByteCount** ارائه می‌کنند که تعداد بایت‌های لازم برای اینکد کردن یک مجموعه کاراکتر را در حقیقت بدون اینکد کردن بدست می‌آورد. اگرچه **GetByteCount** به صورت خاص مفید نیست ولی شما می‌توانید از آن برای تخصیص یک آرایه از بایت‌ها استفاده کنید. یک متد **GetCharCount** نیز وجود دارد که تعداد کاراکترهایی که دیکد می‌شوند را بدون دیکد کردن آن‌ها، برمی‌گرداند. اگر شما سعی در صرفه جویی در حافظه و استفاده مجدد از یک آرایه دارید این متدها مفید خواهند بود.

متدهای **GetByteCount/GetCharCount** آنقدر سریع نیستند. چون آنها باید آرایه کاراکترها/بایت‌ها را تحلیل کنند تا یک نتیجه دقیق برگردانند. اگر شما سرعت را بر نتیجه دقیق ترجیح می‌دهید، شما می‌توانید متد **GetMaxByteCount** یا **GetMaxCharCount** را به جای آن استفاده کنید. هر دو متد یک عدد صحیح می‌گیرند که تعداد کاراکترها یا تعداد بایت‌ها را مشخص می‌کنند و یک مقدار در بدترین حالت ممکن را برمی‌گردانند.

هر شی مشتق شده از **Encoding**، یک مجموعه ویژگی‌های فقط-خواندنی ارائه می‌کند که شما برای بدست آوردن اطلاعات راجع به اینکدینگ می‌توانید آن‌ها را بخوانید. برای توضیح این ویژگی‌ها به مستندات SDK دات‌نت فریمورک مراجعه کنید. برای آنکه اکثر این ویژگی‌ها و معانی آن‌ها را نشان دهیم، من برنامه زیر را نوشتم که مقادیر ویژگی‌ها برای چندین اینکدینگ مختلف را نشان می‌دهد:

```
using System;
using System.Text;

public static class Program {
    public static void Main() {
        foreach (EncodingInfo ei in Encoding.GetEncodings()) {
            Encoding e = ei.GetEncoding();
            Console.WriteLine("{1}{0}" +
                "\tCodePage={2}, WindowsCodePage={3}{0}" +
                "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +
                "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +
                "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",
                Environment.NewLine,
                e.EncodingName, e.CodePage, e.WindowsCodePage,
                e.WebName, e.HeaderName, e.BodyName,
                e.IsBrowserDisplay, e.IsBrowserSave,
                e.IsMailNewsDisplay, e.IsMailNewsSave);
        }
    }
}
```

اجرای این برنامه خروجی زیر را تولید می‌کند (برای مصرف کاغذ کمتر، خلاصه شده است)

```
IBM EBCDIC (US-Canada)
CodePage=37, WindowsCodePage=1252
WebName=IBM037, HeaderName=IBM037, BodyName=IBM037
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False

OEM United States
CodePage=437, WindowsCodePage=1252
WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False

IBM EBCDIC (International)
CodePage=500, WindowsCodePage=1252
```



```
WebName=IBM500, HeaderName=IBM500, BodyName=IBM500
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Arabic (ASMO 708)

```
CodePage=708, windowsCodePage=1256
WebName=ASMO-708, HeaderName=ASMO-708, BodyName=ASMO-708
IsBrowserDisplay=True, IsBrowserSave=True
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode

```
CodePage=1200, windowsCodePage=1200
WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
IsBrowserDisplay=False, IsBrowserSave=True
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode (Big-Endian)

```
CodePage=1201, windowsCodePage=1200
WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Western European (DOS)

```
CodePage=850, windowsCodePage=1252
WebName=ibm850, HeaderName=ibm850, BodyName=ibm850
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode (UTF-8)

```
CodePage=65001, windowsCodePage=1200
WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
IsBrowserDisplay=True, IsBrowserSave=True
IsMailNewsDisplay=True, IsMailNewsSave=True
```

جدول ۳-۱۴ متدهای پر استفاده که توسط تمام کلاس‌های مشتق شده از Encoding ارائه می‌شوند را لیست می‌کند.

جدول ۳-۱۴ متدهای کلاس‌های مشتق شده از Encoding

متد	توضیح
GetPreamble	یک آرایه از بایت‌ها را برمی‌گرداند که تعیین می‌کند چه چیزی قبل از نوشتن هر بایت اینکد شده، به یک استریم باید نوشته شود. غالباً، این بایت‌ها با عنوان بایت‌ها BOM یا BOM d اطلاق می‌شوند. وقتی شما شروع به خواندن از یک استریم می‌کنید، بایت‌های BOM به صورت خودکار در تشخیص اینکدینگ که هنگام نوشتن به استریم استفاده شده است کمک می‌کنند تا دیگر مناسب استفاده شود. برای بعضی کلاس‌های مشتق شده از Encoding ، این متد یک آرایه از بایت‌های 0 برمی‌گرداند — یعنی هیچ بایت مقدمه‌ای وجود ندارد. یک شی UTF8Encoding می‌تواند صریحاً ساخته شود تا متد آن یک آرایه سه بایتی از 0xBF ، 0xBB ، 0xEF برگرداند. یک شی UnicodeEncoding می‌تواند صریحاً ساخته شود تا متد آن یک آرایه دو بایتی از 0xFF و 0xFE برای اینکدینگ big-endian یا یک آرایه ۲ بایتی از 0xFF و 0xFE برای اینکدینگ little-endian برگرداند. پیش فرض، little-edian است.

Convert	یک آرایه از بایت‌های تعیین شده در یک اینکدینگ منبع را به یک آرایه از بایت‌های تعیین شده توسط یک اینکدینگ مقصد تبدیل می‌کند. در درون، این متد استاتیک، متد GetChars از شی اینکدینگ منبع را فراخوانی کرده و نتیجه را به متد GetBytes از شی اینکدینگ مقصد، ارسال می‌کند. آرایه بایتی حاصل، به فراخوانی کننده بازگردانده می‌شود.
Equals	اگر دو شی مشتق شده از Encoding صفحه کد و تنظیمات مقدمه‌ی یکسانی برگردانند، true برمی‌گرداند.
GetHashCode	صفحه کد شی اینکدینگ را برمی‌گرداند.

اینکدینگ و دیکدینگ استریم‌های کاراکترها و بایت‌ها

تصور کنید که شما یک رشته که با UTF-16 اینکد شده را از طریق یک شی **System.Net.Sockets.NetworkStream** می‌خوانید، بایت‌ها به احتمال زیاد به صورت تکه‌های داده ارسال می‌شوند. به بیان دیگر، ابتدا شاید شما ۵ بایت از استریم بخوانید، و سپس ۷ بایت دیگر بخوانید. در UTF-16، هر کاراکتر از ۲ بایت تشکیل می‌شود، پس فراخوانی متد **GetString** از **Encoding** و ارسال آرایه اول ۵ بایتی، یک رشته که فقط حاوی دو کاراکتر است برمی‌گرداند. اگر شما بعداً **GetString** را فراخوانی کنید و ۷ بایت بعدی که از استریم می‌آید را بدان ارسال کنید، **GetString** یک رشته شامل سه کاراکتر را برمی‌گرداند و تمام آن‌ها مقادیر اشتباهی خواهند داشت!

این مسئله تخریب داده به این دلیل اتفاق می‌افتد که هیچ کدام از کلاس‌های مشتق شده از **Encoding** هیچ وضعیتی در بین فراخوانی به متدهایشان را نگهداری نمی‌کنند. اگر شما کاراکترها/بایت را تکه تکه اینکد و دیکد می‌کنید، شما باید کاری اضافه انجام دهید تا وضعیت بین فراخوانی‌ها را نگهداری کنید تا داده‌ها از دست نروند. برای دیکد کردن تکه‌های بایت‌ها، شما باید یک ارجاع به یک شی مشتق شده از **Encoding** (که در بخش قبلی گفته شد) بدست آورید و متد **GetDecoder** آنرا فراخوانی کنید. این متد یک ارجاع به یک شی جدید ساخته شده که نوعش از کلاس **System.Text.Decoder** مشتق گردیده را برمی‌گرداند. همانند کلاس **Encoding**، کلاس **Decoder** یک کلاس پایه خلاصه است. اگر شما به مستندات SDK دات‌نت فریمورک نگاه کنید، شما هیچ کلاسی که پیاده‌سازی‌هایی از کلاس **Decoder** ارائه کند را نمی‌یابید. هر چند، CLR تعدادی کلاس مشتق شده از **Decoder** ارائه می‌کند. این کلاس‌ها تماماً در FCL، درونی هستند اما متد **GetDecoder** می‌تواند نمونه‌هایی از این کلاس‌ها بسازد و آن‌ها را به کد برنامه برگرداند.

تمام کلاس‌های مشتق شده از **Decoder** دو متد مهم ارائه می‌کنند: **GetChars** و **GetCharCount**. واضح است که این متدها برای دیکد کردن یک آرایه از بایت‌ها، استفاده می‌شوند و شبیه متدهای **GetChars** و **GetCharCount** از **Encoding** که قبلاً بحث شدند، عمل می‌کنند. وقتی شما یکی از این متدها را فراخوانی می‌کنید، آرایه بایت‌ها را تا حد ممکن دیکد می‌کند. اگر آرایه بایت شامل تعداد بایت کافی برای تشکیل یک کاراکتر کامل نباشد، بایت‌های باقی مانده، در شی دیکدر ذخیره می‌شوند. دفعه‌ی بعد که شما یکی از این متدها را فراخوانی کنید، شی دیکدر از بایت‌های باقی مانده به اضافه‌ی بایت‌های جدید ارسال به آن، استفاده می‌کند. – این، اطمینان حاصل می‌کند که تکه‌های داده به درستی دیکد می‌شوند. اشیاء **Decoder** هنگام خواندن بایت‌ها از یک استریم بسیار مفید هستند.

یک نوع مشتق شده از **Encoding** می‌تواند برای اینکدینگ و دیکدینگ بدون حفظ وضعیت، استفاده شود. هر چند، یک نوع مشتق شده از **Decoder** می‌تواند تنها برای دیکدینگ استفاده گردد. اگر شما بخواهید رشته‌ها را تکه تکه اینکد کنید، به جای متد **GetDecoder** از شی **Encoding**، متد **GetEncoder** را فراخوانی کنید. **GetEncoder** یک شی جدیداً ساخته شده که نوعش از کلاس پایه **System.Text.Encoder** مشتق شده است را برمی‌گرداند. مجدداً مستندات SDK دات‌نت فریمورک شامل هیچ کلاسی که پیاده‌سازی‌هایی از کلاس **Encoder** ارائه کند نیست. هر چند، FCL تعدادی کلاس مشتق شده از **Encoder** تعریف می‌کند، همانند کلاس‌های مشتق شده از **Decoder**، این کلاس‌ها تماماً در FCL درونی هستند، اما متد **GetEncoder** می‌تواند نمونه‌هایی از این کلاس‌ها بسازد و آن‌ها را به کد برنامه برگرداند.

تمام کلاس‌های مشتق شده از **Encoder** دو متد مهم ارائه می‌کنند: **GetBytes** و **GetByteCount**. در هر فراخوانی، شی مشتق شده از **Encoder** هر وضعیت باقی مانده را نگهداری می‌کند تا شما بتوانید تکه‌های داده را اینکد کنید.

اینکدینگ و دیکدینگ رشته مبنای ۶۴

در زمان این کتاب، اینکدینگ‌های UTF-8 و UTF-16 کاملاً معروف هستند. همچنین بسیار متداول است که یک دنباله از بایت‌ها را به رشته مبنای ۶۴ اینکد کنید. FCL متدهایی ارائه می‌کند که اینکدینگ و دیکدینگ مبنای ۶۴ را انجام می‌دهند و شاید شما انتظار داشته باشید این کار از طریق یک نوع مشتق شده از **Encoding** صورت پذیرد. هر چند، به دلایلی، اینکدینگ و دیکدینگ مبنای ۶۴ با استفاده از تعدادی متد استاتیک که توسط نوع **System.Convert** ارائه می‌شود صورت می‌پذیرد. برای اینکد کردن یک رشته مبنای ۶۴ به عنوان یک آرایه از بایت‌ها شما متد استاتیک

به یک رشته مبنای ۶۴، شما متد استاتیک **ToBase64String** یا **ToBase64CharArray** از **Convert** را فراخوانی می‌کنید. به همین طریق، برای دیدن کردن یک آرایه از بایت‌ها به یک رشته مبنای ۶۴، شما متد استاتیک **FromBase64String** یا **FromBase64CharArray** از **Convert** را فراخوانی می‌کنید. کد زیر نشان می‌دهد چگونه از این متدها استفاده کنید:

```
using System;

public static class Program {
    public static void Main() {
        // Get a set of 10 randomly generated bytes
        Byte[] bytes = new Byte[10];
        new Random().NextBytes(bytes);

        // Display the bytes
        Console.WriteLine(BitConverter.ToString(bytes));

        // Decode the bytes into a base-64 string and show the string
        String s = Convert.ToBase64String(bytes);
        Console.WriteLine(s);

        // Encode the base-64 string back to bytes and show the bytes
        bytes = Convert.FromBase64String(s);
        Console.WriteLine(BitConverter.ToString(bytes));
    }
}
```

کامپایل این کد و اجرای فایل اجرایی، خروجی زیر را تولید می‌کند (خروجی شما شاید متفاوت از مال من باشد چون بایت‌ها به صورت تصادفی تولید شده اند):

```
3B-B9-27-40-59-35-86-54-5F-F1
07knQFk1h1Rf8Q==
3B-B9-27-40-59-35-86-54-5F-F1
```

رشته های امن

اغلب، رشته‌های **String** برای نگهداری اطلاعات حساس مثل رمز عبور یک کاربر یا اطلاعات کارت اعتباری استفاده می‌شوند. متأسفانه، رشته‌های **String** حاوی یک آرایه از کاراکترها در حافظه هستند و اگر کد ناامن یا مدیریت نشده اجازه‌ی اجرا پیدا کند، کد ناامن / مدیریت نشده می‌تواند اطراف فضای آدرس پردازش جاسوسی کرده، رشته حاوی اطلاعات حساس را یافته و از این اطلاعات به روش غیرمجاز استفاده کند. حتی اگر شی **String** برای یک زمان کوتاه استفاده گردد و سپس جمع‌آوری شود، CLR شاید بلافاصله از حافظه‌ی شی **String** استفاده مجدد نکند (بخصوص اگر شی **String** خیلی قبل تر تولید شده باشد)، کاراکترهای رشته را در حافظه‌ی پردازش رها کند، جاییکه اطلاعات می‌تواند به سرقت رود. به علاوه، چون رشته‌ها تغییر ناپذیرند، وقتی شما آن‌ها را دستکاری می‌کنید، کپی‌های قدیمی در حافظه باقی می‌مانند و شما چندین نسخه مختلف از رشته را در سراسر حافظه پخش کرده اید.

برخی اداره‌های دولتی، پیش نیازهای سخت امنیتی دارند که نیاز به تضمین‌های بسیار خاص امنیتی دارد. برای تامین این پیش نیازها، میکروسافت یک کلاس رشته امن تر به **FCL** افزوده است: **System.Security.SecureString**. وقتی شما یک شی **SecureString** می‌سازید، آن در درون، یک بلوک از حافظه مدیریت نشده که حاوی یک آرایه از کاراکترهاست می‌سازد. حافظه مدیریت نشده به این منظور استفاده می‌شود که جمع‌آوری کننده از آن مطلع نباشد. کاراکترهای این رشته رمزنگاری شده‌اند و اطلاعات حساس را از هر کد ناامن / مدیریت نشده مخرب، محافظت می‌کنند. شما می‌توانید با یکی از این متدها، یک کاراکتر به رشته امن اضافه، درج، حذف یا تنظیم کنید: **AppendChar**، **InsertAt**، **RemoveAt** و **SetAt**. هرگاه شما یکی از این متدها را فراخوانی می‌کنید، در درون، متد، کاراکترها را رمزگشایی می‌کند، عملیات را در محل انجام می‌دهد و سپس کاراکترها را مجدداً رمزنگاری می‌کند. این یعنی کاراکترها تنها برای یک مدت زمان بسیار کوتاه در وضعیت رمزنگاری نشده قرار دارند. این همچنین یعنی کارایی این عمل کمتر از حد انتظار است. پس شما باید تا حد ممکن تعداد کمی از این عملیات‌ها را انجام دهید.

کلاس **SecureString**، رابط **IDisposable** را پیاده‌سازی می‌کند تا یک راه آسان برای از بین بردن قطعی محتویات رشته امن، فراهم کند. وقتی برنامه شما دیگر نیازی به اطلاعات رشته‌ی حساس ندارد، شما به سادگی متد **Dispose** از **SecureString** را فراخوانی می‌کنید. در درون، **Dispose** تمام محتویات بافر حافظه را صفر می‌کند تا مطمئن شود که اطلاعات حساس در دسترس کد مخرب نباشد و سپس بافر آزاد می‌شود. در درون، یک شی **SecureString** یک فیلد به یک شی مشتق شده از **SafeBuffer** دارد که رشته واقعی را نگهداری می‌کند. چون کلاس **SafeBuffer** در نهایت از **CriticalFinalizerObject** مشتق می‌شود، که در فصل ۲۱ "مدیریت حافظه خودکار (جمع آوری زباله)" توضیح داده شده است، کاراکترهای رشته تضمین می‌شود که صفر شوند و وقتی آن جمع آوری گردد، بافر آزاد می‌شود. بر خلاف یک شی **String**، وقتی یک شی **SecureString** جمع آوری می‌شود، کاراکترهای رمزنگاری شده‌ی رشته دیگر در حافظه باقی نمی‌مانند.

حال که می‌دانید چگونه یک شی **SecureString** ساخته و آنرا تغییر دهید، بگذارید درباره‌ی نحوه‌ی استفاده از آن نیز توضیح دهم متأسفانه، جدیدترین FCL، پشتیبانی اندکی از کلاس **SecureString** می‌کند. به بیان دیگر، تنها تعداد کمی متد وجود دارد که یک آرگومان **SecureString** می‌پذیرند. در نسخه 4.0 از دات‌نت فریمورک، شما می‌توانید یک **SecureString** را به عنوان رمز عبور ارسال کنید هنگام :

- کار با یک فراهم کننده‌ی سرویس رمزنگاری (CSP). کلاس **System.Security.Cryptography.CspParameters** را ببینید.
- ساخت، وارد کردن یا صادر کردن یک مجوز X.509. کلاس‌های **System.Security.Cryptography.X509Certificates.X509Certificate** و **System.Security.Cryptography.X509Certificates.X509Certificate2** را ببینید.
- شروع یک پردازش جدید تحت یک اکانت کاربری خاص. کلاس‌های **System.Diagnostics.Process** و **System.Diagnostics.ProcessInfo** را ببینید.
- ساخت یک event log session: کلاس **System.Diagnostics.Eventing.Reader.EventLogSession** را ببینید.
- استفاده از کنترل **System.Windows.Controls.PasswordBox**. ویژگی **SecurePassword** از این کلاس را ببینید.

سرانجام، شما می‌توانید متدهای خودتان را بسازید که یک پارامتر شی **SecureString** دریافت کنند. درون متدتان باید شی **SecureString** یک بافر حافظه مدیریت نشده بسازد که قبل از آنکه متدتان از بافر استفاده کند، حاوی کاراکترهای رمزگشایی شده باشد. برای آنکه دامنه شانس کد مخرب در دسترسی به داده‌های حساس را تا حد ممکن کوچکتر کنید، کد شما باید به رشته‌ی رمزگشایی شده برای مدت زمان تا حد ممکن کوتاه، نیاز به دسترسی داشته باشد. وقتی استفاده از رشته پایان یافت، کد شما باید بافر را صفر کرده و به محض ممکن بافر را آزاد کند. همچنین، هرگز محتویات یک **SecureString** را در یک **String** قرار ندهید: اگر این کار را بکنید، **String** به صورت رمزنگاری نشده در هیپ باقی می‌ماند و کاراکترهایش تا زمانی که حافظه پس از جمع آوری مجدداً استفاده نشود، صفر نمی‌گردد. کلاس **SecureString**، مخصوصاً متد **ToString** را بازنویسی نمی‌کند تا از در معرض گذاشتن اطلاعات حساس جلوگیری کند (که تبدیل به یک **String**، این کار را می‌کند).

کد نمونه زیر نحوه‌ی مقداردهی اولیه و استفاده از یک **SecureString** را نشان می‌دهد (هنگام کامپایل این کد، شما نیاز خواهید داشت سویچ **unsafe** را برای کامپایلر سی‌شارپ تعیین کنید):

```
using System;
using System.Security;
using System.Runtime.InteropServices;

public static class Program {
    public static void Main() {
        using (SecureString ss = new SecureString()) {
            Console.WriteLine("Please enter password: ");
            while (true) {
                ConsoleKeyInfo cki = Console.ReadKey(true);
                if (cki.Key == ConsoleKey.Enter) break;

                // Append password characters into the SecureString
                ss.AppendChar(cki.KeyChar);
                Console.WriteLine("*");
            }
        }
    }
}
```

```

    }
    Console.WriteLine();

    // Password entered, display it for demonstration purposes
    DisplaySecureString(ss);
}
// After 'using', the SecureString is Disposed; no sensitive data in memory
}

// This method is unsafe because it accesses unmanaged memory
private unsafe static void DisplaySecureString(SecureString ss) {
    Char* pc = null;
    try {
        // Decrypt the SecureString into an unmanaged memory buffer
        pc = (Char*) Marshal.SecureStringToCoTaskMemUnicode(ss);

        // Access the unmanaged memory buffer that
        // contains the decrypted SecureString
        for (Int32 index = 0; pc[index] != 0; index++)
            Console.write(pc[index]);
    }
    finally {
        // Make sure we zero and free the unmanaged memory buffer that contains
        // the decrypted SecureString characters
        if (pc != null)
            Marshal.ZeroFreeCoTaskMemUnicode((IntPtr) pc);
    }
}
}
}

```

کلاس **System.Runtime.InteropServices.Marshal** پنج متد ارائه می‌کند که شما می‌توانید برای رمزگشایی کاراکترهای یک **SecureString** به درون یک بافر حافظه مدیریت نشده، از آن‌ها استفاده کنید. تمام این متدها استاتیک هستند، همه یک آرگومان **SecureString** می‌گیرند و همه یک **IntPtr** برمی‌گردانند. هر کدام از این متدها یک متد متناظر دارند که برای صفر کردن بافر داخلی و آزاد نمودن آن باید آن‌ها را فراخوانی کنید. جدول ۴-۱۴ متدهای کلاس **System.Runtime.InteropServices.Marshal** را که برای رمزگشایی یک **SecureString** به درون یک بافر حافظه استفاده می‌شوند و متد متناظر آن‌ها را که برای صفر کردن و آزاد کردن بافر است را نشان می‌دهد:

جدول ۴-۱۴ متدهای کلاس **Marshal** برای کار با رشته‌های امن

متد برای صفر کردن و آزاد کردن بافر

ZeroFreeBSTR

ZeroFreeCoTaskMemAnsi

ZeroFreeCoTaskMemUnicode

ZeroFreeGlobalAllocAnsi

ZeroFreeGlobalAllocUnicode

متد برای رمزگشایی **SecureString** به بافر

SecureStringtoBSTR

SecureStringToCoTaskMemAnsi

SecureStringToCoTaskMemUnicode

SecureStringToGlobalAllocAnsi

SecureStringToGlobalAllocUnicode

فصل ۱۵: نوع های شمارشی و پرچم های بیتی

در این فصل، من نوع های شمارشی و پرچم های بیتی را بحث می کنم. چون ویندوز میکروسافت و بسیاری زبان های برنامه نویسی از این ساختارها برای سال ها استفاده می کنند من مطمئنم بسیاری از شما با نحوه ی استفاده از نوع های شمارشی و پرچم های بیتی آشنا هستید. هرچند، CLR و کتابخانه کلاس فریمورک (FCL) با همدیگر همکاری می کنند تا نوع های شمارشی و پرچم های بیتی را به نوع های واقعی شی گرا تبدیل کنند تا ویژگی های جدیدی ارائه کنند که گمان می کنم اکثر برنامه نویسان با آن ها آشنا نیستند. برایم شگفت انگیز است که چگونه این ویژگی های جدید، که بحث این فصل هستند، برنامه نویسی برنامه ها را به مراتب آسانتر می کنند.

نوع های شمارشی

یک نوع شمارشی `enumerated type` یک نوع است که مجموعه ای از جفت های نماد و مقدار را تعریف می کند. برای نمونه، نوع `Color` که در اینجا نشان داده شده است مجموعه ای از نمادها تعریف می کند که هر نماد یک رنگ را شناسایی می کند:

```
internal enum Color {
    White,          // Assigned a value of 0
    Red,            // Assigned a value of 1
    Green,          // Assigned a value of 2
    Blue,           // Assigned a value of 3
    Orange          // Assigned a value of 4
}
```

البته، برنامه نویسان می توانند همیشه برنامه ای بنویسند که از 0 برای نمایش رنگ سفید، از 1 برای نمایش رنگ قرمز و غیر استفاده کند. هر چند، برنامه نویسان نباید اعداد را اینچنین در کدشان قرار دهند و باید به جای آن، از یک نوع شمارشی استفاده کنند، برای حداقل دو دلیل:

- نوع های شمارشی نوشتن، خواندن و نگهداری برنامه را بسیار آسانتر می کنند. با نوع های شمارشی، نام نمادین در سرتاسر کد استفاده می شود و برنامه نویس مجبور نیست به صورت ذهنی معنی هر مقدار (برای مثال، سفید 0 است و برعکس) را حساب کند. همچنین، اگر مقدار عددی یک نماد تغییر کند، کد براحتمی و بدون هیچ تغییری در سورس کد می تواند مجددا کامپایل شود. به علاوه، ابزارهای مستند سازی و دیگر برنامه های کاربردی، مثل یک دیباگر، می توانند نام های نمادین با معنا را به برنامه نویس نشان دهند.
- نوع های شمارشی دارای نوع قوی (`strong type`) هستند. برای نمونه اگر من سعی کنم `Color.Orange` را به متدی که نیاز به یک نوع شمارشی `Fruit` دارد ارسال کنم، کامپایلر خطایی اعلام خواهد کرد.

در دات نت فریمورک میکروسافت، نوع های شمارشی بیش از نمادهایی هستند که فقط، کامپایلر به آن ها اهمیت دهد. نوع های شمارشی با عنوان شهروندان درجه اول در سیستم نوع، رفتار می شوند که اجازه می دهد عملیات های بسیار قدرتمندی که با نوع های شمارشی به سادگی در دیگر محیطها (مثل ++C مدیریت نشده) قابل انجام نیست، صورت پذیرد.

هر نوع شمارشی مستقیما از `System.Enum` مشتق می شود که خودش از `System.ValueType` مشتق می شود که آن هم از `System.Object` مشتق می گردد. پس نوع های شمارشی، نوع های مقداری (بحث شده در فصل ۵ "نوع های اصلی، ارجاعی و شمارشی") هستند و می توانند در فرم های بسته بندی نشده و بسته بندی شده نمایش داده شوند. هر چند، برخلاف دیگر نوع های مقداری، یک نوع شمارشی نمی تواند هیچ گونه متد، ویژگی یا رویداد تعریف کند. هر چند، شما می توانید از ویژگی متدهای گسترشی سی شارپ برای شبیه سازی افزودن متد به یک نوع شمارشی استفاده کنید. بخش "افزودن متد به نوع شمارشی" را در انتهای این فصل نگاه کنید.

وقتی یک نوع شمارشی کامپایل می شود، کامپایلر سی شارپ هر نماد را به یک فیلد ثابت از نوع تبدیل می کند. برای نمونه، کامپایلر با شمارشی (`enum`) `Color` که قبلا نشان داده شد، به گونه ای رفتار می کند که گویا شما کدی شبیه به کد زیر نوشته اید:

```
internal struct Color : System.Enum {
    // Below are public constants defining Color's symbols and values
    public const Color white = (Color) 0;
```

```

public const Color Red = (Color) 1;
public const Color Green = (Color) 2;
public const Color Blue = (Color) 3;
public const Color Orange = (Color) 4;

// Below is a public instance field containing a Color variable's value
// You cannot write code that references this instance field directly
public Int32 value__;
}

```

کامپایلر سی شارپ حقیقتاً این کد را کامپایل نخواهد کرد چون مانع از آن می‌شود که شما یک نوع مشتق شده از نوع خاص **System.Enum** تعریف کنید. هر چند، این تعریف شبه نوع به شما نشان می‌دهد در درون چه رخ می‌دهد. اساساً، یک نوع شمارشی تنها یک ساختار است با تعدادی فیلد ثابت که در آن تعریف شده است و یک فیلد نمونه. فیلدهای ثابت درون متادیتای اسمبلی قرار می‌گیرند و از طریق رفلکشن قابل دسترسی هستند. این یعنی شما می‌توانید تمام نمادها و مقادیر آن‌ها را که با یک نوع شمارشی همراه است را در زمان اجرا بدست آورید. این همچنین یعنی شما می‌توانید یک نماد رشته‌ای را به معادل مقدار عددی آن تبدیل کنید. این عملیات‌ها توسط نوع پایه **System.Enum** برای شما فراهم می‌شود که چندین متد استاتیک و نمونه تعریف می‌کند که می‌توانند روی یک نمونه از یک نوع شمارشی عمل کنند و شما را از دردسر استفاده از رفلکشن رهایی بخشند. من تعدادی از این عملیات‌ها را بحث خواهیم کرد.

مهم نمادهای تعریف شده توسط یک نوع شمارشی مقادیر ثابت هستند. بنابراین وقتی کامپایلر کدی را می‌بیند که به یک نماد نوع شمارشی ارجاع می‌کند، کامپایلر مقدار عددی نماد را در زمان کامپایل جایگزینی می‌کند و این کد دیگر به نوع شمارشی که نماد را تعریف کرده ارجاع نمی‌کند. این یعنی اسمبلی‌ای که نوع شمارشی را تعریف کرده شاید در زمان اجرا مورد نیاز نباشد. آن تنها هنگام کامپایل کردن نیاز بود. اگر شما کدی دارید که به نوع شمارشی ارجاع می‌کند - به جای آنکه فقط ارجاع‌هایی به نمادهای تعریف شده در نوع داشته باشد - اسمبلی حاوی تعریف نوع شمارشی در زمان اجرا نیاز خواهد بود. چون نمادهای نوع شمارشی به جای مقادیر فقط -خواندنی، ثابت هستند، مسائل نسخه‌بندی ممکن است بروز کنند. این مسائل را در بخش "ثابت‌ها" از فصل ۷ "ثابت‌ها و فیلدها" بحث کرده‌ام.

برای نمونه، نوع **System.Enum** یک متد استاتیک به نام **GetUnderlyingType** و نوع **System.Type** یک متد نمونه به نام **GetEnumUnderlyingType** دارند:

```

public static Type GetUnderlyingType(Type enumType); // Defined in System.Enum
public Type GetEnumUnderlyingType(); // Defined in System.Type

```

این متدها، نوع اصلی که برای نگهداری مقدار یک نوع شمارشی استفاده می‌شود را برمی‌گردانند. هر نوع شمارشی یک نوع زیرین دارد که می‌تواند یک **byte**، **sbyte**، **short**، **ushort**، **int** (نوع‌های رایج تر که سی شارپ به صورت پیش فرض انتخاب می‌کند)، **uint**، **long** یا **ulong** باشد. البته این نوع‌های اصلی سی شارپ بر نوع‌های FCL منطبق می‌شوند. هر چند، برای آنکه پیاده‌سازی خود کامپایلر ساده تر شود، کامپایلر سی شارپ نیاز دارد شما نام یک نوع اصلی را در اینجا تعیین کنید؛ استفاده از یک نام نام FCL (مثل **Int32**) باعث تولید پیام زیر می‌شود:

“error CS1008: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected.”

کد زیر نشان می‌دهد چگونه یک نوع شمارشی با یک نوع زیرین **byte** (**System.Byte**) تعریف کنید:

```

internal enum Color : byte {
    White,
    Red,
    Green,
    Blue,
    Orange
}

```

با نوع شمارشی **Color** که بدین گونه تعریف شده است، کد زیر نشان می‌دهد که **GetUnderlyingType** چه چیزی برمی‌گرداند:

```
// The following line displays "System.Byte".
```

```
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

کامپایلر سی شارپ با نوع‌های شمارشی به عنوان نوع‌های اصلی برخورد می‌کند. بهمین خاطر، شما می‌توانید بسیاری از عملگرهای آشنا (<=, <, >, >=, +, -, ^, &, |, ~, ++, --) را برای دستکاری نمونه‌های نوع شمارشی استفاده کنید. تمام این عملگرها در حقیقت روی فیلد نمونه **value** در درون نمونه نوع هر نوع شمارشی، عمل می‌کنند. علاوه بر این، کامپایلر سی شارپ به شما اجازه می‌دهد صریحاً نمونه‌های نوع شمارشی را به یک نوع شمارشی متفاوت تبدیل کنید (**cast**). شما همچنین می‌توانید یک نمونه نوع شمارشی را به یک نوع مقداری تبدیل کنید.

با داشتن یک نمونه از نوع شمارشی، می‌توان آن مقدار را به یکی از چندین نمایش رشته‌ای با فراخوانی **ToString** به ارث برده شده از **System.Enum**، منطبق کرد:

```
Color c = Color.Blue;
Console.WriteLine(c); // "Blue" (General format)
Console.WriteLine(c.ToString()); // "Blue" (General format)
Console.WriteLine(c.ToString("G")); // "Blue" (General format)
Console.WriteLine(c.ToString("D")); // "3" (Decimal format)
Console.WriteLine(c.ToString("X")); // "03" (Hex format)
```

نکته هنگام استفاده از فرمت هگز، **ToString** همیشه حروف بزرگ تولید می‌کند. به علاوه، تعداد ارقام خروجی بستگی به نوع زیرین نوع شمارشی دارد: ۲ رقم برای **byte/sbyte**، ۴ رقم برای **short/ushort**، ۸ رقم برای **int/uint** و ۱۶ رقم برای **long/ulong**. صفرهای آغازین در صورت نیاز تولید می‌شوند.

علاوه بر متد **ToString**، نوع **System.Enum** یک متد استاتیک **Format** ارائه می‌کند که شما می‌توانید برای فرمت کردن مقدار یک نوع شمارشی آنرا فراخوانی کنید:

```
public static String Format(Type enumType, Object value, String format);
```

به طور کلی، من ترجیح می‌دهم متد **ToString** را فراخوانی کنم چون نیاز به کد کمتری داشته و فراخوانی آن آسانتر است. اما استفاده از **Format** یک مزیت بر استفاده از **ToString** دارد: **Format** به شما اجازه می‌دهد یک مقدار عددی را برای پارامتر مقدار اسال کنید؛ شما مجبور نیستید که یک نمونه از نوع شمارشی داشته باشید. برای نمونه، کد زیر "Blue" را نمایش می‌دهد:

```
// The following line displays "Blue".
Console.WriteLine(Enum.Format(typeof(Color), 3, "G"));
```

نکته این امکان هست که، یک نوع شمارشی که چندین نماد با یک مقداری عددی دارد، تعریف کنید. هنگام تبدیل یک مقدار عددی به یک نماد با استفاده از فرمت کلی، متدهای **Enum** یکی از نمادها را برمی‌گردانند. هر چند، تضمینی وجود ندارد که کدام نام نماد برگردانده شود. همچنین اگر برای مقدار عددی که شما به دنبال آن هستید، هیچ نمادی تعریف نشده باشد، یک رشته حاوی مقداری عددی، برگردانده می‌شود.

همچنین این امکان هست که متد استاتیک **GetValues** از **System.Enum** یا متد نمونه **GetEnumerator** از **System.Type** را جهت بدست آوردن یک آرایه که حاوی یک عنصر به ازای هر نام نمادین در یک نوع شمارشی است را فراخوانی کنید؛ هر عنصر حاوی مقدار عددی متعلق به نام نمادین است:

```
public static Array GetValues(Type enumType); // Defined in System.Enum
public Array GetEnumerator(); // Defined in System.Type
```

با استفاده از این متد همراه با متد **ToString**، شما می‌توانید تمام نمادها و مقادیر عددی از یک نوع شمارشی را نمایش دهید، همانند:

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n-----\t-----");
foreach (Color c in colors) {
```



```
// Display each symbol in Decimal and General format.
Console.WriteLine("{0,5:D}\t{0:G}", c);
}
```

کد قبلی، خروجی زیر را تولید می‌کند:

Number of symbols defined: 5

```
Value Symbol
-----
0 White
1 Red
2 Green
3 Blue
4 Orange
```

این بحث، تعدادی از عملیات‌های عالی که می‌تواند روی نوع‌های شمارشی اجرا شوند را نشان می‌دهد. من گمان می‌کنم متد `ToString` با فرمت کلی، بسیار زیاد در نمایش نام‌های نمادینی در عناصر رابط گرافیکی یک برنامه (`list box`، `combo box` و همانند این‌ها) تا زمانی که رشته‌ها نیاز به محلی شدن ندارند، استفاده می‌شود. (چون نوع‌های شمارشی از محلی سازی پشتیبانی نمی‌کنند). علاوه بر متد `GetValues`، نوع `System.Enum` و نوع `System.Type` متدهای زیر که نمادهای نوع شمارشی را برمی‌گرداند را ارائه می‌کنند:

```
// Returns a String representation for the numeric value
public static String GetName(Type enumType, Object value); // Defined in System.Enum
public String GetEnumName(Object value); // Defined in System.Type
```

```
// Returns an array of Strings: one per symbol defined in the enum
public static String[] GetNames(Type enumType); // Defined in System.Enum
public String[] GetEnumNames(); // Defined in System.Type
```

من متدهای زیادی را بحث کردم که شما می‌توانید برای یافتن نماد یک نوع شمارشی از آن‌ها استفاده کنید. اما شما همچنین نیاز به یک متد دارید که بتوانید مقدار معادل یک نماد را بدست آورید، برای مثال، عملیاتی که می‌تواند برای تبدیل یک نماد که یک کاربر در یک `text box` وارد می‌کند استفاده شود. تبدیل یک نماد به یک نمونه از نوع شمارشی به سادگی از طریق متدهای استاتیک `Parse` و `TryParse` میسر است:

```
public static Object Parse(Type enumType, String value);
public static Object Parse(Type enumType, String value, Boolean ignoreCase);
public static Boolean TryParse<TEnum>(String value, out TEnum result) where TEnum: struct;
public static Boolean TryParse<TEnum>(String value, Boolean ignoreCase, out TEnum result)
    where TEnum : struct;
```

کد زیر نحوه استفاده از این متد را نشان می‌دهد:

```
// Because Orange is defined as 4, 'c' is initialized to 4.
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);
```

```
// Because Brown isn't defined, an ArgumentException is thrown.
c = (Color) Enum.Parse(typeof(Color), "Brown", false);
```

```
// Creates an instance of the Color enum with a value of 1
Enum.TryParse<Color>("1", false, out c);
```

```
// Creates an instance of the Color enum with a value of 23
Enum.TryParse<Color>("23", false, out c);
```

سرانجام، با استفاده از متد استاتیک `IsDefined` از `Enum` و متد `IsEnumDefined` از `Type`:

```
public static Boolean IsDefined(Type enumType, Object value); // Defined in System.Enum
```

```

public Boolean IsEnumDefined(Object value); // Defined in System.Type
// شما می‌توانید تعیین کنید آیا یک مقدار عددی برای یک نوع شمارشی مجاز است یا خیر:
// Displays "True" because Color defines Red as 1
Console.WriteLine(Enum.IsDefined(typeof(Color), 1));

// Displays "True" because Color defines white as 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));

// Displays "False" because a case-sensitive check is performed
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));

// Displays "False" because Color doesn't have a symbol of value 10
Console.WriteLine(Enum.IsDefined(typeof(Color), 10));

```

متد **IsDefined** اغلب برای اعتبار سنجی یک پارامتر استفاده می‌شود. یک مثال در اینجا آمده است:

```

public void SetColor(Color c) {
    if (!Enum.IsDefined(typeof(Color), c)) {
        throw(new ArgumentOutOfRangeException("c", c, "Invalid color value."));
    }
    // Set color to white, Red, Green, Blue, or Orange
    ...
}

```

اعتبار سنجی پارامتر می‌تواند مفید باشد چون فردی می‌تواند **SetColor** را اینچنین فراخوانی کند:

```
SetColor((Color) 547);
```

چون هیچ نمادی متناظر با مقدار **547** وجود ندارد، متد **SetColor** یک اکسپشن **ArgumentOutOfRangeException** تولید می‌کند که بیان می‌کند پارامتر غیر معتبر است و چرا.

مهم متد **IsDefined** خیلی راحت و آسان است اما شما باید با احتیاط از آن استفاده کنید. اول اینکه، **IsDefined** همیشه جستجوی حساس به بزرگی و کوچکی حروف انجام می‌دهد، و راهی وجود ندارد که جستجو بدون این حساسیت انجام پذیرد. دوم اینکه، **IsDefined** به شدت کند است چون در درون از رفلکشن استفاده می‌کند. اگر شما دستی کدی بنویسید که هر مقدار ممکن را بررسی کند، عملکرد برنامه مطمئناً بهتر خواهد شد. سوم اینکه، شما باید تنها در صورتی از **IsDefined** استفاده کنید که خود نوع شمارشی در همان اسمبلی‌ای تعریف شده باشد که **IsDefined** را فراخوانی می‌کند. دلیل اینست: فرض کنید شمارشی **Color** در یک اسمبلی و متد **SetColor** در اسمبلی دیگری تعریف شده‌اند. متد **SetColor**، **IsDefined** را فراخوانی می‌کند، و اگر رنگ، **White**، **Red**، **Green**، **Blue** یا **Orange** باشد، **SetColor** کارش را انجام می‌دهد. اما، اگر شمارشی **Color** در آینده، **Purple** را اضافه کند، **SetColor** به **Purple** نیز اجازه خواهد داد، که قبلاً اصلاً انتظار آن نمی‌رفت و متد ممکن است با نتایج غیر قابل پیش‌بینی اجرا شود.

سرانجام، نوع **System.Enum** یک مجموعه متدهای استاتیک **ToObject** ارائه می‌کند که یک نمونه از یک **Byte**، **SByte**، **Int16**، **UInt16**، **Int32**، **UInt32** یا **UInt64** را به یک نمونه از یک نوع شمارشی تبدیل می‌کند.

نوع‌های شمارشی همیشه همراه با دیگر نوع‌ها استفاده می‌شوند. نوعاً آن‌ها برای نوع پارامترهای متد یا نوع برگشتی، ویژگی‌ها و فیلدها استفاده می‌شوند. یک سوال رایج اینست که آیا نوع شمارشی را درون نوعی که بدان نیاز دارد یا در همان سطح نوعی که بدان نیاز دارد تعریف کنیم. اگر شما **FCL** را بررسی کنید، شما خواهید دید که یک نوع شمارشی معمولاً در همان سطح کلاسی که بدان نیاز دارد، تعریف شده است. دلیل اینست که زندگی یک برنامه‌نویس را با کاهش حجم تایپ کردن، کمی آسانتر می‌کند. پس شما باید نوع‌های شمارشی خود را در همان سطح تعریف کنید مگر آنکه نگران تداخل نام‌ها باشید.

پرچم های بیتی

برنامه‌نویسان، اغلب با مجموعه‌های پرچم‌های بیتی کار می‌کنند. وقتی شما متد **GetAttributes** از نوع **System.IO.File** را فراخوانی می‌کنید، آن، یک نمونه از یک نوع **FileAttributes** برمی‌گرداند. یک نوع **FileAttributes** یک نمونه از یک نوع شمارشی بر پایه **Int32** است که در آن هر بیت یک صفت از فایل را منعکس می‌کند. نوع **FileAttributes** در **FCL**، اینگونه تعریف شده است:

```
[Flags, Serializable]
public enum FileAttributes {
    ReadOnly = 0x0001,
    Hidden = 0x0002,
    System = 0x0004,
    Directory = 0x0010,
    Archive = 0x0020,
    Device = 0x0040,
    Normal = 0x0080,
    Temporary = 0x0100,
    SparseFile = 0x0200,
    ReparsePoint = 0x0400,
    Compressed = 0x0800,
    Offline = 0x1000,
    NotContentIndexed = 0x2000,
    Encrypted = 0x4000
}
```

برای آنکه تعیین کنید آیا یک فایل مخفی است، شما کدی شبیه به این اجرا می‌کنید:

```
String file = Assembly.GetEntryAssembly().Location;
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file, (attributes & FileAttributes.Hidden) != 0);
```

نکته کلاس **Enum** یک متد **HasFlag** همانند زیر تعریف می‌کند:

```
public Boolean HasFlag(Enum flag);
```

با استفاده از این متد، شما می‌توانید فراخوانی به **Console.WriteLine** را اینگونه بازنویسی کنید:

```
Console.WriteLine("Is {0} hidden? {1}", file,
    attributes.HasFlag(FileAttributes.Hidden));
```

هر چند، من توصیه می‌کنم از متد **HasFlag** به این دلیل خودداری کنید: چون آن، یک پارامتر از نوع **Enum** می‌گیرد، هر مقداری که بدان ارسال می‌کنید باید بسته بندی شود که نیاز به تخصیص مجدد حافظه دارد.

و کد زیر نشان می‌دهد چگونه صفت‌های یک فایل را به فقط-خواندنی و مخفی تغییر دهید:

```
File.SetAttributes(file, FileAttributes.ReadOnly | FileAttributes.Hidden);
```

همانگونه که نوع **FileAttributes** نشان می‌دهد، استفاده از نوع‌های شمارشی برای بیان یک مجموعه از پرچم‌های بیتی که قابل ترکیب باشند رایج است. هر چند، اگرچه نوع‌های شمارشی و پرچم‌های بیتی مشابه هستند، معانی دقیقاً یکسانی ندارند. برای نمونه، نوع‌های شمارشی مقادیر عددی تکی را نمایش می‌دهند و پرچم‌های بیتی یک مجموعه از بیت‌ها که بعضی روشن و بعضی خاموش هستند را نمایش می‌دهند. هنگام تعریف یک نوع شمارشی که برای شناسایی پرچم‌های بیتی نیز استفاده می‌شود، شما باید صریحاً یک مقدار عددی به هر نماد اختصاص دهید. معمولاً، یک تک بیت از هر نماد روشن می‌شود. همچنین رایج است که یک نماد به نام **None** با یک مقدار **0** ببینید، و شما می‌توانید نمادهایی تعریف کنید که ترکیب‌های پر استفاده را نمایش

می‌دهند (مثل نماد **ReadWrite** در زیر). همچنین قویا توصیه می‌شود که شما صفت سفارشی **System.FileAttribute** را بر نوع شمارشی همانند زیر اعمال کنید:

```
[Flags] // The C# compiler allows either "Flags" or "FlagsAttribute".
```

```
internal enum Actions {
    None = 0
    Read = 0x0001,
    Write = 0x0002,
    ReadWrite = Actions.Read | Actions.Write,
    Delete = 0x0004,
    Query = 0x0008,
    Sync = 0x0010
}
```

چون **Actions** یک نوع شمارشی است، شما می‌توانید از تمام متدهای توضیح داده شده در بخش قبلی هنگام کار با نوع‌های شمارشی پرچم بیتی استفاده کنید. هر چند، خوب می‌شد اگر این توابع کمی متفاوت عمل می‌کردند. برای مثال، فرض کنید شما کد زیر را دارید:

```
Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString()); // "Read, Delete"
```

وقتی **ToString** فراخوانی می‌شود، سعی می‌کند مقدار عددی را به معادل نمادین خود تبدیل کند. مقدار عددی، **0x0005** است که معادل نمادین ندارد. هر چند، **ToString** می‌بیند صفت **[Flags]** روی نوع **Actions** وجود دارد و اکنون **ToString** با مقدار عددی نه به عنوان یک مقدار بلکه به عنوان مجموعه‌ای از پرچم‌های بیتی رفتار می‌کند. چون بیت‌های **0x0001** و **0x0004** تنظیم شده اند، **ToString** رشته زیر را تولید می‌کند: "Read, Delete". اگر شما صفت **[Flags]** را از نوع **Actions** حذف کنید، **ToString**، "5" برمی‌گرداند. من متد **ToString** را در بخش قبلی بحث کردم و نشان دادم که سه روش برای فرمت کردن خروجی ارائه می‌کند: "G" (کلی general)، "D" (دهدهی decimal) و "X" (هگز hex). وقتی شما یک نمونه از یک نوع شمارشی را با استفاده از فرمت کلی، فرمت می‌کنید، ابتدا بررسی می‌شود آیا بر نوع، صفت **[Flags]** اعمال شده است یا خیر. اگر این صفت اعمال نشده باشد، یک نماد که با مقدار عددی مطابقت دارد جستجو شده و برگردانده می‌شود. اگر صفت **[Flags]** اعمال شده است، **ToString** اینگونه کار می‌کند:

۱. مجموعه مقادیر عدد تعریف شده توسط نوع شمارشی بدست می‌آید و اعداد به ترتیب نزولی مرتب می‌شوند.
۲. هر مقدار عددی با مقدار درون نمونه شمارشی **AND** بیتی می‌شود و اگر حاصل، معادل مقدار عددی شود، رشته‌ای که با مقدار عددی همراه است به رشته خروجی اضافه می‌شود و این بیت‌ها حساب شده لحاظ می‌شوند و خاموش می‌گردند. این مرحله تا زمانی که تمام مقادیر عددی بررسی شوند یا تمام بیت‌های نوع شمارشی خاموش شوند، تکرار می‌شود.
۳. اگر پس از آنکه تمام مقادیر عددی بررسی شدند، نمونه شمارشی هنوز **0** نباشد، نمونه شمارشی دارای بیت‌های روشنی است که منطبق بر هیچ نماد تعریف شده‌ای نیستند. در این حالت، **ToString** عدد اصلی در نمونه شمارشی را به عنوان یک رشته برمی‌گرداند.
۴. اگر مقدار اریجینال نمونه شمارشی **0** نبود، رشته‌ای از نمادهایی که با کاما از هم جدا شده‌اند، برگردانده می‌شود.
۵. اگر مقدار اصلی نمونه شمارشی **0** بود و اگر نوع شمارشی دارای یک نماد تعریف شده با مقدار متناظر **0** بود، نماد برگردانده می‌شود.
۶. اگر ما به این مرحله رسیدیم، "0" برگردانده می‌شود.

اگر شما ترجیح می‌دهید، شما می‌توانستید نوع **Actions** را بدون صفت **[Flags]** تعریف کنید و هنوز هم با استفاده از فرمت "F" رشته صحیح را به دست آورید:

```
// [Flags] // Commented out now
internal enum Actions {
    None = 0
    Read = 0x0001,
    Write = 0x0002,
    ReadWrite = Actions.Read | Actions.Write,
    Delete = 0x0004,
```

```

    Query = 0x0008,
    Sync = 0x0010
}

```

```

Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString("F")); // "Read, Delete"

```

اگر مقدار عددی دارای بیتی باشد که قابل انطباق به یک نماد نباشد، رشته برگردانده شده تنها شامل یک عدد دهدهی خواهد بود که مقدار عددی اصلی را نشان می‌دهد، هیچ نمادی در رشته ظاهر نخواهد شد.

دقت کنید نمادهایی که شما در نوع شمارشی خود تعریف می‌کنید به اجبار نباید توان‌های ۲ باشند. برای نمونه، نوع **Actions** می‌توانست یک نماد به نام **All** با یک مقدار **0x001F** تعریف کند. اگر یک نمونه از نوع **Actions** دارای یک مقدار **0x001F** باشد، فرمت کردن نمونه، رشته ای تولید می‌کند که حاوی "All" است. دیگر نمادها ظاهر نخواهند شد.

تاکنون، من بحث کردم چگونه مقادیر عددی را به رشته‌ای از پرچم‌ها تبدیل کنید. همچنین این امکان وجود دارد که یک رشته از نمادهایی که با کاما جدا شده‌اند را به یک مقدار عددی با فراخوانی متد استاتیک **Parse** و **TryParse** از **Enum** تبدیل کنید. کد زیر نحوه‌ی استفاده از این متد را نشان می‌دهد:

```

// Because Query is defined as 8, 'a' is initialized to 8.
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);
Console.WriteLine(a.ToString()); // "Query"

```

```

// Because Query and Read are defined, 'a' is initialized to 9.
Enum.TryParse<Actions>("Query, Read", false, out a);
Console.WriteLine(a.ToString()); // "Read, Query"

```

```

// Creates an instance of the Actions enum with a value of 28
a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString()); // "Delete, Query, Sync"

```

وقتی **Parse** و **TryParse** فراخوانی می‌شوند، در درون عملیات‌های زیر انجام می‌پذیرد:

۱. آن، تمام کاراکترهای فضای خالی را از ابتدا و انتهای رشته پاک می‌کند.
۲. اگر اولین کاراکتر رشته یک رقم، علامت جمع (+)، یا علامت تفریق (-) باشد، رشته یک عدد فرض می‌شود و یک نمونه شمارشی که مقدار عددی آن برابر رشته‌ای است که به معادل عددی‌اش تبدیل شده را برمی‌گرداند.
۳. رشته ارسالی به مجموعه‌ای از نشانه‌ها (ی با کاما جدا شده) تبدیل می‌گردد و تمام فضاهای خالی از هر نشانه حذف می‌گردد.
۴. هر رشته نشانه در نمادهای تعریف شده توسط نوع شمارشی، جستجو می‌شود. اگر نماد یافت نشد، **Parse** یک **System.ArgumentException** تولید کرده و **TryParse** **false** برمی‌گرداند. اگر نماد یافت شد، مقدار عددی آن را با نتیجه جاری **OR** بیتی کرده و سپس علامت بعدی را جستجو می‌کند.
۵. اگر تمام نشانه‌ها یافت شدند، نتیجه جاری را برمی‌گرداند.

شما هرگز نباید از متد **IsDefined** با نوع‌های شمارشی پرچم بیتی استفاده کنید. آن به دو دلیل کار نخواهد کرد:

- اگر شما یک رشته به **IsDefined** ارسال کنید، رشته را به نشانه‌های مجزا برای جستجو تقسیم نمی‌کند بلکه سعی می‌کند رشته را همانند یک نماد بزرگ با کاماهای درون آن جستجو کند. چون شما نمی‌توانید یک نوع شمارشی با یک نماد که حاوی کاما است، تعریف کنید پس نماد هرگز یافت نخواهد شد.
- اگر شما یک مقدار عددی به **IsDefined** ارسال کنید، بررسی می‌کند آیا نوع شمارشی، یک نماد که مقدار عددی‌اش با عدد ارسالی تطابق دارد، یافت می‌شود یا خیر. چون این برای پرچم‌های بیتی بعید است، **IsDefined** معمولاً **false** برمی‌گرداند.

افزودن متد به نوع های شمارشی

قبلا در این فصل، من اشاره کردم که شما نمی‌توانید یک متد را به عنوان بخشی از یک نوع شمارشی تعریف کنید. و برای سال‌ها، این مرا غمگین می‌کرد چون مواقع بسیاری وجود دارد که من دوست دارم بتوانم متدهایی برای نوع‌های شمارشی خودم تعریف کنم. خوشبختانه اکنون من می‌توانم از ویژگی نسبتا جدید متدهای گسترشی سی‌شارپ (بحث شده در فصل ۸ "متدها") برای شبیه سازی افزودن متد به یک نوع شمارشی استفاده کنم.

اگر من بخواهم تعدادی متد به نوع شمارشی **FileAttributes** اضافه کنم، من می‌توانم یک کلاس استاتیک با متدهای گسترشی، مثل زیر تعریف کنم:

```
internal static class FileAttributesExtensionMethods {
    public static Boolean IsSet(this FileAttributes flags, FileAttributes flagToTest) {
        if (flagToTest == 0)
            throw new ArgumentOutOfRangeException("flagToTest", "value must not be 0");
        return (flags & flagToTest) == flagToTest;
    }

    public static Boolean IsClear(this FileAttributes flags, FileAttributes flagToTest) {
        if (flagToTest == 0)
            throw new ArgumentOutOfRangeException("flagToTest", "value must not be 0");
        return !IsSet(flags, flagToTest);
    }

    public static Boolean AnyFlagsSet(this FileAttributes flags, FileAttributes testFlags)
    {
        return ((flags & testFlags) != 0);
    }

    public static FileAttributes Set(this FileAttributes flags, FileAttributes setFlags) {
        return flags | setFlags;
    }

    public static FileAttributes Clear(this FileAttributes flags,
        FileAttributes clearFlags) {
        return flags & ~clearFlags;
    }

    public static void ForEach(this FileAttributes flags,
        Action<FileAttributes> processFlag) {
        if (processFlag == null) throw new ArgumentNullException("processFlag");
        for (UInt32 bit = 1; bit != 0; bit <<= 1) {
            UInt32 temp = ((UInt32)flags) & bit;
            if (temp != 0) processFlag((FileAttributes)temp);
        }
    }
}
```

و در اینجا کدی را می‌بینید که فراخوانی تعدادی از این متدها را نشان می‌دهد. همانگونه که می‌بینید، کد به گونه می‌باشد که گویا من دارم متد را روی نوع شمارشی فراخوانی می‌کنم:

```
FileAttributes fa = FileAttributes.System;
fa = fa.Set(FileAttributes.ReadOnly);
```

३०१

```
fa = fa.Clear(FileAttributes.System);  
fa.ForEach(f => Console.WriteLine(f));
```

فصل ۱۶: آرایه ها

آرایه‌ها مکانیزی هستند که به شما اجازه می‌دهند با چندین آیتیم به عنوان یک مجموعه رفتار کنید. CLR از آرایه‌های تک بعدی، چند بعدی و آرایه‌های دندانه‌دار jagged (یعنی، آرایه‌ای از آرایه‌ها) پشتیبانی می‌کند. تمام نوع‌های آرایه به صورت ضمنی از کلاس خلاصه‌ی **System.Array** مشتق شده‌اند که خودش از **System.Object** مشتق شده است. این یعنی آرایه‌ها همیشه نوع‌های ارجاعی هستند که در هیپ مدیریت شده تخصیص می‌یابند و اینکه متغیر یا فیلد برنامه‌ی شما حاوی یک ارجاع به آرایه است نه به عناصر خود آرایه. کد زیر این مطلب را روشن می‌کند:

```
Int32[] myIntegers;           // Declares a reference to an array
myIntegers = new Int32[100]; // Creates an array of 100 Int32s
```

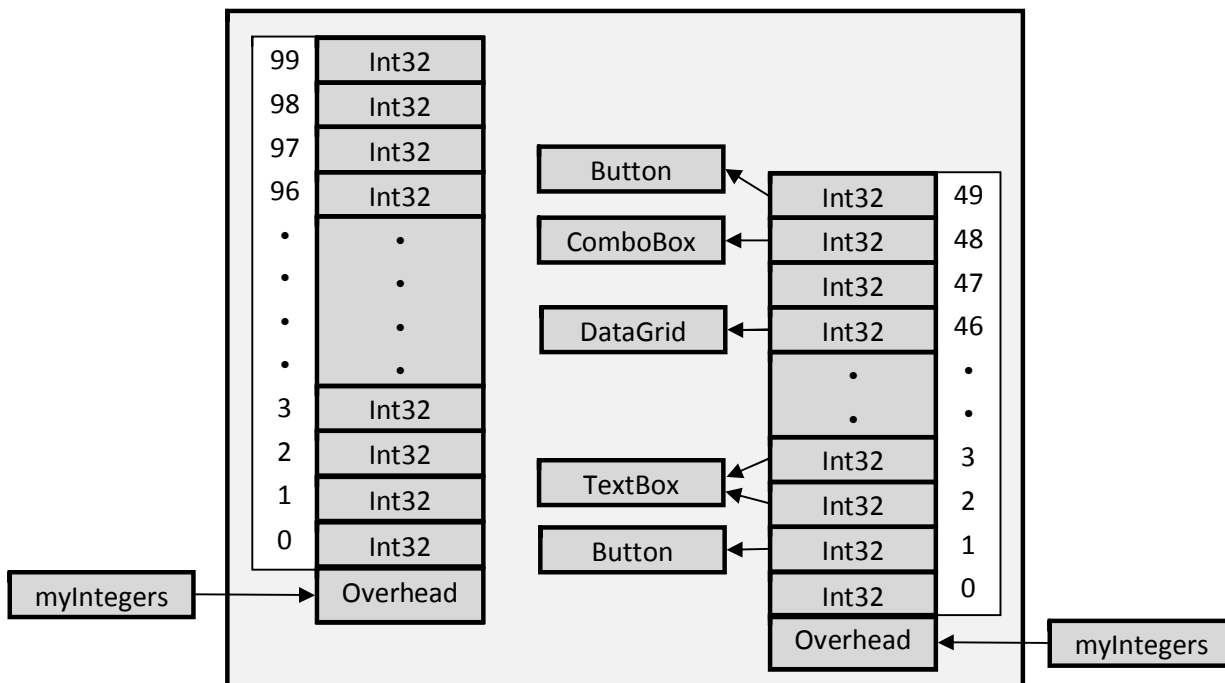
در اولین خط، **myIntegers** یک متغیر است که قادر است به یک آرایه تک بعدی از **Int32** ها اشاره کند. در ابتدا، **myIntegers** به **null** تنظیم می‌شود چون من یک آرایه تخصیص نداده‌ام. خط دوم از کد، یک آرایه از ۱۰۰ مقدار **Int32** تخصیص می‌دهد که همه‌ی **Int32** ها به 0 مقداردهی اولیه می‌گردند. چون آرایه‌ها نوع‌های ارجاعی هستند، بلوک حافظه که برای نگهداری ۱۰۰ عدد **Int32** بسته‌بندی نشده نیاز است در هیپ مدیریت شده تخصیص می‌یابد. در حقیقت، علاوه بر عناصر آرایه، بلوک حافظه اشغال شده توسط شی آرایه، حاوی یک اشاره‌گر شی نوع، یک اندیس بلوک همزمانی و تعدادی اعضای اضافی نیز است. آدرس این بلوک حافظه برگردانده می‌شود و در متغیر **myIntegers** ذخیره می‌گردد.

شما همچنین می‌توانید آرایه‌هایی از نوع‌های ارجاعی بسازید:

```
Control[] myControls;           // Declares a reference to an array
myControls = new Control[50]; // Creates an array of 50 Control references
```

در اولین خط، **myControls** یک متغیر است که قادر است به یک آرایه تک بعدی از اشاره‌گرهای **Control** اشاره کند. در ابتدا، **myControls** به **null** تنظیم می‌شود چون من آرایه‌ای تخصیص نداده‌ام. خط دوم، یک آرایه از ۵۰ اشاره‌گر **Control** تخصیص می‌دهد که تمام این ارجاعات به **null** مقداردهی اولیه می‌شوند. چون **Control** یک نوع ارجاعی است، ساخت آرایه، تنها تعدادی اشاره‌گر ایجاد می‌کند، اشیاء واقعی در این زمان ساخته نمی‌شوند. آدرس این بلوک حافظه برگردانده می‌شود و در متغیر **myControls** ذخیره می‌گردد.

شکل ۱-۱۶ نشان می‌دهد آرایه‌ای از نوع‌های مقداری و آرایه‌ای از نوع‌های ارجاعی چگونه در هیپ به نظر می‌رسند:



شکل ۱-۱۶ آرایه‌هایی از نوع‌های مقداری و ارجاعی در هیپ مدیریت شده

در شکل، آرایه **Controls** نتیجه‌ای را نشان می‌دهد که پس از اجرای خطوط زیر حاصل می‌شود:

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2]; // Two elements refer to the same object.
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

مشخصات مشترک زبان (CLS) نیاز دارد تمام آرایه‌ها پایه صفر باشند. این اجازه می‌دهد یک متد نوشته شده در سی‌شارپ یک آرایه بسازد و اشاره‌گر به آرایه را به کد نوشته شده در زبان دیگری مثل ویژوال بیسیک دات‌نت ارسال کند. به علاوه، چون آرایه‌های پایه صفر تا حد زیادی رایج‌ترین آرایه‌ها هستند، مایکروسافت زمان زیادی برای بهبود عملکرد آن‌ها صرف کرده است. هر چند، CLR از آرایه‌های غیر پایه صفر نیز پشتیبانی می‌کند گرچه استفاده از آن‌ها توصیه نمی‌شود. برای آن دسته از شمایی که به کاهش اندک سرعت یا مشکلات انتقال بین زبان‌ها اهمیت نمی‌دهید، من نحوه‌ی ساخت و استفاده از آرایه‌های غیر پایه صفر را بعداً در این فصل خواهیم گفت.

توجه کنید در شکل ۱-۱۶ هر آرایه تعدادی اطلاعات اضافی همراه با خودش دارد. این اطلاعات درجه هر آرایه (تعداد ابعاد)، حد پایین برای هر بعد از آرایه (تقریباً همیشه 0)، و طول هر بعد از آرایه را شامل می‌شود. اطلاعات اضافی همچنین حاوی نوع عناصر آرایه است. من بعداً در این فصل به متدهایی اشاره خواهیم کرد که به شما اجازه می‌دهند به این اطلاعات اضافی دسترسی یابید.

تاکنون، من مثال‌هایی را نشان دادم تا نمایش دهم چگونه آرایه‌های تک بعدی بسازید. هر وقت ممکن است، شما باید از آرایه‌های تک بعدی پایه صفر (zero-based) که گاهی با عنوان آرایه‌های SZ یا بردارها اطلاق می‌شوند، استفاده کنید. بردارها بهترین عملکرد را دارند چون شما می‌توانید از دستورات خاص زبان میانی (IL) مثل `newarr`، `ldelem`، `ldelem`، `ldel` و `stelem` برای دستکاری آن‌ها استفاده کنید. هر چند، اگر ترجیح می‌دهید، شما می‌توانید با آرایه‌های چند بعدی کار کنید. مثال‌هایی از آرایه‌های چند بعدی:

```
// Create a two-dimensional array of Doubles.
Double[,] myDoubles = new Double[10, 20];

// Create a three-dimensional array of String references.
String[, ,] myStrings = new String[5, 3, 10];

CLR از آرایه‌های دندانه‌دار نیز پشتیبانی می‌کند که آرایه‌هایی از آرایه‌ها هستند. آرایه‌های پایه صفر تک بعدی دندانه‌دار همان عملکردی را دارند که بردارهای معمولی دارند. هر چند، دسترسی به عناصر یک آرایه دندانه دار به معنی دو یا بیشتر دسترسی به آرایه است. مثال‌های زیر نحوه ساخت یک آرایه از چند ضلعی‌ها به این صورت که هر چند ضلعی از یک آرایه از نمونه‌های Point تشکیل شده است را نشان می‌دهد:

// Create a single-dimensional array of Point arrays.
Point[][] myPolygons = new Point[3][];

// myPolygons[0] refers to an array of 10 Point instances.
myPolygons[0] = new Point[10];

// myPolygons[1] refers to an array of 20 Point instances.
myPolygons[1] = new Point[20];

// myPolygons[2] refers to an array of 30 Point instances.
myPolygons[2] = new Point[30];

// Display the Points in the first polygon.
for (Int32 x = 0; x < myPolygons[0].Length; x++)
    Console.WriteLine(myPolygons[0][x]);
```

نکته CLR بررسی می‌کند یک اندیس در یک آرایه معتبر باشد. به بیان دیگر، شما نمی‌توانید یک آرایه با ۱۰۰ عنصر درست کنید (از ۰ تا ۹۹ عددگذاری می‌شوند) و سپس سعی کنید به عنصری با اندیس ۵- یا ۱۰۰ دسترسی پیدا کنید. انجام چنین کاری باعث می‌شود یک **System.IndexOutOfRangeException** تولید گردد. اجازه دادن در دسترسی به حافظه‌ی خارج از بازه یک آرایه، نقض امنیت نوع و یک حفره امنیتی احتمالی است و CLR اجازه نمی‌دهد کد قابل‌بازبینی چنین کاری کند. معمولاً، کاهش سرعتی که با بررسی اندیس همراه است آنقدر قابل ملاحظه نیست چون کامپایلر فقط در لحظه (JIT) به صورت عادی حدود آرایه را یکبار قبل از اجرای حلقه بررسی می‌کند به جای آنکه در هر اجرای حلقه این کار را بکند. هر چند، اگر شما هنوز هم نگران کاهش سرعت به خاطر بررسی اندیس توسط CLR هستید، شما می‌توانید از کد ناامن در سی‌شارپ برای دسترسی به آرایه استفاده کنید. بخش "کارایی دسترسی به آرایه" در این فصل نشان می‌دهد چگونه این کار را انجام دهید.

مقداردهی اولیه عناصر آرایه

در بخش قبلی، من نشان دادم چگونه یک شی آرایه بسازید و سپس نشان دادم چگونه عناصر آرایه را مقداردهی اولیه کنید. سی‌شارپ نحوی ارائه می‌کند که به شما اجازه می‌دهد این دو عملیات را در یک عبارت انجام دهید. برای نمونه:

```
String[] names = new String[] { "Aidan", "Grant" };
```

نشانه‌هایی که درون آکولات با کاما از هم جدا شده اند، یک مقداردهی کننده آرایه **array initializer** نامیده می‌شوند. هر نشانه می‌تواند یک عبارت پیچیده دلخواه یا در مورد آرایه چند بعدی، یک مقداردهی کننده آرایه‌ی تودرتو باشد. در مثال بالا، من فقط از دو عبارت ساده‌ی **String** استفاده کردم. اگر شما یک متغیر محلی در یک متد برای اشاره به یک آرایه‌ی مقداردهی اولیه شده تعریف می‌کنید، آنگاه شما می‌توانید از ویژگی متغیر محلی با نوع ضمنی سی‌شارپ (**var**) برای ساده کردن کد استفاده کنید:

```
// Using C#'s implicitly typed local variable feature:
```

```
var names = new String[] { "Aidan", "Grant" };
```

در اینجا، کامپایلر استنتاج می‌کند متغیر محلی **names** باید از نوع **String[]** باشد چون نوع عبارت سمت راست عملگر انتساب (=) همین است. شما می‌توانید از ویژگی آرایه با نوع ضمنی سی‌شارپ استفاده کنید تا کامپایلر، نوع عناصر آرایه را استنتاج کند. دقت کنید کد زیر بین **new** و **[]** نوعی را تعیین نمی‌کند:

```
// Using C#'s implicitly typed local variable and implicitly typed array features:
```

```
var names = new[] { "Aidan", "Grant", null };
```

در خط فوق، کامپایلر نوع‌های عبارت‌هایی که درون آرایه برای مقداردهی عناصر آرایه استفاده شده‌اند را بررسی می‌کند و کامپایلر نزدیک ترین نوع پایه که تمام عناصر در آن مشترک هستند را برای تعیین نوع آرایه انتخاب می‌کند. در این مثال، کامپایلر دو **String** و **null** می‌بیند. چون **null** به صورت ضمنی قابل تبدیل به هر نوع ارجاعی (شامل **String**) است، کامپایلر استنتاج می‌کند که باید یک آرایه از اشاره‌گرهای **String** را ساخته و مقداردهی اولیه کند اگر شما این کد را داشتید:

```
// Using C#'s implicitly typed local variable & implicitly typed array features: (error)
```

```
var names = new[] { "Aidan", "Grant", 123 };
```

کامپایلر پیام زیر را اعلام می‌کند:

"error CS0820: No best type found for implicitly-typed array."

این بدین خاطر است که نوع پایه مشترک بین دو **String** و **int32** **Object** است که یعنی کامپایلر مجبور خواهد بود آرایه‌ای از اشاره‌گرهای **Object** ساخته و **123** را بسته‌بندی کرده و آخرین عنصر آرایه به یک **int32** بسته‌بندی شده با یک مقدار **123** اشاره کند. تیم کامپایلر سی‌شارپ فکر کردند که بسته‌بندی عناصر کار سنگینی برای کامپایلر است که این کار را برای شما به صورت ضمنی انجام دهد و به همین خاطر کامپایلر اعلام خطا می‌کند. و به عنوان یک امتیاز نحوی اضافه، هنگام مقداردهی اولیه یک آرایه، شما می‌توانید کد زیر را بنویسید:

```
string[] names = { "Aidan", "Grant" };
```

به سمت راست عملگر انتساب (=) دقت کنید، تنها، عبارت مقداردهی کننده آرایه بدون هیچ گونه **new**، هیچ گونه نوع و هیچ گونه [] آمده است. این نحو خوب است اما متأسفانه، کامپایلر سی شارپ اجازه نمی‌دهد از متغیرهای محلی با نوع ضمنی در این نحو استفاده کنید:

```
// This is a local variable now (error)
var names = { "Aidan", "Grant" };
```

اگر شما سعی کنید کد فوق را کامپایل کنید، کامپایلر دو پیام اعلام می‌کند:

“error CS0820: Cannot initialize an implicitly-typed local variable with an array initialize”

و

“error CS0622: Can only use array initialize expressions to assign to array types. Try using a new expression instead.”

در حالیکه که کامپایلر می‌توانید این کار را انجام دهد، تیم سی شارپ فکر کردند که کامپایلر در اینجا کار بسیار زیادی برای شما انجام خواهد داد. نوع آرایه را استنتاج می‌کند، آرایه را **new** می‌کند، آرایه را مقداردهی اولیه می‌کند و نوع متغیر محلی را نیز استنتاج می‌کند.

آخرین چیزی که دوست دارم به شما نشان دهم اینست که چگونه از آرایه‌های با نوع ضمنی همراه نوع‌های ناشناس و متغیرهای با نوع ضمنی استفاده کنید. نوع‌های ناشناس و اینکه چگونه هویت نوع بر آن‌ها اعمال می‌شود در فصل ۱۰ "ویژگی‌ها" بحث شده است. کد زیر را بررسی کنید:

```
// Using C#'s implicitly typed local, implicitly typed array, and anonymous type features:
var kids = new[] {new { Name="Aidan" }, new { Name="Grant" }};
```

```
// Sample usage (with another implicitly typed local variable):
```

```
foreach (var kid in kids)
    Console.WriteLine(kid.Name);
```

در این مثال، من از مقداردهی کننده آرایه که دو عبارت برای عناصر آرایه دارد استفاده می‌کنم. هر عبارت یک نوع ناشناس (چون هیچ نام نوعی بعد از عملگر **new** نیامده است) را نمایش می‌دهد. چون دو نوع ناشناس ساختار یکسانی دارند (یک کلید به نام **Name** از نوع **String**)، کامپایلر می‌داند که این دو شی دقیقاً از یک نوع هستند. حالا، من از ویژگی آرایه با نوع ضمنی سی شارپ (هیچ نوعی بین **new** و [] تعیین نشده است) استفاده می‌کنم تا خود کامپایلر، نوع آرایه را استنتاج کند، شی این آرایه را بسازد و اشاره‌گرهای آن به دو نمونه از یک نوع ناشناس را مقداردهی اولیه کند.^{۵۴} سرانجام، یک اشاره‌گر به این شی آرایه به متغیر محلی **kids** اختصاص می‌یابد، از نوعی که توسط کامپایلر به خاطر ویژگی متغیر محلی با نوع ضمنی سی شارپ استنتاج شده است. من حلقه **foreach** را به عنوان یک مثال نشان دادم تا ببینید چگونه از این آرایه که با دو شی از نوع ناشناس ساخته و مقداردهی اولیه شده است استفاده کنید. من مجبورم از یک متغیر محلی با نوع ضمنی (**kid**) نیز برای حلقه استفاده کنم. وقتی من این کد را اجرا می‌کنم، خروجی زیر را می‌گیرم:

```
Aidan
Grant
```

تبدیل آرایه‌ها

برای آرایه‌هایی با عناصر نوع ارجاعی، CLR اجازه می‌دهد به صورت ضمنی، نوع عنصر آرایه مبدا را به یک نوع هدف تبدیل کند. برای آنکه تبدیل موفقیت آمیز باشد هر دو نوع آرایه باید تعداد بعد یکسانی داشته باشند و یک تبدیل ضمنی یا صریح از نوع عنصر مبدا به نوع عنصر هدف، باید موجود باشد. CLR اجازه‌ی تبدیل آرایه‌هایی با عناصر نوع مقداری به هیچ نوع دیگری را نمی‌دهد. (هر چند با استفاده از متد **Array.Copy** شما می‌توانید یک آرایه جدید ساخته و عناصرش را به منظور دستیابی به نتیجه مورد نظر پر کنید). کد زیر نشان می‌دهد تبدیل آرایه چگونه کار می‌کند:

```
// Create a two-dimensional FileStream array.
FileStream[,] fs2dim = new FileStream[5, 10];

// Implicit cast to a two-dimensional Object array
Object[,] o2dim = fs2dim;
```

^{۵۴} اگر شما فکر می‌کنید این جملات برای خواندن جالب هستند، تصور کنید نوشتن آن‌ها در ابتدا چقدر جالب است.

```
// Can't cast from two-dimensional array to one-dimensional array
// Compiler error CS0030: Cannot convert type 'object[*,*]' to
// 'System.IO.Stream[]'
Stream[] s1dim = (Stream[]) o2dim;

// Explicit cast to two-dimensional Stream array
Stream[,] s2dim = (Stream[,]) o2dim;

// Explicit cast to two-dimensional String array
// Compiles but throws InvalidCastException at runtime
String[,] st2dim = (String[,]) o2dim;

// Create a one-dimensional Int32 array (value types).
Int32[] i1dim = new Int32[5];

// Can't cast from array of value types to anything else
// Compiler error CS0030: Cannot convert type 'int[]' to 'object[]'
Object[] o1dim = (Object[]) i1dim;

// Create a new array, then use Array.Copy to coerce each element in the
// source array to the desired type in the destination array.
// The following code creates an array of references to boxed Int32s.
Object[] ob1dim = new Object[i1dim.Length];
Array.Copy(i1dim, ob1dim, i1dim.Length);
```

متد **Array.Copy** تنها یک متد نیست که عناصر را از یک آرایه به دیگری کپی کند. متد **Copy** نواحی همپوشانی شده از حافظه را به درستی همانند تابع **memmove** از C مدیریت می‌کند. تابع **memcpy** از C، در سوی دیگر، نواحی همپوشانی شده از حافظه را به درستی مدیریت نمی‌کند. متد **Copy** همچنین می‌تواند هر عنصر آرایه را در صورتی که تبدیل نیاز باشد تبدیل کند. متد **Copy** قادر است تبدیلات زیر را انجام دهد:

- بسته‌بندی عناصر نوع مقداری به عناصر نوع ارجاعی، مثل کپی یک **Int32[]** به یک **Object[]**.
- باز کردن (عکس بسته‌بندی) عناصر نوع ارجاعی به عناصر نوع مقداری، مثل کپی یک **Object[]** به یک **Int32[]**.
- عریض کردن نوع‌های مقداری اصلی CLR، مثل کپی کردن عناصر از یک **Int32[]** به یک **Double[]**.
- تبدیل رو به پایین^{۵۵} عناصر هنگام کپی بین نوع‌های آرایه که نتوان بر اساس نوع آرایه، سازگاری را تعیین کرد مثل وقتی که از یک **Object[]** به یک **IFormattable[]** تبدیل می‌کنید. اگر هر شی موجود در **Object[]**، **IFormattable** را پیاده‌سازی کنند، **Copy** موفق خواهد بود.

مثال دیگری که از سودمندی **Copy** استفاده می‌کند:

```
// Define a value type that implements an interface.
internal struct MyValueType : IComparable {
    public Int32 CompareTo(Object obj) {
        ...
    }
}

public static class Program {
    public static void Main() {
```

^{۵۵} Downcasting در مقابل Upcasting قرار دارد. تبدیل رو به پایین یعنی شما در سلسه مراتب شی به سمت پایین(نوع‌های مشتق شده) حرکت می‌کنید.

```
// Create an array of 100 value types.
MyValueType[] src = new MyValueType[100];

// Create an array of IComparable references.
IComparable[] dest = new IComparable[src.Length];

// Initialize an array of IComparable elements to refer to boxed
// versions of elements in the source array.
Array.Copy(src, dest, src.Length);
}
}
```

همانگونه که احتمالاً تصور می‌کنید، کتابخانه کلاس فریمورک (FCL) از متد **Copy** از **Array** بسیار زیاد بهره می‌برد.

در بعضی موارد، مفید است که یک آرایه از یک نوع را به دیگری تبدیل کنید. این گونه کاربرد، کواریانس آرایه **array covariance** نامیده می‌شود. وقتی شما از کواریانس آرایه بهره می‌برید، شما باید از ضربه‌ای که به کارایی وارد می‌شود هم مطلع باشید. فرض کنید کد زیر را دارید:

```
String[] sa = new String[100];
Object[] oa = sa; // oa refers to an array of string elements
oa[5] = "Jeff"; // Perf hit: CLR checks oa's element type for String; OK
oa[3] = 5; // Perf hit: CLR checks oa's element type for Int32; throws
// ArrayTypeMismatchException
```

در کد فوق، متغیر **oa** از نوع **Object[]** است، هر چند در واقع به یک **String[]** اشاره دارد. کامپایلر به شما اجازه می‌دهد کدی بنویسید که یک 5 در یک عنصر آرایه قرار دهد، چون 5 یک **Int32** است که از **Object** مشتق شده است. البته CLR باید از امنیت نوع اطمینان حاصل کند. هنگام انتساب به یک عنصر آرایه، CLR باید مطمئن شود انتساب مجاز است. پس CLR باید در زمان اجرا بررسی کند آیا آرایه حاوی عناصر **Int32** است. در این مورد اینگونه نیست و انتساب اجازه داده نمی‌شود؛ CLR یک **ArrayTypeMismatchException** تولید می‌کند.

نکته اگر شما فقط نیاز دارید یک کپی از برخی عناصر آرایه به آرایه دیگری انجام دهید، متد **BlockCopy** از **System.Buffer** سریعتر از متد **Copy** از **Array** کار می‌کند. پارامترهای **Int32** به عنوان آفست‌های بیتی در آرایه بیان می‌شوند نه به عنوان اندیس‌های عناصر. **BlockCopy** واقعا برای کپی داده از یک نوع آرایه به نوع آرایه **bittable** (نوع‌هایی که نمایش مشترکی در حافظه مدیریت شده و نشده دارند. به همین خاطر هنگام ارسال بین کد مدیریت شده و نشده نیاز به تبدیل ندارند) دیگر که از لحاظ بیتی سازگار است، طراحی شده است مثل کپی یک **Byte[]** حاوی کاراکترهای یونیکد (با ترتیب صحیح بایت‌ها) به یک **Char[]**. این متد به برنامه‌نویس اجازه می‌دهد تا حدی فقدان این قابلیت که با آرایه به عنوان بلوکی از حافظه از هر نوع رفتار شود را بپوشاند.

اگر شما نیاز دارید با اطمینان، یک مجموعه از عناصر آرایه را از یک آرایه به آرایه دیگر انتقال دهید، شما باید از متد **ConstrainedCopy** از **System.Array** استفاده کنید. این متد تضمین می‌کند که عمل کپی یا کامل انجام می‌شود یا بدون تخریب هر داده‌ای در آرایه مقصد، یک اکسپشن تولید می‌کند. این اجازه می‌دهد **ConstrainedCopy** در ناحیه اجرایی محدود شده (CER) اجرا گردد. به منظور ارائه‌ی تضمین، **ConstrainedCopy** نیاز دارد که نوع عنصر آرایه مبدا از همان نوع یا مشتق شده از نوع عنصر آرایه مقصد باشد. به علاوه، آن هیچ گونه بسته بندی، باز کردن یا تبدیل رو به پایین انجام نمی‌دهد.

تمام آرایه‌ها به صورت ضمنی از **System.Array** مشتق شده‌اند

وقتی شما یک متغیر شبیه به این تعریف می‌کنید:

```
FileStream[] fsArray;
```

آنگاه CLR به صورت خودکار یک نوع **FileStream[]** برای **AppDomain** می‌سازد. این نوع ضمنی از نوع **System.Array** مشتق می‌شود و بنابراین تمام متدهای نمونه و ویژگی‌های تعریف شده در نوع **System.Array** توسط نوع **FileStream[]** به ارث برده می‌شود که اجازه می‌دهد این

متدها و ویژگی‌ها توسط متغیر **fsArray** فراخوانی شوند. این، کار با آرایه‌ها را بسیار آسان می‌کند چون متدها و ویژگی‌های کمکی بسیاری در **System.Array** تعریف شده است مثل **Clone**، **CopyTo**، **GetLength**، **GetLongLength**، **GetLowerBound** و غیره. نوع **System.Array** تعداد زیادی متد استاتیک بسیار پرکاربرد ارائه می‌کند که روی آرایه‌ها عمل می‌کنند. این متدها تماما یک اشاره‌گر به یک آرایه به عنوان یک پارامتر می‌گیرند. برخی از متدهای استاتیک مفید مثل **AsReadOnly**، **BinarySearch**، **Clear**، **ConstrainedCopy**، **ConvertAll**، **Copy**، **Exists**، **Find**، **FindAll**، **FindIndex**، **FindLast**، **ForEach**، **IndexOf**، **LastIndexOf**، **Resize**، **Reserve**، **Sort** و **TrueForAll**. تعداد زیادی سربارگذاری از هر کدام از این متدها وجود دارد. در واقع، بسیاری از متدها، سربارگذاری‌های جنریک را برای امنیت نوع در زمان کامپایل و عملکرد خوب، ارائه می‌کنند. من شما را به بررسی مستندات SDK تشویق می‌کنم تا درکی از سودمندی و قدرت این متدها بدست آورید.

تمام آرایه‌ها به صورت ضمنی **IEnumerable**، **ICollection** و **IList** را پیاده‌سازی می‌کنند

تعداد زیادی متد وجود دارند که روی اشیاء مختلف مجموعه عمل می‌کنند چون متدها با پارامترهایی مثل **IEnumerable**، **ICollection** و **IList** تعریف شده‌اند. چون **System.Array** نیز این سه رابط را پیاده‌سازی می‌کند؛ این امکان وجود دارد که آرایه‌ها را به این متدها ارسال کنید. **System.Array** این رابط‌های غیرجنریک را برای این پیاده‌سازی می‌کند که آن‌ها با تمام عناصر به عنوان **System.Object** رفتار می‌کنند. هرچند، خوب بود اگر **System.Array** معادل جنریک این رابط‌ها را پیاده‌سازی می‌کرد تا امنیت نوع بهتری در زمان کامپایل و عملکرد مطلوب تری را فراهم کند. تیم CLR به خاطر مسائل مربوط به آرایه‌های چند بعدی و آرایه‌های غیر پایه صفر، نخواستند که **System.Array** رابط‌های **ICollection<T>** و **IEnumerable<T>** را پیاده‌سازی کند. تعریف این رابط‌ها روی **System.Array**، این رابط‌ها را قادر می‌ساخت روی تمام نوع‌های آرایه عمل کنند. به جای آن، CLR حقه کوچکی زد؛ وقتی یک آرایه تک بعدی با حد پایین صفر ساخته می‌شود، CLR به صورت خودکار کاری می‌کند که نوع آرایه، رابط‌های **ICollection<T>**، **IEnumerable<T>** و **IList<T>** را پیاده‌سازی کند (در حالیکه **T** نوع عنصر آرایه است). همچنین این سه رابط را برای تمام نوع‌های پایه پیاده‌سازی می‌کند تا زمانی که آن‌ها نوع ارجاعی باشند. دیاگرام سلسه مراتب زیر این را روشن می‌کند:

Object

```

Array (non-generic IEnumerable, ICollection, IList)
  Object[] (IEnumerable, ICollection, IList of Object)
  String[] (IEnumerable, ICollection, IList of String)
  Stream[] (IEnumerable, ICollection, IList of Stream)
  FileStream[] (IEnumerable, ICollection, IList of FileStream)
  .
  . (other arrays of reference types)
  .

```

پس اگر برای مثال، شما کد زیر را داشته باشید:

```
FileStream[] fsArray;
```

آنگاه وقتی CLR نوع **FileStream[]** را می‌سازد، آن باعث می‌شود که این نوع به صورت خودکار رابط‌های **IEnumerable<FileStream>**، **ICollection<FileStream>** و **IList<FileStream>** را پیاده‌سازی کند. علاوه بر این، نوع **FileStream[]** رابط‌های نوع‌های پایه را نیز پیاده‌سازی می‌کند: **IEnumerable<Stream>**، **IEnumerable<Object>**، **ICollection<Stream>**، **ICollection<Object>**، **IList<Stream>**، **IList<Object>**. چون تمام این رابط‌ها به صورت خودکار توسط CLR پیاده‌سازی می‌شوند، متغیر **fsArray** می‌تواند هر جایی که این رابط‌ها حضور دارند، استفاده شود. برای نمونه، متغیر **fsArray** می‌تواند به متدهایی ارسال شود که هر یک از فرم‌های کلی زیر را دارند:

```

void M1(IList<FileStream> fsList) { ... }
void M2(ICollection<Stream> sCollection) { ... }
void M3(IEnumerable<Object> oEnumerable) { ... }

```

دقت کنید اگر آرایه، حاوی عناصر نوع مقداری باشند، نوع آرایه رابط‌های نوع‌های پایه را پیاده‌سازی نخواهد کرد. برای نمونه، اگر شما کد زیر را داشته باشید:

```
Date[] dtArray; // An array of value types
```

آنگاه نوع `DateTime[]` فقط `IEnumerable<DateTime>`، `ICollection<DateTime>` و `IList<DateTime>` را پیاده‌سازی خواهد کرد، نسخه‌هایی از این رابطها که روی `System.ValueType` یا `System.Object` جنریک هستند را پیاده‌سازی نمی‌کند. این یعنی متغیر `dtArray` نمی‌تواند به عنوان یک آرگومان به متد `M3` که قبلا نشان داده شد ارسال گردد. علت اینست که آرایه‌هایی از نوع‌های مقداری، متفاوت از آرایه‌هایی از نوع ارجاعی، در حافظه جای می‌گیرند. قالب بندی حافظه آرایه قبلا در این فصل بحث شد.

ارسال و برگرداندن آرایه‌ها

هنگام ارسال یک آرایه به عنوان یک آرگومان به یک متد، شما در حقیقت یک اشاره‌گر به آرایه، ارسال می‌کنید. بنابراین، متد فراخوانی شده قادر است عناصر آرایه را تغییر دهد. اگر شما نمی‌خواهید این رخ دهد، شما باید یک کپی از آرایه تهیه کنید و کپی را به متد ارسال کنید. توجه کنید متد `Array.Copy` یک کپی سطحی انجام می‌دهد و بنابراین اگر عناصر آرایه نوع ارجاعی هستند، آرایه جدید به اشیای موجود قبلی اشاره می‌کند.

به طریق مشابه، بعضی متدها یک اشاره‌گر به یک آرایه برمی‌گردانند. اگر متد، آرایه را ساخته و مقداردهی اولیه کند، برگرداندن یک اشاره‌گر به یک آرایه خوب است. اگر متد بخواد یک اشاره‌گر به یک آرایه داخلی که توسط یک فیلد نگهداری می‌شود را برگرداند، شما باید تصمیم بگیرید اگر می‌خواهید فراخوانی کننده‌ی متد، دسترسی مستقیم به آرایه و عناصرش داشته باشد. اگر شما این را می‌خواهید، تنها اشاره‌گر آرایه را برگردانید. اما اغلب شما نمی‌خواهید فراخوانی کننده‌ی متد چنین دسترسی‌ای داشته باشد، پس متد باید یک آرایه جدید ساخته و `Array.Copy` را فراخوانی کند و یک اشاره‌گر به آرایه جدید برگرداند. مجدداً، آگاه باشید که `Array.Copy` یک کپی سطحی از آرایه اصلی می‌سازد.

اگر شما یک متد تعریف کنید که یک اشاره‌گر به یک آرایه برگرداند و اگر آن آرایه عنصری درونش نباشد، متد شما می‌تواند `null` یا یک اشاره‌گر به یک آرایه با صفر عنصر برگرداند. وقتی شما اینگونه متد را پیاده‌سازی می‌کنید، میکروسافت قویاً توصیه می‌کند که شما متد را برگرداندن یک آرایه با اندازه صفر پیاده‌سازی کنید چون انجام چنین کاری، کدی که برنامه‌نویس فراخوانی کننده‌ی متد باید بنویسد را ساده می‌کند. برای نمونه، این کد که به آسانی قابل درک است، صحیح اجرا می‌شود حتی اگر قرار ملاقاتی برای بررسی وجود نداشته باشد:

```
// This code is easier to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0; a < appointments.Length; a++) {
    ...
}
```

کد زیر نیز به درستی کار می‌کند اگر قرار ملاقاتی برای بررسی وجود نداشته باشد. اما نوشتن و درک این کد کمی سخت تر است:

```
// This code is harder to write and understand.
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
    for (Int32 a = 0, a < appointments.Length; a++) {
        // Do something with appointments[a]
    }
}
```

اگر شما متدهایتان را با برگرداندن آرایه‌هایی با صفر عنصر به جای برگرداندن `null`، طراحی کنید، فراخوانی کننده‌های متدهای شما در کار با آن‌ها راحت ترند. ضمناً، شما باید همین کار را برای فیلدها نیز بکنید. اگر نوع شما یک فیلد دارد که یک اشاره‌گر به یک آرایه است، شما باید در نظر بگیرید فیلد به یک آرایه اشاره کند حتی اگر عنصری درون آن نباشد.

ساخت آرایه‌هایی با حد پایین غیر صفر

قبلا اشاره کردم که این امکان وجود دارد که آرایه‌هایی با حد پایین غیر صفر ساخته و با آن‌ها کار کنید. شما می‌توانید به صورت پویا آرایه‌های خود را با فراخوانی متد استاتیک `CreateInstance` از `Array`، بسازید. چندین سربارگذاری از این متد وجود دارد که به شما اجازه می‌دهند نوع عناصر آرایه، تعداد ابعاد آرایه، حدهای پایین هر بعد و تعداد عناصر در هر بعد را تعیین نمایید. `CreateInstance` حافظه را برای آرایه تخصیص داده، اطلاعات پارامتر را در بخش اضافی (`overhead`) از بلوک حافظه ذخیره می‌کند و یک اشاره‌گر به آرایه برمی‌گرداند. اگر آرایه دو یا بیشتر بعد داشته باشد، شما می‌توانید اشاره‌گر برگشتی از `CreateInstance` را به یک متغیر `ElementType[]` (که `ElementType` نام یک نوع است) تبدیل کنید تا دسترسی شما به عناصر

آرایه را آسانتر کند. اگر آرایه تنها یک بعد داشته باشد، در سی شارپ شما مجبورید از متدهای **GetValue** و **SetValue** از **Array** برای دسترسی به عناصر آرایه استفاده کنید.

کد زیر نشان می‌دهد چگونه به صورت پویا یک آرایه دو بعدی از مقادیر **System.Decimal** بسازید. اولین بعد، سال‌های تقویم از ۲۰۰۵ تا ۲۰۰۹ را نمایش می‌دهد و بعد دوم فصل‌های ۱ تا ۴ را نمایش می‌دهد. کد در بین تمام عناصر موجود در آرایه‌ی پویا حرکت می‌کند. من می‌توانم مستقیماً حدود آرایه را در کد بنویسم که عملکرد بهتری را بدست می‌داد اما تصمیم گرفتیم از متد **GetUpperBound** و **GetLowerBound** از **System.Array** برای نمایش کاربردشان استفاده کنیم:

```
using System;
```

```
public static class DynamicArrays {
    public static void Main() {
        // I want a two-dimensional array [2005..2009][1..4].
        Int32[] lowerBounds = { 2005, 1 };
        Int32[] lengths = { 5, 4 };
        Decimal[,] quarterlyRevenue = (Decimal[,])
            Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);

        Console.WriteLine("{0,4} {1,9} {2,9} {3,9} {4,9}",
            "Year", "Q1", "Q2", "Q3", "Q4");
        Int32 firstYear = quarterlyRevenue.GetLowerBound(0);
        Int32 lastYear = quarterlyRevenue.GetUpperBound(0);
        Int32 firstQuarter = quarterlyRevenue.GetLowerBound(1);
        Int32 lastQuarter = quarterlyRevenue.GetUpperBound(1);

        for (Int32 year = firstYear; year <= lastYear; year++) {
            Console.Write(year + " ");
            for (Int32 quarter = firstQuarter; quarter <= lastQuarter; quarter++) {
                Console.Write("{0,9:C} ", quarterlyRevenue[year, quarter]);
            }
            Console.WriteLine();
        }
    }
}
```

اگر شما این کد را کامپایل و اجرا کنید، خروجی زیر را می‌گیرید:

Year	Q1	Q2	Q3	Q4
2005	\$0.00	\$0.00	\$0.00	\$0.00
2006	\$0.00	\$0.00	\$0.00	\$0.00
2007	\$0.00	\$0.00	\$0.00	\$0.00
2008	\$0.00	\$0.00	\$0.00	\$0.00
2009	\$0.00	\$0.00	\$0.00	\$0.00

کارایی دسترسی به آرایه

در درون، CLR، در حقیقت از دو گونه آرایه پشتیبانی می‌کند:

- آرایه‌های تک بعدی با حد پایین صفر. این آرایه‌ها گاهی آرایه‌های SZ (برای single-dimensional و zero-based) یا بردارها نامیده می‌شوند.
- آرایه‌های تک بعدی و چند بعدی با حد پایین نامشخص.

شما می‌توانید گونه‌های مختلف آرایه‌ها را با اجرای کد زیر ببینید (خروجی در کامنت‌های کد نشان داده شده است):

```
using System;

public sealed class Program {
    public static void Main() {
        Array a;

        // Create a 1-dim, 0-based array, with no elements in it
        a = new String[0];
        Console.WriteLine(a.GetType());    // "System.String[]"

        // Create a 1-dim, 0-based array, with no elements in it
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0 }, new Int32[] { 0 });
        Console.WriteLine(a.GetType());    // "System.String[]"

        // Create a 1-dim, 1-based array, with no elements in it
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0 }, new Int32[] { 1 });
        Console.WriteLine(a.GetType());    // "System.String[*]" <-- INTERESTING!
        Console.WriteLine();

        // Create a 2-dim, 0-based array, with no elements in it
        a = new String[0, 0];
        Console.WriteLine(a.GetType());    // "System.String[,]"

        // Create a 2-dim, 0-based array, with no elements in it
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0, 0 }, new Int32[] { 0, 0 });
        Console.WriteLine(a.GetType());    // "System.String[,]"

        // Create a 2-dim, 1-based array, with no elements in it
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0, 0 }, new Int32[] { 1, 1 });
        Console.WriteLine(a.GetType());    // "System.String[,]"
    }
}
```

کنار هر **Console.WriteLine** یک کامنت هست که خروجی را تعیین می‌کند. برای آرایه‌های تک بعدی، آرایه‌ها پایه صفر یک نام نوع **System.String[]** را نمایش می‌دهند در حالیکه آرایه‌های پایه ۱، یک نام **System.String[*]** نمایش می‌دهند. * بیان می‌کند که CLR می‌داند این آرایه پایه صفر نیست. توجه کنید سی‌شارپ اجازه نمی‌دهد یک متغیر از نوع **String[*]** تعریف کنید و بنابراین غیرممکن است که از نحو سی‌شارپ برای دسترسی به یک آرایه تک بعدی با پایه غیر صفر استفاده کنید. اگرچه شما می‌توانید متدهای **GetValue** و **SetValue** از **Array** را برای دسترسی به عناصر آرایه فراخوانی کنید، این دسترسی به خاطر فراخوانی متد، کند خواهد بود.

برای آرایه‌های چند بعدی، آرایه‌های پایه صفر و پایه ۱، همه نوع یکسانی را نمایش می‌دهند، **System.String[]**. CLR با همه‌ی آرایه‌های چند بعدی به گونه‌ای رفتار می‌کند که گویا آن‌ها در زمان اجرا پایه صفر نیستند. این باعث می‌شود شما فکر کنید که نام نوع باید **System.String[*]** نمایش داده شود، هر چند، CLR از * ها برای آرایه‌های چند بعدی استفاده نمی‌کند چون همیشه حضور خواهند داشت و کاراکتر ستاره اکثر برنامه‌نویسان را گیج خواهد کرد. دسترسی به عناصر یک آرایه تک بعدی پایه صفر، کمی سریعتر از دسترسی به عناصر یک آرایه پایه غیر صفر تک بعدی یا چند بعدی است. چندین

دلیل برای این وجود دارد: اول اینکه، دستورات IL خاصی — مثل `newarr`، `delem`، `delema`، `ldlen` و `stelem`— برای دستکاری آرایه‌های تک بعدی پایه صفر وجود دارند و این دستورات خاص IL باعث می‌شوند کامپایلر JIT کد بهینه تولید کند. برای نمونه، کامپایلر JIT کدی تولید می‌کند که فرض می‌کند آرایه پایه صفر است و این یعنی هنگام دسترسی به عنصر، یک آفست مجبور نیست از اندیس تعیین شده کسر شود. دوم اینکه، در وضعیت‌های رایجی کامپایلر JIT قادر است که بررسی دامنه اندیس را به خارج از حلقه ببرد و باعث شود کد فقط یکبار اجرا گردد. برای نمونه به کد راجع زیر توجه کنید:

```
using System;

public static class Program {
    public static void Main() {
        Int32[] a = new Int32[5];
        for(Int32 index = 0; index < a.Length; index++) {
            // Do something with a[index]
        }
    }
}
```

اولین نکته درباره‌ی این کد، فراخوانی ویژگی `Length` از آرایه در عبارت تست حلقه `for` است. چون `Length` یک ویژگی است، خواندن طول در حقیقت یک متد را فراخوانی می‌کند. هرچند، کامپایلر JIT می‌داند که `Length` یک ویژگی در کلاس `Array` است و در حقیقت کامپایلر JIT کدی تولید می‌کند که ویژگی را فقط یکبار فراخوانی کرده و نتیجه را در متغیر موقتی که با هر اجرای حلقه بررسی می‌شود، ذخیره می‌کند. نتیجه اینست که کد JIT شده، سریع است. در حقیقت، بعضی برنامه‌نویسان قابلیت‌های کامپایلر JIT را دست کم گرفته‌اند و سعی کرده‌اند "کد باهوش" بنویسند تا به کامپایلر JIT کمک کنند. هر چند، هر سعی هوشمندانه‌ای که انجام دهید مطمئناً اثر عکس داشته و خواندن کدتان را سخت‌تر کرده و قابلیت نگهداری آن را پایین می‌آورد.

شما بهتر است فراخوانی به ویژگی `Length` از آرایه را در کد فوق رها کنید به جای اینکه خودتان آن را در یک متغیر محلی جای دهید.

دومین نکته درباره‌ی کد فوق اینست که کامپایلر JIT می‌داند که حلقه `for` به عناصر `0` تا `Length - 1` از آرایه دسترسی پیدا می‌کند. پس کامپایلر JIT کدی تولید می‌کند که در زمان اجرا تست می‌کند تمام دسترسی‌ها به آرایه در بازه‌ی معتبر آرایه باشد. به خصوص، کامپایلر JIT کدی تولید می‌کند تا بررسی کند آیا شرط زیر برقرار است:

`(0 >= a.GetLowerBound(0)) && ((Length - 1) <= 1.GetUpperBound(0))`

این بررسی درست قبل از حلقه رخ می‌دهد. اگر بررسی درست باشد، کامپایلر JIT کدی درون حلقه برای بررسی اینکه هر دسترسی در بازه‌ی معتبر باشد، تولید نمی‌کند. این باعث می‌شود دسترسی به آرایه درون حلقه خیلی سریع باشد.

متأسفانه، همانگونه که قبلاً در این فصل اشاره کردم، دسترسی به آرایه پایه غیر صفر تک بعدی یا چند بعدی به مراتب از آرایه تک بعدی پایه صفر آهسته‌تر است. برای این نوع‌های آرایه، کامپایلر JIT بررسی اندیس را بیرون حلقه نمی‌برد، پس هر دسترسی به آرایه، اندیس‌های تعیین شده را ارزیابی می‌کند. به علاوه کامپایلر JIT کدی تولید می‌کند تا حدهای پایین آرایه را از اندیس تعیین شده کسر کند که کد را کند می‌کند حتی اگر شما از یک آرایه چند بعدی استفاده کنید که پایه صفر نیز است.

پس اگر نگران عملکرد هستید، شما باید از آرایه‌ای از آرایه‌ها (آرایه‌های دندانه‌دار) به جای یک آرایه مستطیلی استفاده کنید. سی‌شارپ و CLR اجازه می‌دهند شما به یک آرایه توسط کد ناامن (غیرقابل بررسی) دسترسی پیدا کنید که تکنیکی است که به شما اجازه می‌دهد بررسی حدود اندیس هنگام دسترسی به یک آرایه را غیرفعال کنید. توجه کنید این تکنیک دستکاری غیرامن آرایه برای آرایه‌هایی که عناصرشان `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`، `Boolean`، `Decimal`، `Double`، `Single`، `Char`، `UInt64`، `Int64`، `UInt32`، `Int32`، `UInt16`، `Int16`، `Byte`، `SByte`

```

using System.Diagnostics;

public static class Program {
    private const Int32 c_numElements = 10000;

    public static void Main() {
        const Int32 testCount = 10;
        Stopwatch sw;

        // Declare a two-dimensional array
        Int32[,] a2Dim = new Int32[c_numElements, c_numElements];

        // Declare a two-dimensional array as a jagged array (a vector of vectors)
        Int32[][] aJagged = new Int32[c_numElements][];
        for (Int32 x = 0; x < c_numElements; x++)
            aJagged[x] = new Int32[c_numElements];

        // 1: Access all elements of the array using the usual, safe technique
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            Safe2DimArrayAccess(a2Dim);
        Console.WriteLine("{0}: Safe2DimArrayAccess", sw.Elapsed);

        // 2: Access all elements of the array using the jagged array technique
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            SafeJaggedArrayAccess(aJagged);
        Console.WriteLine("{0}: SafeJaggedArrayAccess", sw.Elapsed);

        // 3: Access all elements of the array using the unsafe technique
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            Unsafe2DimArrayAccess(a2Dim);
        Console.WriteLine("{0}: Unsafe2DimArrayAccess", sw.Elapsed);
        Console.ReadLine();
    }

    private static Int32 Safe2DimArrayAccess(Int32[,] a) {
        Int32 sum = 0;
        for (Int32 x = 0; x < c_numElements; x++) {
            for (Int32 y = 0; y < c_numElements; y++) {
                sum += a[x, y];
            }
        }
        return sum;
    }

    private static Int32 SafeJaggedArrayAccess(Int32[][] a) {

```

```

Int32 sum = 0;
for (Int32 x = 0; x < c_numElements; x++) {
    for (Int32 y = 0; y < c_numElements; y++) {
        sum += a[x][y];
    }
}
return sum;
}

private static unsafe Int32 Unsafe2DimArrayAccess(Int32[,] a) {
    Int32 sum = 0;
    fixed (Int32* pi = a) {
        for (Int32 x = 0; x < c_numElements; x++) {
            Int32 baseOfDim = x * c_numElements;
            for (Int32 y = 0; y < c_numElements; y++) {
                sum += pi[baseOfDim + y];
            }
        }
    }
    return sum;
}
}

```

متد **Unsafe2DimArrayAccess** با تغییر دهنده‌ی **unsafe** علامت زده شده است که برای استفاده از عبارت **fixed** سی‌شارپ لازم است. برای کامپایل این کد، شما باید سویچ **/unsafe** را برای کامپایلر سی‌شارپ تعیین کرده یا گزینه‌ی **“Allow Unsafe Code”** در تب **Build** از **Project Properties** در ویژوال استودیو را تیک بزنید.

وقتی من این برنامه را روی ماشینم اجرا می‌کنم، خروجی زیر را می‌گیرم:

```

00:00:02.0017692: Safe2DimArrayAccess
00:00:01.5197844: SafeJaggedArrayAccess
00:00:01.7343436: Unsafe2DimArrayAccess

```

همانگونه که می‌بینید، تکنیک دسترسی امن به آرایه دوبعدی کند است. تکنیک دسترسی امن به آرایه دندانه‌دار برای کامل شدن کمی از تکنیک دسترسی امن به آرایه دو بعدی کمتر زمان می‌گیرد. هر چند، شما باید توجه کنید ساخت آرایه دندانه‌دار از ساخت آرایه دو بعدی زمان برتر است چون ساخت آرایه دندانه دار نیاز دارد برای هر بعد یک شی در هیپ تخصیص یابد که باعث می‌شود در فواصلی، جمع‌آوری صورت پذیرد. پس یک معامله وجود دارد: اگر شما نیاز به ساخت تعداد زیادی "آرایه چند بعدی" دارید و قصد دارید به ندرت به عناصر آن دسترسی داشته باشید، سریعتر است که یک آرایه چند بعدی بسازید. اگر شما نیاز دارید "آرایه چند بعدی" را فقط یکبار بسازید و به عناصرش مکرراً دسترسی داشته باشید، یک آرایه دندانه‌دار عملکرد بهتری به شما می‌دهد، مطمئناً در اغلب برنامه‌ها، سناریوی دوم رایج تر است.

سرانجام، توجه کنید که تکنیک دسترسی ناامن به آرایه دو بعدی به اندازه‌ی تکنیک دسترسی امن به آرایه دو بعدی سرعت دارد و اگر شما دسترسی به یک تک آرایه دو بعدی (یک تخصیص حافظه) را مد نظر بگیرید در مقایسه با ساخت آرایه دندانه دار (که نیاز به تعداد بیشتری تخصیص حافظه دارد)، سریعترین آن‌ها نیز خواهد بود. واضح است که تکنیک ناامن وقتی به خوبی در کد شما استفاده شود جای خود را دارد اما آگاه باشید که سه جنبه منفی جدی در استفاده از این تکنیک وجود دارد:

- کدی که به عناصر آرایه دسترسی دارد برای خواندن و نوشتن، پیچیده تر از وقتی است که شما به عناصر با استفاده از دیگر تکنیک‌ها دسترسی دارید چون شما دارید از عبارت **fixed** سی‌شارپ استفاده کرده و محاسبات آدرس حافظه را انجام می‌دهید.
- اگر شما در محاسبات اشتباهی مرتکب شوید، شما به حافظه‌ای که بخشی از آرایه نیست دسترسی پیدا می‌کنید. این می‌تواند منجر به یک محاسبه اشتباه، تخریب حافظه، نقض امنیت نوع و حفره امنیتی احتمالی گردد.
- به خاطر مشکلات احتمالی، CLR مانع از اجرای کد ناامن در محیط‌هایی با امنیت محدود (مثل **Silverlight**) می‌شود.

دسترسی ناامن به آرایه ها و آرایه های با اندازه ثابت

دسترسی ناامن به آرایه، خیلی قدرتمند است چون به شما اجازه می‌دهد به موارد زیر دسترسی پیدا کنید:

- عناصر درون یک شی آرایه مدیریت شده که در هیپ قرار دارد (همانگونه که بخش قبلی نشان داد).
- عناصر درون یک آرایه که در هیپ مدیریت نشده قرار دارد. مثال **SecureString** در فصل ۱۴ "کاراکترها، رشته‌ها و کار با متن"، استفاده از دسترسی ناامن به آرایه روی یک آرایه برگشتی از متد **SecureStringToCoTaskMemUnicode** از کلاس **System.Runtime.InteropServices.Marshal** را نشان داد.
- عناصر درون یک آرایه که در پشت‌پشته ترد قرار دارند.

در مواقعی که عملکرد بسیار بحرانی است، شما می‌توانید از تخصیص یک شی آرایه مدیریت شده در هیپ خودداری کرده و به جای آن آرایه را در پشت‌پشته ترد با استفاده از عبارت **stackalloc** (که همانند تابع **alloca** از C کار می‌کند) تخصیص دهید. عبارت **stackalloc** می‌تواند برای ساخت یک آرایه تک بعدی پایه صفر از فقط عناصر نوع مقداری استفاده شود و نوع مقداری نباید هیچ گونه فیلد نوع ارجاعی داشته باشد. واقعا شما باید به این به عنوان تخصیص یک بلوک از حافظه که می‌توانید با استفاده از اشاره‌گرهای ناامن آن را دستکاری کنید نگاه کنید و بنابراین، شما نمی‌توانید آدرس این بافر حافظه را به اکثر متدهای FCL ارسال کنید. البته، حافظه تخصیص یافته در پشت‌پشته (آرایه) وقتی متد برگردد به طور خودکار آزاد می‌شود، این جاییست که ما شاهد بهبود عملکرد خواهیم بود. استفاده از این ویژگی همچنین نیاز دارد شما سوئیچ **unsafe** را برای کامپایلر تعیین کنید.

متد **StackallocDemo** در کد زیر مثالی از چگونگی استفاده از عبارت **stackalloc** سی‌شارپ را نشان می‌دهد:

```
using System;
```

```
public static class Program {
    public static void Main() {
        stackallocDemo();
        inlineArrayDemo();
    }

    private static void StackallocDemo() {
        unsafe {
            const Int32 width = 20;
            Char* pc = stackalloc Char[width]; // Allocates array on stack

            String s = "Jeffrey Richter"; // 15 characters

            for (Int32 index = 0; index < width; index++) {
                pc[width - index - 1] =
                    (index < s.Length) ? s[index] : '.';
            }

            // The line below displays ".....rethciR yerffej"
            Console.WriteLine(new String(pc, 0, width));
        }
    }

    private static void InlineArrayDemo() {
        unsafe {
            CharArray ca; // Allocates array on stack
            Int32 widthInBytes = sizeof(CharArray);
            Int32 width = widthInBytes / 2;
        }
    }
}
```

```
String s = "Jeffrey Richter"; // 15 characters

for (Int32 index = 0; index < width; index++) {
    ca.Characters[width - index - 1] =
        (index < s.Length) ? s[index] : '.';
}

// The line below displays ".....rethciR yerffej"
Console.WriteLine(new String(ca.Characters, 0, width));
}
}
}
```

```
internal unsafe struct CharArray {
    // This array is embedded inline inside the structure
    public fixed Char Characters[20];
}
```

به صورت عادی، چون آرایه‌ها نوع‌های ارجاعی هستند، یک فیلد آرایه تعریف شده در یک ساختار، یک اشاره‌گر یا ارجاع به یک آرایه است، خود آرایه بیرون از حافظه‌ی ساختار زندگی می‌کند. هر چند، این امکان وجود دارد که یک آرایه را مستقیماً درون یک ساختار تعبیه کرد، همانند ساختار **CharArray** در کد قبلی. برای تعبیه کردن یک آرایه درون یک ساختار، چندین پیش‌نیاز وجود دارد:

- نوع باید یک ساختار باشد (نوع مقداری)؛ شما نمی‌توانید یک آرایه را درون یک کلاس (نوع ارجاعی) تعبیه کنید.
- فیلد یا ساختار تعریف‌کننده‌ی آن باید با کلمه کلیدی **unsafe** علامت زده شود.
- فیلد آرایه باید با کلمه کلیدی **fixed** علامت زده شود.
- آرایه باید تک‌بعدی پایه صفر باشد.
- نوع عنصر آرایه باید یکی از این نوع‌ها باشد: **Single**، **UInt64**، **Int64**، **UInt32**، **Int32**، **Byte**، **Sbyte**، **Char**، **Boolean** یا **Double**.

آرایه‌های خطی (**inline**) نوعاً برای سناریوهایی است که نیاز به کار با کد مدیریت نشده دارد و ساختمان داده‌ی مدیریت نشده نیز یک آرایه خطی داشته باشد. هر چند، آرایه‌های خطی می‌توانند در سناریوهای دیگری نیز استفاده شوند. متد **InlineArrayDemo** که در کد قبلی نشان داده شد یک مثال از چگونگی استفاده از آرایه خطی است. متد **InlineArrayDemo** همان کاری را می‌کند که متد **StackallocDemo** انجام می‌دهد، فقط این کار را به روشی متفاوت انجام می‌دهد.

فصل ۱۷: نماینده ها

در این فصل، من درباره‌ی توابع کالباک (callback) صحبت می‌کنم. توابع کالباک مکانیزم برنامه‌نویسی فوق العاده مفیدی هستند که سال‌ها مورد استفاده-اند. دات‌نت فریمورک میکروسافت مکانیزم یک تابع کالباک را با استفاده از نماینده‌ها delegates ارائه می‌کند. بر خلاف مکانیزم‌های کالباک استفاده شده در دیگر پلتفرم‌ها مثل ++C مدیریت نشده، نماینده‌ها کاربرد بسیار بیشتری ارائه می‌کنند. برای نمونه، نماینده‌ها مطمئن می‌شوند متد کالباک نوع-امن است تا یکی از مهمترین اهداف CLR حفظ شود. نماینده‌ها همچنین قابلیت فراخوانی چندین متد به صورت سریالی را دارند و از فراخوانی متدهای استاتیک و متدهای نمونه پشتیبانی می‌کنند.

نگاه ابتدایی به نماینده ها

متد **qsort** از **C** یک اشاره‌گر به یک تابع کالباک می‌گیرد تا عناصر درون یک آرایه را مرتب کند. در ویندوز میکروسافت، توابع کالباک برای رویه‌های پنجره، رویه‌های **hook**، فراخوانی رویه‌های غیرهمزمان و غیره نیاز هستند. در دات‌نت فریمورک، متدهای کالباک برای چیزهای بسیار زیادی استفاده می‌شوند. برای نمونه، شما می‌توانید متدهای کالباک را ثبت کنید تا تنوعی از اطلاع رسانی‌ها را از اکسپشن‌های مدیریت نشده، تغییرات وضعیت پنجره، انتخاب آیتم‌های منو، تغییرات سیستم فایل، رویدادهای کنترل فرم و عملیات‌های غیرهمزمان کامل شده، دریافت کنید.

در **C/C++** مدیریت نشده، آدرس یک تابع غیر عضو، تنها یک آدرس حافظه است. این آدرس هیچ گونه اطلاعات اضافی مثل تعداد پارامترهایی که تابع انتظار دارد، نوع این پارامترها، نوع مقدار برگشتی تابع و قرارداد فراخوانی تابع ندارد. به طور خلاصه، توابع کالباک **C/C++** مدیریت نشده، نوع-امن نیستند (اگر چه مکانیزم‌های بسیار سبکی هستند).

در دات‌نت فریمورک، توابع کالباک به همان میزان که در برنامه‌نویسی مدیریت نشده ویندوز کاربردی و فراگیر هستند، سودمند نیز می‌باشند. به هر حال، دات‌نت فریمورک یک مکانیزم نوع-امن به نام نماینده‌ها delegates فراهم می‌کند. من بحث نماینده‌ها را با نشان دادن چگونگی استفاده از آن‌ها آغاز می‌کنم. کد زیر نشان می‌دهد چگونه نماینده‌ها را تعریف، ساخته و استفاده کنید.

```
using System;
```

```
using System.Windows.Forms;
```

```
using System.IO;
```

```
// Declare a delegate type; instances refer to a method that
```

```
// takes an Int32 parameter and returns void.
```

```
internal delegate void Feedback(Int32 value);
```

```
public sealed class Program {
```

```
    public static void Main() {
```

```
        StaticDelegateDemo();
```

```
        InstanceDelegateDemo();
```

```
        ChainDelegateDemo1(new Program());
```

```
        ChainDelegateDemo2(new Program());
```

```
    }
```

```
private static void StaticDelegateDemo() {
```

```
    Console.WriteLine("----- Static Delegate Demo -----");
```

```
    Counter(1, 3, null);
```

```
    Counter(1, 3, new Feedback(Program.FeedbackToConsole));
```

```
    Counter(1, 3, new Feedback(FeedbackToMsgBox)); // "Program." is optional
```

```
    Console.WriteLine();
```

```
}
```

```

private static void InstanceDelegateDemo() {
    Console.WriteLine("----- Instance Delegate Demo -----");
    Program p = new Program();
    Counter(1, 3, new Feedback(p.FeedbackToFile));

    Console.WriteLine();
}

private static void ChainDelegateDemo1(Program p) {
    Console.WriteLine("----- Chain Delegate Demo 1 -----");
    Feedback fb1 = new Feedback(FeedbackToConsole);
    Feedback fb2 = new Feedback(FeedbackToMsgBox);
    Feedback fb3 = new Feedback(p.FeedbackToFile);

    Feedback fbChain = null;
    fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
    fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
    fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
    Counter(1, 2, fbChain);

    Console.WriteLine();
    fbChain = (Feedback)
    Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
    Counter(1, 2, fbChain);
}

private static void ChainDelegateDemo2(Program p) {
    Console.WriteLine("----- Chain Delegate Demo 2 -----");
    Feedback fb1 = new Feedback(FeedbackToConsole);
    Feedback fb2 = new Feedback(FeedbackToMsgBox);
    Feedback fb3 = new Feedback(p.FeedbackToFile);

    Feedback fbChain = null;
    fbChain += fb1;
    fbChain += fb2;
    fbChain += fb3;

    Counter(1, 2, fbChain);
    Console.WriteLine();
    fbChain -= new Feedback(FeedbackToMsgBox);
    Counter(1, 2, fbChain);
}

private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // If any callbacks are specified, call them
        if (fb != null)

```



```

        fb(val);
    }
}

private static void FeedbackToConsole(Int32 value) {
    Console.WriteLine("Item=" + value);
}

private static void FeedbackToMsgBox(Int32 value) {
    MessageBox.Show("Item=" + value);
}

private void FeedbackToFile(Int32 value) {
    StreamWriter sw = new StreamWriter("Status", true);
    sw.WriteLine("Item=" + value);
    sw.Close();
}
}

```

حال توضیح خواهیم داد این کد چه کار می‌کند. در بالا، به تعریف نماینده درونی **Feedback** توجه کنید. یک نماینده، امضای یک متد کالباک را تعیین می‌کند. در این مثال، یک نماینده **Feedback**، یک متد که یک پارامتر (یک **Int32**) می‌گیرد و **void** برمی‌گرداند را شناسایی می‌کند. به طریقی، یک نماینده بسیار شبیه یک **typedef** در **C/C++** مدیریت نشده است که آدرس یک تابع را نمایش می‌دهد.

کلاس **Program** یک متد استاتیک خصوصی به نام **Counter** تعریف می‌کند. این متد اعداد صحیح را از آرگومان **from** تا آرگومان **to** می‌شمارد. متد **Counter** یک **fb** نیز می‌گیرد، که یک ارجاع به یک شی نماینده **Feedback** است. **Counter** بین تمام اعداد صحیح حرکت کرده و برای هر عدد صحیح، اگر **fb**، **null** نباشد، متد کالباک (تعیین شده توسط متغیر **fb**) فراخوانی می‌شود. به این متد کالباک مقدار آیتم در حال پردازش که شماره آیتم است، ارسال می‌شود. متد کالباک می‌تواند طراحی و پیاده‌سازی شود تا به هر روش که صحیح باشد هر آیتم را پردازش کند.

استفاده از نماینده ها برای کالباک کردن (Call Back) متدهای استاتیک

حال که شما فهمیدید متد **Counter** چگونه طراحی شده است و چگونه کار می‌کند، بگذارید ببینیم چگونه از نماینده‌ها برای کالباک کردن (call back) متدهای استاتیک استفاده کنیم. متد **StaticDelegateDemo** که در نمونه کد قبلی نمایش داده شد، هدف این بخش است.

متد **StaticDelegateDemo** متد **Counter** را فراخوانی کرده، **null** را به عنوان سومین پارامتر ارسال می‌کند که متناظر با پارامتر **fb** از **Counter** است. چون پارامتر **fb** از **Counter**، **null** دریافت می‌کند، هر آیتم بدون فراخوانی هیچ گونه متد کالباکی پردازش می‌شود.

بعد، متد **StaticDelegateDemo** متد **Counter** را برای بار دوم فراخوانی کرده شی نماینده **Feedback** که جدیداً ساخته شده است را به عنوان سومین پارامتر بدان ارسال می‌کند. این شی نماینده یک پوشش در اطراف یک متد است که به متد اجازه می‌دهد به صورت غیر مستقیم از طریق پوشاننده، کالباک شود. در این مثال، نام این متد استاتیک، **Program.FeedbackToConsole**، به سازنده‌ی نوع **Feedback** ارسال می‌گردد که بیان می‌کند این متدی است که باید پوشاننده شود. اشاره‌گر برگردانده شده از عملگر **new** به عنوان سومین پارامتر به **Counter** ارسال می‌شود. حال وقتی **Counter** اجرا می‌گردد، متد استاتیک **FeedbackToConsole** از نوع **Program** را برای هر آیتم در دنباله، فراخوانی خواهد کرد. **FeedbackToConsole** به سادگی یک رشته را در کنسول می‌نویسد تا نشان دهد آیتم در حال پردازش است.

نکته متد **FeedbackToConsole** درون نوع **Program** به عنوان **private** تعریف شده است اما متد **Counter** قادر است متد خصوصی متعلق به **Program** را فراخوانی کند. در این مورد، شاید شما انتظار مشکلی را نداشته باشید چون هر دوی **Counter** یا **FeedbackToConsole** در یک نوع تعریف شده‌اند. هر چند، این کد اگر هم متد **Counter** در نوع دیگری تعریف شده باشد باز هم خوب کار می‌کند. به طور خلاصه، برای یک نوع، این نقض امنیت یا دسترس پذیری نیست اگر کدی داشته باشد که عضو خصوصی نوع دیگری را از طریق یک نماینده فراخوانی کند تا زمانی که شی نماینده توسط کدی ساخته شده است که امنیت/دسترس پذیری وسیع تری دارد.

سومین فراخوانی به **Counter** در متد **StaticDelegateDemo** تقریباً معادل دومی است. تنها تفاوت اینست که شی نماینده **Feedback** متد استاتیک **Program.FeedbackToMsgBox** را می‌پوشاند. **FeedbackToMsgBox** یک رشته می‌سازد که نشان می‌دهد آیتم در حال پردازش است. آنگاه رشته در یک جعبه پیام، نمایش داده می‌شود.

همه چیز در این مثال نوع-امن است. برای نمونه، هنگام ساخت یک شی نماینده **Feedback**، کامپایلر مطمئن می‌شود که امضای متدهای **FeedbackToConsole** و **FeedbackToMsgBox** از **Program** با امضای تعریف شده توسط نماینده **Feedback** سازگار است. بخصوص، هر دو متد باید یک آرگومان (یک **Int32**) بگیرند و هر دو متد باید نوع برگشتی یکسانی (**void**) داشته باشند. اگر **FeedbackToConsole** اینگونه تعریف شده بود:

```
private static Boolean FeedbackToConsole(String value) {
    ...
}
```

کامپایلر سی‌شارپ کد را کامپایل نکرده و خطای زیر را اعلام می‌کرد:

“error CS0123: No overload for ‘FeedbackToConsole’ matches delegate ‘Feedback’.”

هر دوی سی‌شارپ و CLR اجازه‌ی کواریانس و کنتراواریانس نوع‌های ارجاعی هنگام اتصال یک متد به یک نماینده را می‌دهند. کواریانس **covariance** یعنی یک متد می‌تواند نوعی برگرداند که مشتق شده از نوع برگشتی نماینده باشد. کنتراواریانس **contra-variance** یعنی یک متد می‌تواند پارامتری بگیرد که یک نوع پایه برای نوع پارامتر نماینده باشد. برای مثال، اگر نماینده‌ای شبیه به این داشته باشیم:

```
delegate Object MyCallback(FileStream s);
```

این امکان وجود دارد که یک نمونه از این نوع نماینده بسازیم که به متدی که دارای فرم کلی زیر است متصل شود:

```
String SomeMethod(Stream s);
```

در اینجا، نوع برگشتی **SomeMethod (String)** نوعی است که از نوع برگشتی نماینده **(Object)** مشتق شده است. این کواریانس، مجاز است. نوع پارامتر **SomeMethod (Stream)** نوعی است که یک کلاس پایه برای نوع پارامتر نماینده **(FileStream)** است، این کنتراواریانس، مجاز است. توجه کنید که کواریانس و کنتراواریانس تنها برای نوع‌های ارجاعی پشتیبانی می‌شوند، نه برای نوع‌های مقداری یا برای **void**. پس برای مثال، من نمی‌توانم متد زیر را به نماینده **MyCallback** متصل کنم:

```
Int32 SomeOtherMethod(Stream s);
```

اگرچه نوع برگشتی **SomeOtherMethod (Int32)** از نوع برگشتی **MyCallback (Object)** مشتق شده است، این فرم از کواریانس مجاز نیست چون **Int32** یک نوع مقداری است. واضح است که علت اینکه چرا نوع‌های مقداری و **void** نمی‌توانند برای کواریانس و کنتراواریانس استفاده شوند، اینست که ساختار حافظه برای آن‌ها متفاوت است در حالیکه ساختار حافظه برای نوع‌های ارجاعی همیشه یک اشاره‌گر است. خوشبختانه، کامپایلر سی‌شارپ در صورتی که شما سعی کنید چیزی که پشتیبانی نمی‌شود را انجام دهید، اعلام خطا می‌کند.

استفاده از نماینده‌ها برای کالک کردن (Call back) متدهای نمونه

من توضیح دادم چگونه نماینده‌ها می‌توانند برای فراخوانی متدهای استاتیک استفاده شوند، اما آن‌ها می‌توانند برای فراخوانی متدهای نمونه برای یک شی خاص نیز استفاده شوند. برای آنکه درک کنید چگونه کالک کردن یک متد نمونه کار می‌کند، به متد **InstanceDelegateDemo** که در کدی که در ابتدای فصل نشان دادم، قرارداد را نگاه کنید.

توجه کنید که یک شی **Program** به نام **p** در متد **InstanceDelegateDemo** ساخته می‌شود. این شی **Program** هیچ فیلد نمونه یا ویژگی همراه ندارد. من آن را فقط برای اهداف نمایشی، درست کرده‌ام. وقتی شی نماینده **Feedback** جدید در فراخوانی متد **Counter** ساخته می‌شود، به سازنده‌اش **p.FeedbackToFile** ارسال می‌گردد. این باعث می‌شود نماینده یک اشاره‌گر به متد **FeedbackToFile** که یک متد نمونه (نه یک متد استاتیک) است را بپوشاند. وقتی **Counter** متد کالیکی که توسط آرگومان **fb** شناسایی می‌شود را فراخوانی می‌کند، متد نمونه **FeedbackToFile** فراخوانی می‌گردد و آدرس شی اخیراً ساخته شده‌ی **p** به عنوان آرگومان ضمنی **this** به متد نمونه ارسال می‌گردد.

متد **FeedbackToFile** شبیه متدهای **FeedbackToConsole** و **FeedbackToMsgBox** کار می‌کند به جز آنکه یک فایل را باز کرده و رشته را به انتهای فایل اضافه می‌کند. (فایل **Status** که متد می‌سازد را می‌توان در دایرکتوری **AppBase** برنامه پیدا کرد).

مجدداً هدف این مثال نمایش این بود که نماینده‌ها می‌توانند فراخوانی به متدهای نمونه و همچنین متدهای استاتیک را بپوشانند. برای متدهای نمونه، نماینده نیاز دارد نمونه شی‌ای که متد می‌خواهد روی آن عمل کند را بداند. پوشاندن یک متد نمونه مفید است چون کد درون شی می‌تواند به اعضای نمونه شی دسترسی داشته باشد. این یعنی شما می‌توانید وضعیت‌هایی داشته باشید که در حالیکه متد کالیک پردازشش را انجام می‌دهد، مورد استفاده قرار گیرد.

روشن کردن موضوع نماینده‌ها

در نگاه اول، استفاده از نماینده‌ها آسان به نظر می‌رسد: شما آن‌ها را با استفاده از کلمه کلید **delegate** سی‌شارپ تعریف کرده، نمونه‌هایی از آن‌ها را با استفاده از عملگر آشنای **new** می‌سازید و کالیک را با نحوی شبیه فراخوانی متد، فراخوانی (یا درخواست، **invoke**) می‌کنید (جز آنکه، به جای نام یک متد، شما از متغیری که به شی نماینده اشاره دارد استفاده می‌کنید).

هرچند، آنچه واقعاً اتفاق می‌افتد بسی پیچیده‌تر از آنی است که مثال‌های اولیه نشان دادند. کامپایلرها و CLR پردازش‌های پشت صحنه‌ی فراوانی انجام می‌دهند تا پیچیدگی را مخفی کنند. در این بخش، من بر این موضوع که کامپایلر و CLR چگونه با هم کار می‌کنند تا نماینده‌ها را پیاده‌سازی کنند، تمرکز می‌کنم. دانستن این موضوع درک شما از نماینده‌ها را بالا برده و به شما کمک می‌کند چگونه از آن‌ها به صورت موثر و کارا استفاده کنید. من همچنین به برخی از ویژگی‌های اضافی که نماینده‌ها فراهم می‌کنند اشاره می‌کنم.

بگذارید با بررسی این کد آغاز کنیم:

```
internal delegate void Feedback(Int32 value);
```

کامپایلر وقتی این خط را می‌بیند، در حقیقت یک کلاس کاملاً شبیه به این تولید می‌کند:

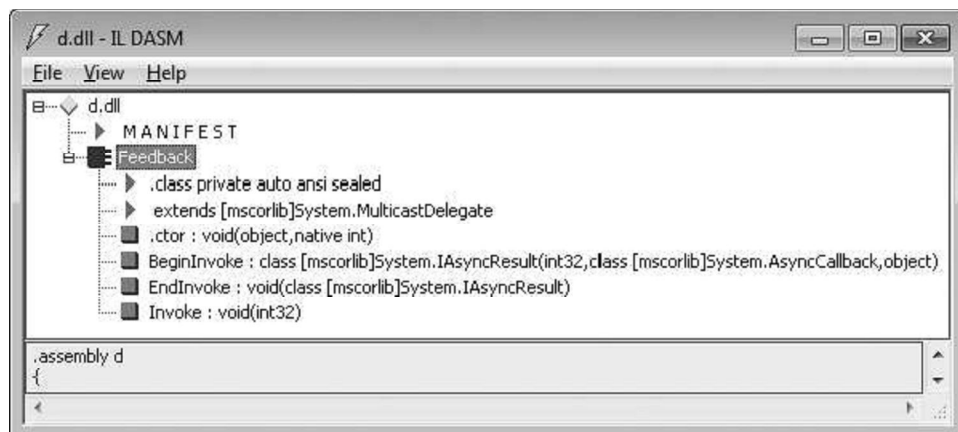
```
internal class Feedback : System.MulticastDelegate {
    // Constructor
    public Feedback(Object object, IntPtr method);

    // Method with same prototype as specified by the source code
    public virtual void Invoke(Int32 value);

    // Methods allowing the callback to be called asynchronously
    public virtual IAsyncResult BeginInvoke(Int32 value,
        AsyncCallback callback, Object object);
    public virtual void EndInvoke(IAsyncResult result);
}
```

کلاس تعریف شده توسط کامپایلر چهار متد دارد: یک سازنده، **Invoke**، **BeginInvoke** و **EndInvoke**. در این فصل، من بر متدهای سازنده و **Invoke** تمرکز می‌کنم. در فصل ۲۷ "عملیات‌های همزمان وابسته به ورودی خروجی" در بخش "عملیات‌های APM و وابسته به پردازش"، متدهای **BeginInvoke** و **EndInvoke** را توضیح می‌دهم.

در واقع شما می‌توانید با بررسی اسمبلی تولیدی، توسط **ILDasm.exe** همانگونه که در شکل ۱-۱۷ نشان داده شده است، آنچه کامپایلر به صورت خودکار تولید می‌کند را ببینید.



شکل ۱-۱۷ ILDasm.exe در حال نمایش متادیتای تولیدی توسط کامپایلر برای نماینده

در این مثال، کامپایلر یک کلاس به نام **Feedback** تعریف کرده که از نوع **System.MulticastDelegate** که در کتابخانه کلاس فریمورک (FCL) تعریف شده، مشتق شده است. (تمام نوع‌های نماینده از **MulticastDelegate** مشتق می‌شوند.)

مهم کلاس **System.MulticastDelegate** از **System.Delegate** مشتق شده که خودش از **System.Object** مشتق گردیده است. دلیل اینکه چرا دو کلاس نماینده وجود دارد تاریخی بوده و مایه تاسف است؛ تنها باید یک کلاس نماینده در FCL موجود باشد. متأسفانه، شما باید از هر دو کلاس آگاه باشید چون گرچه تمام نوع‌های نماینده که شما می‌سازید کلاس **MulticastDelegate** را به عنوان کلاس پایه دارند، شما ندرتا نوع‌های نماینده خود را با استفاده از متدهای تعریف شده توسط کلاس **Delegate** به جای کلاس **MulticastDelegate** دستکاری می‌کنید. برای نمونه، کلاس **Delegate** متدهای استاتیکی به نام **Combine** و **Remove** دارد. (من بعداً توضیح می‌دهم این متدها چه کاری انجام می‌دهند). امضای هر دوی این متدها بیان می‌کند که آن‌ها پارامترهای **Delegate** می‌گیرند. چون نوع نماینده شما از **MulticastDelegate** مشتق شده که آن هم از **Delegate** مشتق می‌شود، نمونه‌های نوع نماینده شما می‌توانند به این متدها ارسال شوند.

کلاس دارای پدیداری خصوصی است چون نماینده به عنوان **internal** در سورس کد تعریف شده است. اگر سورس کد پدیداری **public** را تعیین کرده بود، کلاس **Feedback** که کامپایلر تولید می‌کرد نیز عمومی می‌بود. شما باید بدانید که نوع‌های نماینده می‌توانند درون یک نوع (تودرتو درون نوعی دیگر)، یا در میدان سراسری تعریف شوند. اساساً، چون نماینده‌ها کلاس هستند، یک نماینده هر جایی که یک کلاس بتواند تعریف شود، می‌تواند تعریف شود. چون تمام نوع‌های نماینده از **MulticastDelegate** مشتق شده‌اند، آن‌ها فیلدها، ویژگی‌ها و متدهای **MulticastDelegate** را به ارث می‌برند. از تمام این اعضا، سه فیلد غیرعمومی مهم‌ترین آن‌ها هستند.

جدول ۱-۱۷ این سه فیلد مهم را نشان می‌دهد.

جدول ۱-۱۷ فیلدهای مهم غیرعمومی از **MulticastDelegate**

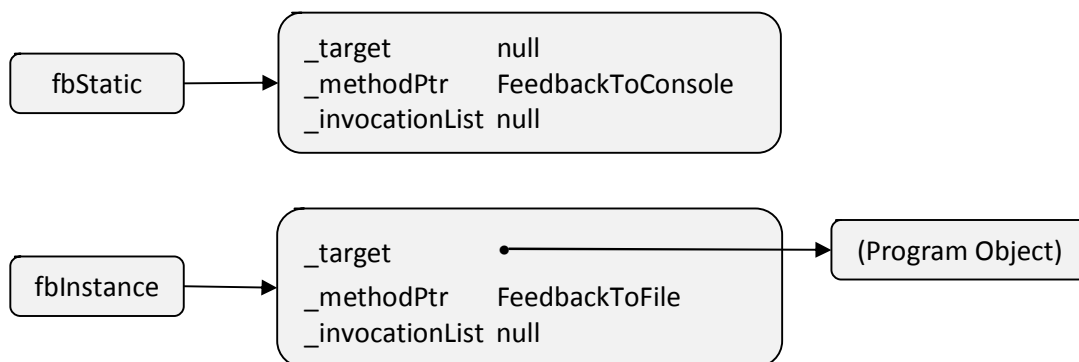
فیلد	نوع	توضیح
_target	System.Object	وقتی شی نماینده یک متد استاتیک را می‌پوشاند، این فیلد null است. وقتی شی نماینده یک متد نمونه را می‌پوشاند، این فیلد به شی‌ای که وقتی متد کالک فراخوانی می‌شود باید روی آن عمل انجام شود اشاره دارد. به بیان دیگر، این فیلد، مقداری که باید برای پارامتر ضمنی this به متد نمونه ارسال گردد را تعیین می‌کند.
_methodPtr	System.IntPtr	یک عدد صحیح که CLR برای شناسایی متدی که باید کالک شود استفاده می‌کند.
_invocationList	System.Object	این فیلد معمولاً null است. آن می‌تواند به یک آرایه از نماینده‌ها هنگام ساخت یک زنجیره‌ی نماینده اشاره کند (بعداً در این فصل بحث می‌شود).

توجه کنید تمام نماینده‌ها دارای یک سازنده هستند که دو پارامتر می‌گیرد: یک ارجاع به یک شی و یک عدد صحیح که به متد کالیک اشاره دارد. هر چند، اگر شما سورس کد را بررسی کنید، شما خواهید دید که من مقادیری مثل **Program.FeedbackToConsole** یا **p.FeedbackToFile** را ارسال می‌کنم. هر آنچه درباره‌ی برنامه‌نویسی فرا گرفته اید به شما می‌گوید که این کد نباید کامپایل شود! هر چند، کامپایلر سی‌شارپ می‌داند که یک نماینده در حال ساخت است و سورس کد را تجزیه می‌کند تا تعیین نماید به چه شی و متدی اشاره می‌شود. یک ارجاع به شی برای پارامتر **object** سازنده و یک مقدار خاص (**IntPtr** خاص (بدست آمده از نشانه متادیتای **MethodDef** یا **MemberRef**) که متد را شناسایی می‌کند، برای پارامتر **method** ارسال می‌گردد. برای متدهای استاتیک، **null** برای پارامترهای **object** ارسال می‌شود. درون سازنده، این دو آرگومان به ترتیب در فیلدهای خصوصی **_target** و **_methodPtr** ذخیره می‌شوند. به علاوه، سازنده، فیلد **_invocationList** را به **null** تنظیم می‌کند. من توضیح فیلد **_invocationList** را تا بخش بعدی "استفاده از نماینده‌ها برای فراخوانی چندین متد (زنجیره‌ای)" به تعویق می‌اندازم.

پس هر شی نماینده در واقع یک پوشاننده حول یک متد و یک شی که وقتی متد فراخوانی می‌شود روی آن عمل می‌کند است. پس من اگر دو خط کد شبیه به این داشته باشم:

```
Feedback fbStatic = new Feedback(Program.FeedbackToConsole);
Feedback fbInstance = new Feedback(new Program().FeedbackToFile);
```

متغیرهای **fbInstance** و **fbStatic** به دو شی نماینده **Feedback** مجزا که مقداردهی اولیه شده‌اند اشاره می‌کنند؛ همانطور که در شکل ۲-۱۷ نشان داده شده است.



شکل ۲-۱۷ یک متغیر که به یک نماینده که به یک متد استاتیک اشاره می‌کند و یک متغیر که به یک متد نمونه اشاره می‌کند.

کلاس **Delegate** دو ویژگی نمونه عمومی فقط خواندنی تعریف می‌کند: **Method Target**. با داشتن یک ارجاع به یک شی نماینده، شما می‌توانید این دو ویژگی را بخوانید. ویژگی **Target** یک اشاره‌گر به شی‌ای برمی‌گرداند که اگر متد، کالیک شود روی آن شی عمل می‌کند. اساساً، ویژگی **Target** مقدار ذخیره شده در فیلد خصوصی **_target** را برمی‌گرداند. اگر شی نماینده یک متد استاتیک را بیوشاند، **Target**، **null** برمی‌گرداند. ویژگی **Method** یک ارجاع به یک شی **System.Reflection.MethodInfo** برمی‌گرداند که متد کالیک را شناسایی می‌کند. اساساً، ویژگی **Method**، یک مکانیزم داخلی دارد که مقدار ذخیره شده در فیلد خصوصی **_methodPtr** را به یک شی **MethodInfo** تبدیل کرده و آن را برمی‌گرداند.

شما می‌توانید از این اطلاعات به روش‌های مختلف استفاده کنید. برای نمونه، شما می‌توانید ببینید آیا یک شی نماینده به یک متد نمونه از یک نوع خاص اشاره دارد:

```
Boolean DelegateRefersToInstanceMethodOfType(MulticastDelegate d, Type type) {
    return((d.Target != null) && d.Target.GetType() == type);
}
```

شما همچنین می‌توانید کدی بنویسید که بررسی کند آیا متد کالیک یک نام مخصوص دارد (مثل **FeedbackToMsgBox**):

```
Boolean DelegateRefersToMethodOfName(MulticastDelegate d, String methodName) {
    return(d.Method.Name == methodName);
}
```

استفاده‌های فراوان دیگری می‌توان از این ویژگی‌ها کرد.

حال که شما می‌دانید اشیاء نماینده چگونه ساخته می‌شوند و ساختار درونی آن‌ها شبیه چیست، بگذارید درباره‌ی اینکه متد کالیک چگونه درخواست (فراخوانی) می‌شود صحبت کنیم. برای راحتی، من کد متد **Counter** را در اینجا تکرار کرده‌ام:

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // If any callbacks are specified, call them
        if (fb != null)
            fb(val);
    }
}
```

به خط زیر کامنت در کد نگاه کنید. عبارت **if** ابتدا بررسی می‌کند ببیند که **fb**، **null** نباشد. اگر **fb**، **null** نباشد، در خط بعدی، شما کدی می‌بینید که متد کالیک را فراخوانی می‌کند. بررسی **null** بودن نیاز است چون **fb** تنها یک متغیر است که می‌تواند به یک شی نماینده **Feedback** اشاره کند. آن می‌تواند **null** هم باشد. شاید به نظر برسد که گویا من یک تابع به نام **fb** را فراخوانی کرده و یک پارامتر (**val**) بدان ارسال می‌کنم. هرچند، تابعی به نام **fb** وجود ندارد. مجدداً، چون می‌داند که **fb** یک متغیر است که به یک شی نماینده اشاره دارد، کامپایلر کدی تولید می‌کند تا متد **Invoke** از شی نماینده را فراخوانی کند. به بیان دیگر، کامپایلر این را می‌بیند:

```
fb(val);
```

اما کامپایلر کدی تولید می‌کند که گویا شما کد زیر را نوشته اید:

```
fb.Invoke(val);
```

شما می‌توانید با استفاده از **ILDasm.exe** کد IL تولید شده برای متد **Counter** را بررسی کرده و ببینید که کامپایلر کدی تولید می‌کند که متد **Invoke** از نوع نماینده را فراخوانی کند. کد IL برای متد **Counter** در اینجا آمده است. دستور موجود در **IL_0009** در شکل، فراخوانی به متد **Invoke** از **Feedback** را بیان می‌کند:

```
.method private hidebysig static void Counter(    int32 from,
                                                int32 'to',
                                                class Feedback fb) cil managed
{
    // Code size      23 (0x17)
    .maxstack 2
    .locals init (int32 v_0)
    IL_0000: ldarg.0
    IL_0001: stloc.0
    IL_0002: br.s          IL_0012
    IL_0004: ldarg.2
    IL_0005: brfalse.s    IL_000e
    IL_0007: ldarg.2
    IL_0008: ldloc.0
    IL_0009: callvirt     instance void Feedback::Invoke(int32)
    IL_000e: ldloc.0
    IL_000f: ldc.i4.1
    IL_0010: add
    IL_0011: stloc.0
    IL_0012: ldloc.0
    IL_0013: ldarg.1
    IL_0014: b!e.s          IL_0004
    IL_0016: ret
} // end of method Program::Counter
```

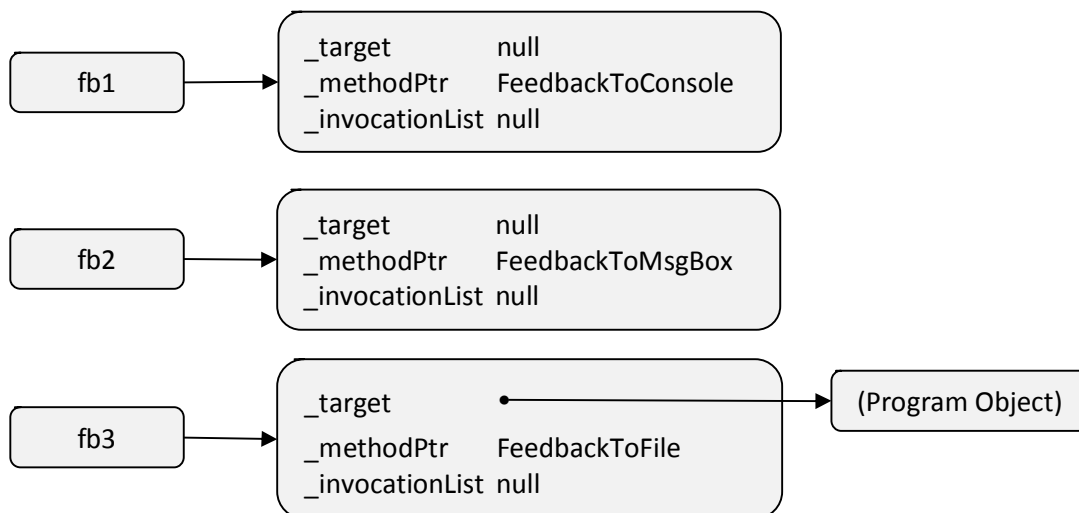
در واقع، شما می‌توانید متد **Counter** را تغییر دهید تا صریحاً **Invoke** را فراخوانی کند، مثل این:

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // If any callbacks are specified, call them
        if (fb != null)
            fb.Invoke(val);
    }
}
```

شما به خاطر دارید که کامپایلر وقتی کلاس **Feedback** را تعریف کرد متد **Invoke** را نیز تعریف کرد. وقتی **Invoke** فراخوانی می‌شود، آن از فیلدهای خصوصی **_target** و **_methodPtr** برای فراخوانی متد مورد نظر روی شی تعیین شده استفاده می‌کند. توجه کنید که امضای متد **Invoke** با امضای نماینده مطابقت دارد؛ چون نماینده **Feedback** یک پارامتر **Int32** گرفته و **void** برمی‌گرداند، متد **Invoke** (تولید شده توسط کامپایلر) یک پارامتر **Int32** گرفته و **void** برمی‌گرداند.

استفاده از نماینده‌ها برای فراخوانی چند متد (زنجیربندی)

نماینده‌ها به خودی خود بسیار مفید هستند. اما با افزودن پشتیبانی آن‌ها از زنجیربندی، نماینده‌ها را باز هم مفیدتر می‌کند. زنجیربندی **Chaining**، یک مجموعه از اشیاء نماینده است و آن، قابلیت درخواست یا فراخوانی تمام متدهای ارائه شده توسط نماینده‌ها در مجموعه را می‌دهد. برای درک این، متد **ChainDelegateDemo1** که در کدی که ابتدای فصل نشان داده‌ام آمده است را ببینید. در این متد، بعد از عبارت **Console.WriteLine**، من سه شی نماینده ساختم و متغیرهای **fb1**، **fb2** و **fb3** - را دارم که به هر شی اشاره می‌کنند. همانگونه که در شکل ۱۷-۳ نشان داده شده است.

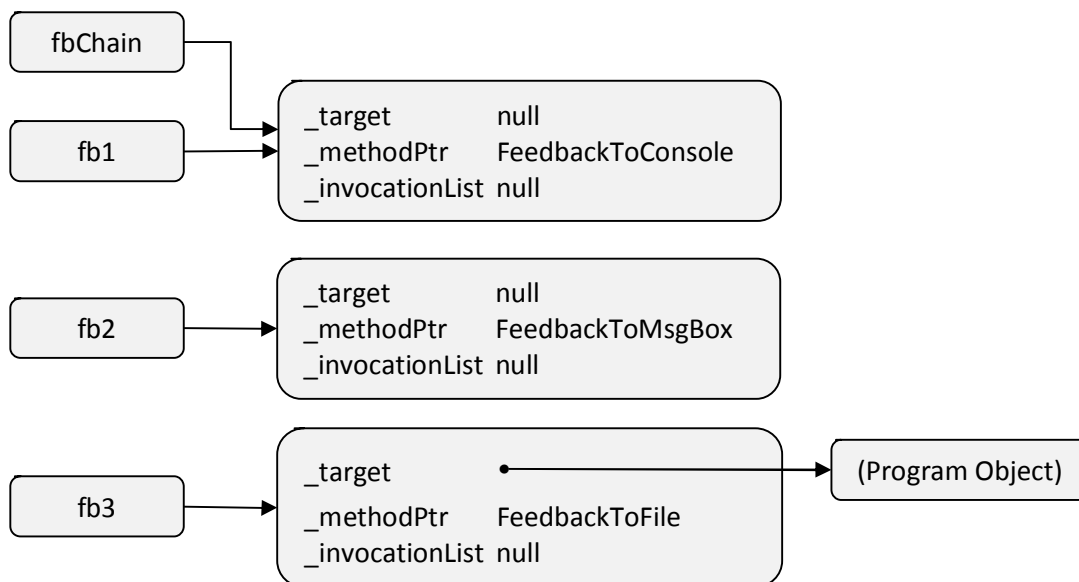


شکل ۱۷-۳ وضعیت ابتدایی اشیاء نماینده که توسط متغیرهای **fb1**، **fb2** و **fb3** ارجاع می‌شوند.

متغیر اشاره‌گر به یک شی نماینده **fbChain**، **Feedback**، قصد دارد به یک زنجیر یا مجموعه از اشیاء نماینده اشاره کند که متدهایی که باید کالک شوند را می‌پوشاند. مقداره‌ی اولیه **fbChain** به **null** بیان می‌کند که در حال حاضر متدی برای کالک شدن وجود ندارد. متد عمومی استاتیک **Combine** از کلاس **Delegate** برای افزودن یک نماینده به زنجیر استفاده می‌شود:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
```

وقتی این خط از کد اجرا می‌شود، متد **Combine** می‌بیند که ما قصد داریم **null** و **fb2** را ترکیب کنیم. در درون، به سادگی **Combine** فقط مقدار **fb1** را برمی‌گرداند و متغیر **fbChain** به همان شی نماینده که توسط متغیر **fb2** ارجاع می‌شود، اشاره می‌کند که در شکل ۱۷-۴ نشان داده شده است.



شکل ۴-۱۷ وضعیت اشیاء نماینده بعد از درج اولین نماینده در زنجیر

برای افزودن نماینده‌ی دیگری به زنجیر، متد **Combine** مجدداً فراخوانی می‌شود:

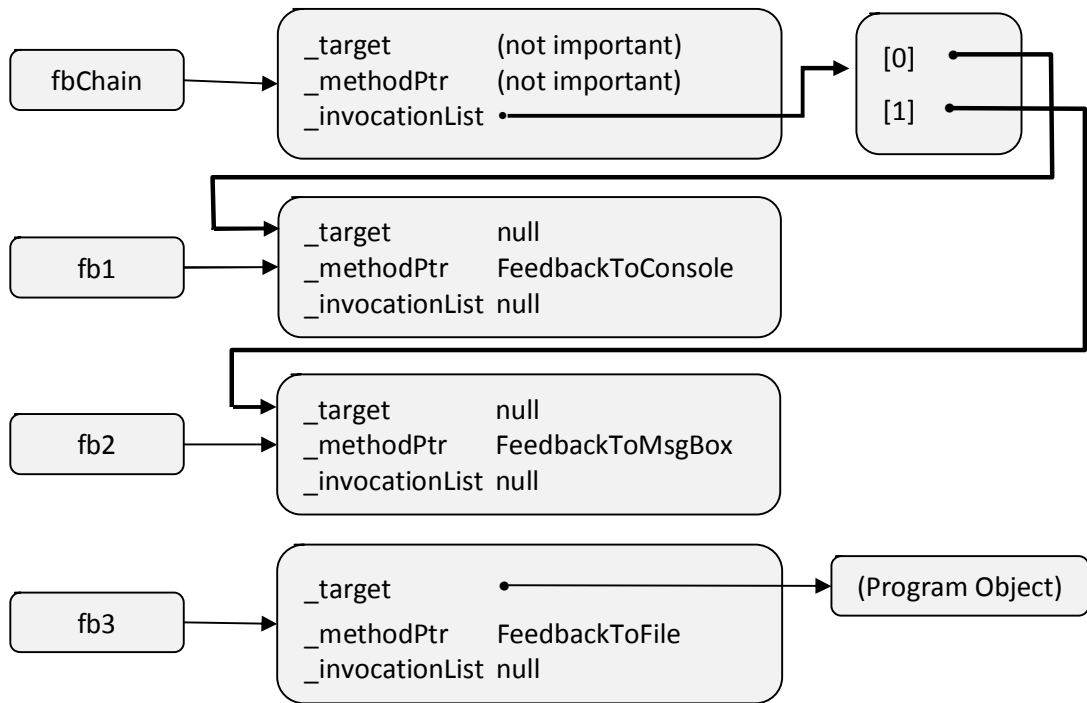
```
fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
```

در درون، متد **Combine** می‌بیند که **fbChain** قبلاً به یک شی نماینده اشاره دارد، پس **Combine** یک شی نماینده جدید خواهد ساخت. این شی جدید فیلدهای خصوصی **_target** و **_methodPtr** خود را با مقادیری مقداردهی اولیه می‌کند که برای این بحث مهم نیستند. اما آنچه مهم است اینست که فیلد **_invocationList** مقداردهی اولیه می‌شود تا به یک آرایه از اشیاء نماینده اشاره کند. اولین عنصر این آرایه (اندیس 0) مقداردهی اولیه می‌شود تا به نماینده‌ای اشاره کند که متد **FeedbackToConsole** را می‌پوشاند (این نماینده‌ای است که **fbChain** اکنون به آن اشاره دارد). دومین عنصر آرایه (اندیس 1) مقداردهی اولیه می‌شود تا به نماینده‌ای اشاره کند که متد **FeedbackToMsgBox** را می‌پوشاند (این نماینده‌ای است که **fb2** به آن اشاره می‌کند). سرانجام **fbChain** تنظیم می‌شود تا به شی نماینده جدیداً ساخته شده که در شکل ۵-۱۷ نمایش داده شده اشاره کند.

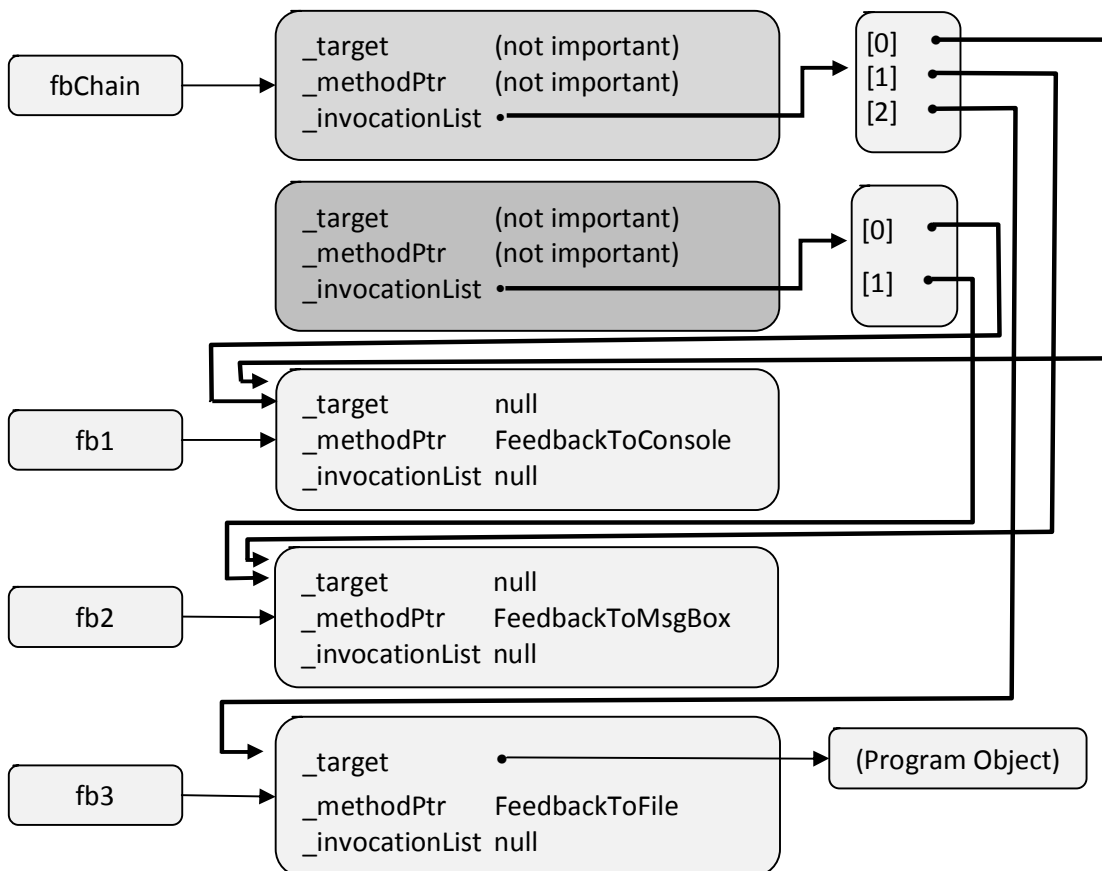
برای افزودن سومین نماینده به زنجیر، متد **Combine** یکبار دیگر فراخوانی می‌شود:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
```

مجدداً، **Combine** می‌بیند که **fbChain** قبلاً به یک شی نماینده اشاره می‌کند و این باعث می‌شود یک شی نماینده جدید ساخته شود، که در شکل ۶-۱۷ نشان داده شده است. همانند قبل، این شی نماینده جدید فیلدهای خصوصی **_target** و **_methodPtr** خودش را به مقادیری که برای این بحث کم اهمیت هستند، مقداردهی اولیه می‌کند و فیلد **_invocationList** برای اشاره به یک آرایه از اشیاء نماینده مقداردهی اولیه می‌شود. اولین و دومین عناصر این آرایه (اندیس‌های 0 و 1) برای ارجاع به همان نماینده‌هایی که شی نماینده‌ی قبلی در آرایه‌اش بدان‌ها ارجاع می‌کرده، مقداردهی اولیه می‌شوند. سومین عنصر آرایه (اندیس 2) برای ارجاع به نماینده‌ای که متد **FeedbackToFile** را می‌پوشاند، مقداردهی اولیه می‌شود (این نماینده‌ای است که **fb3** بدان اشاره می‌کند). سرانجام، **fbChain** برای اشاره به این شی نماینده جدیداً ساخته شده، تنظیم می‌شود. توجه کنید که نماینده‌ای که قبلاً ساخته شده و آرایه‌ای که توسط فیلد **_invocationList** از آن نماینده اشاره می‌شد، اکنون کاندیداهای جمع‌آوری هستند.



شکل ۵-۱۷ وضعیت اشیاء نماینده بعد از درج دومین نماینده در زنجیر



شکل ۶-۱۷ وضعیت نهایی اشیاء نماینده وقتی زنجیر کامل است

بعد از آنکه کد برای برقرار کردن زنجیر اجرا شد، متغیر fbChain به متد Counter ارسال می‌شود:

```
Counter(1, 2, fbChain);
```

درون متد **Counter** کدی است که به صورت ضمنی متد **Invoke** را روی شی نماینده **Feedback** فراخوانی می‌کند که قبلاً این را توضیح دادم. وقتی **Invoke** روی شی‌ای که **fbChain** بدان اشاره دارد فراخوانی می‌شود، نماینده می‌بند که فیلد خصوصی **_invocationList**، **null** نیست. که باعث می‌شود حلقه‌ای را اجرا کند که در بین تمام عناصر آرایه حرکت کرده، متدی که توسط هر نماینده پوشانده می‌شود را فراخوانی کند. در این مثال، **FeedbackToConsole** فراخوانی می‌شود بعد از آن **FeedbackToMsgBox** و بعد از آن **FeedbackToFile**.

متد **Invoke** از **Feedback** اساساً شبیه به این پیاده‌سازی می‌شود(در شبه کد):

```
public void Invoke(Int32 value) {
    Delegate[] delegatesSet = _invocationList as Delegate[];
    if (delegatesSet != null) {
        // This delegate's array indicates the delegates that should be called
        foreach (Feedback d in delegatesSet)
            d(value); // Call each delegate
    } else {
        // This delegate identifies a single method to be called back
        // Call the callback method on the specified target object.
        _methodPtr.Invoke(_target, value);
        // The line above is an approximation of the actual code.
        // what really happens cannot be expressed in C#.
    }
}
```

توجه کنید که این امکان هم وجود دارد که با فراخوانی متد استاتیک عمومی **Remove** از **Delegate**، یک نماینده را از زنجیر حذف کنید. این در بخش پایانی متد **ChainDelegateDemo1** نشان داده شده است:

```
fbChain = (Feedback) Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
```

وقتی **Remove** فراخوانی می‌شود، آرایه نماینده را که درون شی نماینده که توسط اولین پارامتر ارجاع می‌شود (در مثال من **fbChain**) را (از انتها به سمت اندیس 0) اسکن می‌کند. **Remove** به دنبال یک نماینده می‌گردد که فیلدهای **_target** و **_methodPtr** آن با آن‌هایی که در دومین پارامتر (در مثال من، نماینده جدید **Feedback**) تعیین شده تطابق داشته باشد. اگر یک تطابق یافت شد و چندین آیتم در آرایه باقی ماند، یک شی نماینده جدید ساخته می‌شود — آرایه **_invocationList** ساخته و مقداردهی اولیه شده، به تمام آیتم‌های موجود در آرایه اصلی البته به جز آیتمی که می‌خواهد حذف شود، اشاره می‌کند — و یک ارجاع به این شی نماینده جدید برگردانده می‌شود. اگر شما تنها عنصر زنجیر را حذف کنید، **Remove**، **null** برمی‌گرداند. توجه کنید که هر فراخوانی به **Remove** تنها یک نماینده را از زنجیر حذف می‌کند، تمام نماینده‌هایی که فیلدهای **_target** و **_methodPtr** مطابق داشته باشند را حذف نمی‌کند.

تاکنون، من مثال‌هایی نشان داده‌ام که در آن‌ها نوع نماینده من، **Feedback**، به گونه‌ای تعریف شده است که یک مقدار برگشتی **void** دارد. هر چند، من می‌توانم نماینده **Feedback** خودم را اینچنین تعریف کنم:

```
public delegate Int32 Feedback(Int32 value); public delegate Int32 Feedback(Int32 value);
```

اگر این کار را بکنم، متد **Invoke** آن، در درون شبیه به این خواهد بود (مجدداً با شبه کد):

```
public Int32 Invoke(Int32 value) {
    Int32 result;
    Delegate[] delegatesSet = _invocationList as Delegate[];
    if (delegatesSet != null) {
        // This delegate's array indicates the delegates that should be called
        foreach (Feedback d in delegatesSet)
            result = d(value); // Call each delegate
    } else {
        // This delegate identifies a single method to be called back
        // Call the callback method on the specified target object.
    }
}
```

```

    result = _methodPtr.Invoke(_target, value);
    // The line above is an approximation of the actual code.
    // what really happens cannot be expressed in C#.
}
return result;
}

```

وقتی هر نماینده در آرایه فراخوانی می‌شود، مقدار برگشتی آن در متغیر **result** ذخیره می‌شود. وقتی حلقه کامل می‌شود، متغیر **result** تنها حاوی نتیجه-ی آخرین نماینده‌ای است که فراخوانی شده (مقادیر برگشتی قبلی از بین می‌روند)؛ این مقدار به کدی که **Invoke** را فراخوانی کرده، برگردانده می‌شود.

پشتیبانی سی شارپ برای زنجیرهای نماینده

برای ساده سازی کار برنامه‌نویسان سی‌شارپ، کامپایلر سی‌شارپ به صورت خودکار سربارگذاری‌هایی از عملگر += و -= برای نمونه‌های نوع نماینده فراهم می‌کند. این عملگرها به ترتیب **Delegate.Combine** و **Delegate.Remove** را فراخوانی می‌کنند. استفاده از این عملگرها، ساخت زنجیرهای نماینده را آسان می‌کند. متدهای **ChainDelegateDemo1** و **ChainDelegateDemo2** در سورس‌کد نشان داده شده در ابتدای این فصل، کد IL مطلقاً یکسانی تولید می‌کنند.

تنها تفاوت بین این متدها اینست که متد **ChainDelegateDemo2** سورس‌کد را با بهره بردن از عملگر += و -= سی‌شارپ، ساده می‌کند. اگر شما به مدرک نیاز دارید تا باور کنید کد IL تولیدی برای هر دو متد یکسان است، شما می‌توانید کد را ساخته و با استفاده از **ILDasm.exe** به IL هر دو متد نگاه کنید. این تایید می‌کند که کامپایلر سی‌شارپ در حقیقت عملگرهای += و -= را به ترتیب با فراخوانی به متدهای استاتیک **Combine** و **Remove** از نوع **Delegate**، جایگزین می‌کند.

کنترل بیشتر برای فراخوانی زنجیر نماینده

در اینجا، شما درک می‌کنید چگونه یک زنجیر از اشیاء نماینده ساخته و چگونه تمام اشیاء درون زنجیر را فراخوانی کنید. تمام آیتم‌های درون زنجیر فراخوانی (درخواست) می‌شوند چون متد **Invoke** از نوع نماینده حاوی کدی است که در میان تمام آیتم‌های آرایه حرکت کرده و هر آیتم را فراخوانی می‌کند. واقعا این الگوریتم بسیار ساده است. و اگرچه این الگوریتم ساده برای تعداد زیادی از سناریوها خوب است، اما محدودیت‌های بسیار زیادی دارد. برای نمونه، مقادیر برگشتی متدهای کالیک به جز آخرین آن‌ها، تماما از بین می‌روند.

با استفاده از این الگوریتم ساده، راهی وجود ندارد که مقادیر برگشتی تمام متدهای کالیک که فراخوانی می‌شوند را بدست آورد. اما این تنها محدودیت آن نیست. اگر یکی از نماینده‌های فراخوانی (درخواست) شده یک اکسپشن تولید کند یا برای مدت خیلی طولانی مسدود شود، چه رخ می‌دهد؟ چون الگوریتم، هر نماینده در زنجیر را به صورت سریالی و پشت سر هم فراخوانی می‌کند، یک "مشکل" با یکی از اشیاء نماینده مانع از فراخوانی تمام نماینده‌های بعد از آن در زنجیر می‌شود. روشن است که این الگوریتم قوی نیست.

برای سناریوهایی که در آن، این الگوریتم کافی نیست، کلاس **MulticastDelegate** یک متد نمونه، **GetInvocationList**، ارائه می‌کند که شما برای فراخوانی صریح هر نماینده در زنجیر با هر الگوریتمی که نیاز شما را برطرف کند، می‌توانید از آن استفاده کنید:

```

public abstract class MulticastDelegate : Delegate {
    // Creates a delegate array where each element refers
    // to a delegate in the chain.
    public sealed override Delegate[] GetInvocationList();
}

```

متد **GetInvocationList** روی یک شی متشقق شده از **MulticastDelegate** عمل کرده و یک آرایه از ارجاع‌های **Delegate** برمی‌گرداند که هر ارجاع به یکی از اشیاء نماینده در زنجیر اشاره می‌کند. در درون، **GetInvocationList** یک آرایه ساخته و هر عنصر آن را برای ارجاع به یک نماینده در زنجیر، مقداردهی اولیه می‌کند، سپس یک ارجاع به آرایه برگردانده می‌شود. اگر فیلد **_invocationList**، **null** باشد، آرایه برگشتی حاوی یک عنصر است که به تنها نماینده در زنجیر، خود نمونه نماینده، اشاره می‌کند.

شما می‌توانید براحتی الگوریتمی بنویسید که صریحا هر شی موجود در آرایه را فراخوانی کند. کد زیر این را نشان می‌دهد:

```

using System;
using System.Text;

```

```

// Define a Light component.
internal sealed class Light {
    // This method returns the light's status.
    public String SwitchPosition() {
        return "The light is off";
    }
}

// Define a Fan component.
internal sealed class Fan {
    // This method returns the fan's status.
    public String Speed() {
        throw new InvalidOperationException("The fan broke due to overheating");
    }
}

// Define a Speaker component.
internal sealed class Speaker {
    // This method returns the speaker's status.
    public String volume() {
        return "The volume is loud";
    }
}

public sealed class Program {
    // Definition of delegate that allows querying a component's status.
    private delegate String GetStatus();

    public static void Main() {
        // Declare an empty delegate chain.
        GetStatus getStatus = null;

        // Construct the three components, and add their status methods
        // to the delegate chain.
        getStatus += new GetStatus(new Light().SwitchPosition);
        getStatus += new GetStatus(new Fan().Speed);
        getStatus += new GetStatus(new Speaker().Volume);

        // Show consolidated status report reflecting
        // the condition of the three components.
        Console.WriteLine(GetComponentStatusReport(getStatus));
    }

    // Method that queries several components and returns a status report
    private static String GetComponentStatusReport(GetStatus status) {
        // If the chain is empty, there is nothing to do.
        if (status == null) return null;
    }
}

```

```

// Use this to build the status report.
StringBuilder report = new StringBuilder();

// Get an array where each element is a delegate from the chain.
Delegate[] arrayOfDelegates = status.GetInvocationList();

// Iterate over each delegate in the array.
foreach (GetStatus getStatus in arrayOfDelegates) {

    try {
        // Get a component's status string, and append it to the report.
        report.AppendFormat("{0}{1}{1}", getStatus(), Environment.NewLine);
    }
    catch (InvalidOperationException e) {
        // Generate an error entry in the report for this component.
        Object component = getStatus.Target;
        report.AppendFormat(
            "Failed to get status from {1}{2}{0} Error: {3}{0}{0}",
            Environment.NewLine,
            ((component == null) ? "" : component.GetType() + "."),
            getStatus.Method.Name,
            e.Message);
    }
}

// Return the consolidated report to the caller.
return report.ToString();
}
}

```

وقتی شما این کد را بسازید و اجرا کنید، خروجی زیر ظاهر می‌شود:

```

The light is off
Failed to get status from Fan.Speed
    Error: The fan broke due to overheating
The volume is loud

```

قبلا به اندازه کافی نماینده‌ها را معرفی کرده ایم (نماینده‌های جنریک)

سال‌ها پیش، وقتی دانت فریمورک تازه می‌خواست ساخته شود، مایکروسافت مفهوم نماینده‌ها را معرفی کرد. به تدریج که برنامه‌نویسان به FCL، کلاس اضافه می‌کردند، آن‌ها نوع‌های جدید نماینده را هر جایی که یک متد کالیک معرفی می‌کردند، تعریف می‌نمودند. در طول زمان، نماینده‌های بسیار زیادی تعریف شدند. در واقع، تنها در MSCorLib.dll اکنون نزدیک به ۵۰ نوع نماینده تعریف شده است. بگذارید به تعدادی از آن‌ها نگاهی بیاندازیم:

```

public delegate void TryCode(Object userData);
public delegate void waitCallback(Object state);
public delegate void TimerCallback(Object state);
public delegate void ContextCallback(Object state);
public delegate void sendOrPostCallback(Object state);
public delegate void ParameterizedThreadStart(Object obj);

```

آیا شما در این تعداد کم از تعاریف نماینده‌ها که انتخاب کرده‌ام، چیز مشابهی می‌بینید؟ در واقع، تمام آن‌ها یکسان هستند: یک متغیر از هر کدام از این نوع-های نماینده باید به یک متد که یک **Object** گرفته و **void** برمی‌گرداند اشاره کند. واقعا دلیلی وجود ندارد که تمام این نوع‌های نماینده را تعریف کنیم، فقط به یکی نیاز است.

در واقع، حال که دات‌نت فریمورک از جنریک‌ها پشتیبانی می‌کند، ما به تعداد کمی از نماینده‌های جنریک (تعریف شده در فضای نام **System**) نیاز داریم که متدهایی که تا ۱۶ آرگومان می‌گیرند را نمایش می‌دهند:

```
public delegate void Action(); // OK, this one is not generic
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);
```

پس دات‌نت فریمورک اکنون با ۱۷ نماینده **Action** ارائه می‌شود که از صفر آرگومان تا ۱۶ آرگومان را شامل می‌شوند. اگر شما نیاز داشته باشید متدی را فراخوانی کنید که بیش از ۱۶ آرگومان داشته باشد، شما مجبورید نوع نماینده خود را تعریف کنید، اما این خیلی بعید است.

علاوه بر نماینده‌های **Action**، دات‌نت فریمورک با ۱۷ نماینده **Func** عرضه می‌شود که اجازه می‌دهد متد کالیک یک مقدار برگرداند:

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate TResult Func<T1, ..., T16, TResult>(T1 arg1, ..., T16 arg16);
```

اکنون توصیه می‌شود هر جا که ممکن است به جای آنکه برنامه‌نویسان نوع‌های نماینده‌ی بیشتری تعریف کنند، از این نوع‌های نماینده استفاده کنند. این کار تعداد نوع‌های موجود در سیستم را کاهش داده و کدنویسی را ساده می‌کند. هر چند، اگر شما نیاز دارید یک آرگومان را با ارجاع با استفاده از کلمه کلیدی **ref** یا **out** ارسال کنید، نیاز خواهید داشت نماینده خودتان را تعریف کنید:

```
delegate void Bar(ref Int32 z);
```

شاید هم شما مجبور باشید این کار را بکنید اگر بخواهید نوع شی، یک تعداد متغیر از آرگومان‌ها را از طریق کلمه کلیدی **params** سی‌شارپ دریافت کند، اگر شما بخواهید مقادیر پیش فرض برای هر یک از آرگومان‌های نماینده‌تان را تعیین کنید، یا اگر شما نیاز داشته باشید آرگومان نوع جنریک یک نماینده که همانگونه که در کد زیر نشان داده شده است را محدود کنید:

```
delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e)
where TEventArgs : EventArgs;
```

نکته نوع نماینده **Action** و **Func** که 0 تا 8 آرگومان می‌گیرند در **mscorlib.dll** تعریف شده‌اند چون متدهایی که این تعداد آرگومان می‌گیرند بسیار رایجند. هر چند، نوع‌های نماینده **Action** و **Func** که ۹ تا ۱۶ آرگومان می‌گیرند در **System.Core.dll** تعریف شده‌اند، چرا که متدهایی که این تعداد آرگومان می‌گیرند نادر هستند. و در واقع، این تعاریف نماینده‌ها اغلب به صورت داخلی توسط زبان‌های برنامه‌نویسی پویا استفاده می‌شوند و معمولا توسط برنامه‌نویسان به صورت مستقیم استفاده نمی‌شوند.

هنگام استفاده از نماینده‌هایی که آرگومان و نوع برگشتی جنریک می‌گیرند، کنترا-واریانس و کواریانس وارد صحنه می‌شوند و توصیه می‌شود که شما همیشه از این ویژگی‌ها استفاده کنید چرا که اثر بدی نداشته و نماینده‌های شما را قادر می‌سازد تا در سناریوهای بیشتری استفاده شوند. برای اطلاعات بیشتر در این باره، بخش "آرگومان‌های نوع جنریک **Contravariant** و **Covariant** از نماینده و رابط" در فصل ۱۲، "جنریک‌ها" را ببینید.

شکر نحوی سی‌شارپ برای نماینده‌ها

بسیاری از برنامه‌نویسان کار با نماینده‌ها را سخت و دشوار می‌دانند چرا که نحو آن بسیار نا آشناست. برای نمونه، خط زیر از کدی را در نظر بگیرید:

```
button1.Click += new EventHandler(button1_Click);
```

که **button1_Click** متدی شبیه به این است:

```
void button1_Click(Object sender, EventArgs e) {
    // Do something, the button was clicked...
}
```

ایده پشت اولین خط از کد اینست که آدرس متد **button1_Click** را با یک کنترل دکمه ثبت کند تا وقتی دکمه کلیک می‌شود، متد فراخوانی گردد. برای اکثر برنامه‌نویسان، کاملاً غیرطبیعی به نظر می‌رسد که یک شی نماینده **EventHandler** فقط برای تعیین آدرس متد **button1_Click**، درست کنند. هر چند، ساخت یک شی نماینده **EventHandler** برای CLR نیاز است چون این شی یک پوشاننده فراهم می‌کند که این اطمینان را حاصل می‌کند که متد فقط به روشی نوع-امن می‌تواند فراخوانی شود. پوشاننده همچنین اجازه‌ی فراخوانی متدهای نمونه و زنجیربندی را می‌دهد. متأسفانه، اغلب برنامه‌نویسان درباره‌ی جزییات، تفکر نمی‌کنند. برنامه‌نویسان ترجیح می‌دهند کد فوق را اینگونه بنویسند:

```
button1.Click += button1_Click;
```

خوشبختانه، کامپایلر سی‌شارپ مایکروسافت، برای برنامه‌نویسان تعدادی میانبر نحوی هنگام کار با نماینده‌ها ارائه می‌کند. من این میانبر را در این بخش توضیح خواهم داد. یک نکته آخر قبل از آنکه شروع کنم: آنچه می‌خواهم توضیح دهم واقعا مربوط به شکر نحوی^{۵۶} سی‌شارپ است، این میانبرهای نحوی جدید فقط به برنامه‌نویسان راهی آسانتر برای تولید کد IL که باید تولید شود، می‌دهند تا CLR و دیگر زبان‌های برنامه‌نویسی بتوانند با نماینده‌ها کار کنند. این همچنین یعنی آنچه می‌خواهم بگویم مخصوص سی‌شارپ است، دیگر کامپایلرها شاید نحو اضافی برای نماینده‌ها ارائه نکنند.

میانبر نحوی شماره ۱: نیاز به ساخت یک شی نماینده نیست

همانگونه که قبلاً نشان داده شد، سی‌شارپ به شما اجازه می‌دهد نام یک متد کالیک را بدون اینکه مجبور باشید یک شی نماینده‌ی پوشاننده بسازید، تعیین کنید. مثالی دیگر را ببینید:

```
internal sealed class AClass {
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(SomeAsyncTask, 5);
    }

    private static void SomeAsyncTask(Object o) {
        Console.WriteLine(o);
    }
}
```

در اینجا، متد استاتیک **QueueUserWorkItem** از کلاس **ThreadPool** انتظار یک ارجاع به یک شی نماینده **WaitCallback** را دارد که حاوی یک ارجاع به متد **SomeAsyncTask** باشد. چون کامپایلر سی‌شارپ قادر است خودش این را استنتاج کند، به من اجازه می‌دهد کدی که شی نماینده **WaitCallback** را می‌سازد، حذف کنم، و کد را خواناتر و قابل درک تر بکنم. البته، وقتی کد کامپایل می‌شود، کامپایلر سی‌شارپ کد IL ای را تولید می‌کند که در حقیقت، شی نماینده **WaitCallback** را می‌سازد - ما فقط یک میانبر نحوی بدست آورده ایم.

میانبر نحوی شماره ۲: نیاز به تعریف یک متد کالیک نیست

در کد فوق، نام متد کالیک، **SomeAsyncTask** به متد **QueueUserWorkItem** از **ThreadPool** ارسال گردیده است. سی‌شارپ اجازه می‌دهد شما کد متد کالیک را خطی بنویسید تا مجبور نباشید آن را در متد خودش بنویسید. برای نمونه، کد فوق می‌تواند اینگونه نوشته شود:

```
internal sealed class AClass {
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem( obj => Console.WriteLine(obj), 5);
    }
}
```

^{۵۶} Syntactical Sugar: به نحوی گفته می‌شود که برنامه‌نویسی را آسانتر کرده و زبان را "شیرین تر" می‌کند.

توجه کنید اولین "آرگومان" به متد `QueueUserWorkItem`، کد است (که به صورت ایتالیک آمده است). به صورت رسمی تر، کد ایتالیک در اینجا یک عبارت لامبدا `lambda expression` سی شارپ نامیده می شود و شناسایی آن به خاطر عملگر عبارت لامبدا `سی شارپ` (`=>`) آسان است. شما می توانید در هر جایی از کدتان که کامپایلر به صورت عادی انتظار دیدن یک نماینده را دارد، از عبارت لامبدا استفاده کنید. و وقتی کامپایلر استفاده از این عبارت لامبدا را می بیند، کامپایلر به صورت خودکار یک متد خصوصی جدید در کلاس (`Aclass` در این مثال) تولید می کند. این متد جدید یک تابع ناشناس `anonymous function` نامیده می شود چون کامپایلر نام متد را به صورت خودکار برای شما درست می کند و به صورت عادی شما نام آن را نخواهید دانست. هر چند، می توانید از ابزاری مثل `ILDasm.exe` برای بررسی کد تولیدی توسط کامپایلر استفاده کنید. بعد از آنکه من کد فوق را نوشته و آن را کامپایل کردم، من با استفاده از `ILDasm.exe` قادرم ببینم که کامپایلر سی شارپ تصمیم گرفته نام این متد را `void b_0<CallbackWithoutNewingADelegateObject>` بگذارد و مطمئن می شود که این متد یک آرگومان `Object` گرفته و برمی گرداند.

کامپایلر تصمیم گرفت نام متد را با یک علامت < آغاز کند چون در سی شارپ، یک شناسه نمی تواند شامل علامت < باشد؛ این باعث می شود شما به صورت تصادفی متدی تعریف نکنید که با نام متدی که کامپایلر برای شما انتخاب کرده تداخل پیدا کند. تصادفاً، در حالیکه سی شارپ مانع از آن می شود که شناسه ها حاوی یک علامت < باشند، CLR اجازه ی آن را می دهد و به همین خاطر است که این کار را می کند. همچنین، توجه کنید در حالیکه شما می توانید به متد از طریق رفلکشن با ارسال نام متد به عنوان رشته، دسترسی پیدا کنید، مشخصات زبان سی شارپ بیان می کند که هیچ تضمینی وجود ندارد که کامپایلر چگونه نام را تولید می کند. برای نمونه، هر بار شما کد را کامپایل کنید، کامپایلر می تواند نام متفاوتی برای متد تولید کند.

با استفاده از `ILDasm.exe`، شاید شما متوجه شده باشید که کامپایلر سی شارپ، صفت `System.Runtime.CompilerServices.CompilerGeneratedAttribute` را به این متد اعمال می کند تا به ابزارها و برنامه های مختلف بیان کند که این متد توسط یک کامپایلر و نه یک برنامه نویس تولید شده است. کد سمت راست عملگر `=>` سپس در متد تولید شده توسط کامپایلر جای می گیرد.

نکته هنگام نوشتن یک عبارت لامبدا، راهی وجود ندارد که صفت دلخواه خود را به متد تولید شده توسط کامپایلر اعمال کنید. علاوه بر این، شما نمی توانید تغییر دهنده متد (مثل `unsafe`) را به متد اعمال کنید. اما این اغلب مشکل نیست چون متدهای ناشناس تولیدی توسط کامپایلر همیشه خصوصی بوده و متد بسته به اینکه به اعضای نمونه دسترسی دارد یا نه، استاتیک یا غیراستاتیک است. پس نیازی به اعمال تغییردهنده ای مثل `public`، `protected`، `internal`، `virtual`، `sealed` یا `abstract` بر متد نیست.

سرانجام، اگر شما کد نشان داده شده در فوق را بنویسید و آن را کامپایل کنید، کامپایلر کد شما را شبیه به این بازنویسی می کند (کامنت ها را من خودم گذاشته ام):

```
internal sealed class Aclass {
    // This private field is created to cache the delegate object.
    // Pro: CallbackWithoutNewingADelegateObject will not create
    //      a new object each time it is called.
    // Con: The cached object never gets garbage collected
    [CompilerGenerated]
    private static WaitCallback <>9__CachedAnonymousMethodDelegate1;

    public static void CallbackWithoutNewingADelegateObject() {
        if (<>9__CachedAnonymousMethodDelegate1 == null) {
            // First time called, create the delegate object and cache it.
            <>9__CachedAnonymousMethodDelegate1 =
                new waitCallback(<CallbackWithoutNewingADelegateObject>b_0);
        }
        ThreadPool.QueueUserWorkItem(<>9__CachedAnonymousMethodDelegate1, 5);
    }
}

[CompilerGenerated]
```



```
private static void <CallbackWithoutNewingADelegateObject>b__0(Object obj) {
    Console.WriteLine(obj);
}
}
```

عبارت لامبدا باید با نماینده **WaitCallback** مطابقت داشته باشد، **void** برگرداند و یک پارامتر **Object** بگیرد. هر چند، من نام پارامتر را با گذاشتن **obj** در سمت چپ عملگر => تعیین کرده‌ام. در سمت راست عملگر =>، **Console.WriteLine**، **void** برمی‌گرداند. اما اگر من عبارتی قرار می‌دادم که **void** بر نمی‌گرداند، کد تولیدی توسط کامپایلر، مقدار برگشتی را نادیده می‌گرفت چون متدی که کامپایلر تولید می‌کند باید نوع برگشتی **void** داشته باشد تا با نماینده **WaitCallback** مطابق باشد.

همچنین گفتن این نکته با ارزش است که تابع ناشناس با **private** علامت زده شده است؛ این مانع از آن می‌شود که هر کدی که درون نوع تعریف نشده باشد، به متد دسترسی پیدا کند (اگرچه رفلکشن نشان می‌دهد که متد وجود دارد). همچنین، توجه کنید که متد ناشناس با **static** علامت زده شده، این بدین خاطر است که به هیچ عضو نمونه‌ای دسترسی پیدا نمی‌کند (که آن نمی‌تواند این کار را بکند چون **CallbackWithoutNewingADelegateObject** خودش یک متد استاتیک است). هر چند، کد می‌تواند به هر فیلد استاتیک یا متد استاتیک تعریف شده درون کلاس ارجاع کند. یک مثال در اینجا آمده است:

```
internal sealed class AClass {
    private static string sm_name; // A static field
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(
            // The callback code can reference static members.
            obj => Console.WriteLine(sm_name+ ": " + obj),
            5);
    }
}
```

اگر متد **CallbackWithoutNewingADelegateObject** استاتیک نبود، کد متد ناشناس می‌توانست حاوی ارجاع‌هایی به اعضای نمونه باشد. اگر آن حاوی ارجاع‌هایی به اعضای نمونه نبود، کامپایلر هنوز هم یک متد ناشناس استاتیک تولید می‌کرد چرا که به خاطر عدم وجود پارامتر **this**، کاراتر می‌بود. اما اگر کد متد ناشناس یک عضو نمونه را ارجاع می‌کرد، کامپایلر یک متد ناشناس غیراستاتیک تولید خواهد کرد:

```
internal sealed class AClass {
    private string m_name; // An instance field
    // An instance method
    public void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(
            // The callback code can reference instance members.
            obj => Console.WriteLine(m_name+ ": " + obj),
            5);
    }
}
```

در سمت چپ عملگر => جایی است که شما نام‌های آرگومان‌هایی که به عبارت لامبدا ارسال می‌شوند را تعیین می‌کنید. قواعدی در اینجا وجود دارد که شما باید آن‌ها را رعایت کنید. مثال‌های زیر را ببینید:

```
// If the delegate takes no arguments, use ()
Func<String> f = () => "Jeff";

// If the delegate takes 1+ arguments, you can explicitly specify the types
Func<Int32, String> f2 = (Int32 n) => n.ToString();
Func<Int32, Int32, String> f3 = (Int32 n1, Int32 n2) => (n1 + n2).ToString();

// If the delegate takes 1+ arguments, the compiler can infer the types
```

```
Func<Int32, String> f4 = (n) => n.ToString();
Func<Int32, Int32, String> f5 = (n1, n2) => (n1 + n2).ToString();
```

```
// If the delegate takes 1 argument, you can omit the ()s
Func<Int32, String> f6 = n => n.ToString();
```

```
// If the delegate has ref/out arguments, you must explicitly specify ref/out and the type
Bar b = (out Int32 n) => n = 5;
```

برای آخرین مثال فرض کنید که Bar اینگونه تعریف شده است:

```
delegate void Bar(out Int32 z);
```

در سمت راست عملگر => جایی است که شما بدنه تابع ناشناس را تعیین می‌کنید. خیلی رایج است که بدنه شامل یک عبارت ساده یا پیچیده باشد که سرانجام یک مقدار غیر void برگرداند. در کد فوق، من عبارت‌های لامبدا که String برمی‌گرداند را به تمام متغیرهای نماینده Func اختصاص داده‌ام. همچنین بسیار رواج دارد که بدنه از یک تک عبارت تشکیل شود. یک مثال از این وقتی است که من ThreadPool.QueueUserWorkItem را فراخوانی کردم و یک عبارت لامبدا که Console.WriteLine (که void برمی‌گرداند) را بدان ارسال کردم. اگر شما بخواهید بدنه از دو یا بیشتر عبارت تشکیل شود، آنگاه شما باید آن را در آکولات قرار دهید و اگر نماینده انتظار یک مقدار برگشتی را دارد، آنگاه شما باید یک عبارت return درون بدنه داشته باشید. مثالی ببینید:

```
Func<Int32, Int32, String> f7 = (n1, n2) => { Int32 sum = n1 + n2; return sum.ToString();
};
```

مهم اگر واضح نیست بگذارید صریحا اشاره کنم فایده‌ی اصلی عبارت‌های لامبدا اینست که آن‌ها یک سطح از مسیر غیر مستقیم را از کد شما حذف می‌کنند. به صورت عادی، شما مجبورید یک متد مجزا بنویسید، یک نام به متد بدهید و سپس نام متد را در جاییکه یک نماینده نیاز است، ارسال کنید. نام به شما راهی برای اشاره به یک بدنه می‌دهد و اگر شما نیاز دارید به همان بدنه از کد در چند جای سورس کدتان اشاره کنید، آنگاه نوشتن یک متد و دادن یک نام به آن راه صحیحی است که باید طی کنید. هر چند، اگر شما نیاز دارید یک بدنه از کد داشته باشید که فقط یکبار در سورس کدتان ارجاع می‌شود، آنگاه یک عبارت لامبدا به شما اجازه می‌دهد آن کد را مستقیما به صورت خطی بدون تخصیص یک نام، قرار دهید که بهره‌وری برنامه‌نویس را بالا می‌برد.

نکته وقتی سی‌شارپ 2.0 عرضه شد یک ویژگی به نام متدهای ناشناس anonymous methods را معرفی کرد. همانند عبارت‌های لامبدا (که در سی‌شارپ 3.0 معرفی شدند)، متدهای ناشناس یک نحو برای ساخت توابع ناشناس فراهم می‌کنند. (در بخش ۷.۱۴ از مشخصات زبان سی-شارپ) توصیه می‌شود که برنامه‌نویسان به جای نحو متدهای ناشناس قدیمی از نحو عبارت‌های لامبدا جدید استفاده کنند. نحو عبارت لامبدا خلاصه تر است و نوشتن، خواندن و نگهداری کد را آسانتر می‌کند. البته، کامپایلر سی‌شارپ مایکروسافت از هر دو نحو برای ساخت توابع ناشناس پشتیبانی می‌کند تا برنامه‌نویسان مجبور نباشند هر کدی که برای سی‌شارپ 2.0 نوشته‌اند را تغییر دهند. در این کتاب، من تنها نحو عبارت لامبدا را توضیح داده و از آن استفاده می‌کنم.

میانبر نحوی شماره ۳: نیاز نیست متغیرهای محلی در یک کلاس را جهت ارسال به یک متد کالک، به صورت دستی بیوشانید

من قبلا نشان دادم چگونه کد کالک می‌تواند دیگر اعضای تعریف شده در کلاس را ارجاع کند. هر چند، گاهی، شاید شما بخواهید کد کالک به متغیرهای محلی یا متغیرهایی که در متد تعریف کننده وجود دارند، ارجاع کند. یک مثال جالب را ببینید:

```
internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {
```

```

// Some local variables
Int32[] squares = new Int32[numToDo];
AutoResetEvent done = new AutoResetEvent(false);

// Do a bunch of tasks on other threads
for (Int32 n = 0; n < squares.Length; n++) {
    ThreadPool.QueueUserWorkItem(
        obj => {
            Int32 num = (Int32) obj;

            // This task would normally be more time consuming
            squares[num] = num * num;

            // If last task, let main thread continue running
            if (Interlocked.Decrement(ref numToDo) == 0)
                done.Set();
        },
        n);
}

// wait for all the other threads to finish
done.WaitOne();

// Show the results
for (Int32 n = 0; n < squares.Length; n++)
    Console.WriteLine("Index {0}, Square={1}", n, squares[n]);
}
}

```

این مثال نشان می‌دهد چگونه سی‌شارپ پیاده‌سازی یک عملیات بسیار پیچیده را آسان می‌کند. متد فوق یک پارامتر، **numToDo**، و دو متغیر محلی **square** و **done** تعریف می‌کند و بدنه عبارت لامبدا به این متغیرها اشاره می‌کند. حال تصور کنید که درون بدنه‌ی عبارت لامبدا در متد مجزایی قرار دارد (آنچنان که CLR بدان نیاز دارد). چگونه مقادیر متغیرها به متد مجزا ارسال می‌شود؟ تنها راه انجام چنین کاری تعریف یک کلاس کمکی جدید است که به ازای هر مقداری که می‌خواهید به کد کالیک ارسال کنید، یک فیلد تعریف می‌کند. به علاوه، کد کالیک مجبور خواهد بود به عنوان یک متد نمونه در این کلاس کمکی تعریف شود. آنگاه، متد **UsingLocalVariablesInTheCallbackCode** مجبور است یک نمونه از کلاس کمکی ساخته، فیلدها را از روی مقادیر متغیرهای محلی‌اش مقداردهی اولیه کرده و آنگاه شی نمایندده را به شی کمکی/متد نمونه متصل کند.

نکته وقتی یک عبارت لامبدا باعث می‌شود کامپایلر یک کلاس با پارامتر/متغیرهای محلی که به فیلد تبدیل شده‌اند، تعریف کند، دروه‌ی حیات اشیایی که متغیرها به آن‌ها اشاره می‌کنند طولانی‌تر می‌شود. معمولاً، یک پارامتر/متغیر محلی در آخرین استفاده از متغیر درون یک متد، از میدان خارج می‌شود. هر چند، تبدیل متغیر به یک فیلد باعث می‌شود فیلد، شی را در تمام دروه حیات شی حاوی فیلد، زنده نگه دارد. این مشکل بزرگی در اغلب برنامه‌ها نیست، اما چیزی است که شما باید از آن مطلع باشید.

این کار بسیار خسته‌کننده و همراه با خطای احتمالی است و البته کامپایلر سی‌شارپ، تمام این را به صورت خودکار برای شما انجام می‌دهد. وقتی شما کد نشان داده شده در بالا را می‌نویسید، کامپایلر سی‌شارپ کد شما را شبیه به این، باز می‌نویسد (کامنت‌ها را من گذاشته‌ام):

```

internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {

```

```

// Some local variables
waitCallback callback1 = null;

// Construct an instance of the helper class
<>c__DisplayClass2 class1 = new <>c__DisplayClass2();

// Initialize the helper class's fields
class1.numToDo = numToDo;
class1.squares = new Int32[class1.numToDo];
class1.done = new AutoResetEvent(false);

// Do a bunch of tasks on other threads
for (Int32 n = 0; n < class1.squares.Length; n++) {
    if (callback1 == null) {
        // New up delegate object bound to the helper object and
        // its anonymous instance method
        callback1 = new waitCallback(
            class1.<UsingLocalVariablesInTheCallbackCode>b__0);
    }

    ThreadPool.QueueUserWorkItem(callback1, n);
}

// wait for all the other threads to finish
class1.done.WaitOne();

// Show the results
for (Int32 n = 0; n < class1.squares.Length; n++)
    Console.WriteLine("Index {0}, Square={1}", n, class1.squares[n]);
}

// The helper class is given a strange name to avoid potential
// conflicts and is private to forbid access from outside AClass
[CompilerGenerated]
private sealed class <>c__DisplayClass2 : Object {

    // One public field per local variable used in the callback code
    public Int32[] squares;
    public Int32 numToDo;
    public AutoResetEvent done;

    // public parameterless constructor
    public <>c__DisplayClass2 { }

    // Public instance method containing the callback code
    public void <UsingLocalVariablesInTheCallbackCode>b__0(Object obj) {
        Int32 num = (Int32) obj;
        squares[num] = num * num;
    }
}

```

```

        if (Interlocked.Decrement(ref numToDo) == 0)
            done.Set();
    }
}
}

```

مهم بدون شک زمان زیادی طور نمی‌کشد که برنامه‌نویسان شروع به استفاده ناصحیح از ویژگی عبارت لامبدای سی‌شارپ می‌کنند. وقتی من برای اولین بار شروع به استفاده از عبارت‌های لامبدا کردم، برایم زمان برد تا به آن‌ها عادت کردم. گذشته از این، کدی که شما در یک متد می‌نویسید واقعا درون آن متد نیست و این خطایابی و حرکت تک‌قدمی در کد را کمی به چالش می‌کشد. در حقیقت، من شگفت‌زده‌ام که چقدر خوب دیباگر ویژوال استودیوی مایکروسافت، حرکت بین عبارت‌های لامبدا در سورس‌کد را مدیریت می‌کند.

من یک قاعده برای خودم درست کرده‌ام: اگر نیاز دارم که متد کالیک من بیش از سه خط کد داشته باشد، من از عبارت لامبدا استفاده نمی‌کنم، به جای آن من به صورت دستی متد را نوشته و یک نام از پیش خودم به آن اختصاص می‌دهم. اما اگر درست استفاده شوند، عبارت‌های لامبدا بهره‌وری برنامه‌نویس را افزایش داده و قابلیت نگهداری کد را نیز بالا می‌برند. در کد زیر استفاده از عبارت‌های لامبدا بسیار طبیعی است:

```

// Create an initialize a String array
String[] names = { "Jeff", "Kristin", "Aidan", "Grant" };

// Get just the names that have a lowercase 'a' in them.
Char charToFind = 'a';
names = Array.FindAll(names, name => name.IndexOf(charToFind) >= 0);

// Convert each string's characters to uppercase
names = Array.ConvertAll(names, name => name.ToUpper());

// Display the results
Array.ForEach(names, Console.WriteLine);

```

نماینده‌ها و رفلکشن

تاکنون در این فصل، استفاده از نماینده‌ها به این نیاز داشت که برنامه‌نویس فرم کلی‌متدی که باید کالیک شود را بداند. برای مثال، اگر **fb** یک متغیر به یک نماینده **Feedback** است (اولین برنامه این فصل را ببینید) برای فراخوانی (درخواست) نماینده، کد شبیه به این می‌شد:

```
fb(item); // item is defined as Int32
```

همانگونه که می‌بینید، هنگام کدنویسی، برنامه‌نویس باید بداند متد کالیک چه تعداد پارامتر نیاز داشته و نوع این پارامترها چیست. خوشبختانه، برنامه‌نویس اغلب همیشه این اطلاعات را دارد، پس نوشته کدی شبیه کد قبلی مسئله نیست.

هر چند در شرایط نادری، برنامه‌نویس این اطلاعات را در زمان کامپایل ندارد. من یک نمونه از این را در فصل ۱۱ "رویدادها" وقتی نوع **EventSet** را بحث می‌کردم نشان دادم. در این مثال، یک دیکشنری، یک مجموعه از نوع‌های نماینده مختلف را نگهداری می‌کرد. در زمان اجرا، برای فعال کردن یک رویداد، یکی از نماینده‌ها در دیکشنری جستجو شده و فراخوانی (درخواست) می‌شود. در زمان کامپایل، این امکان وجود نداشت که بدانیم دقیقا چه نماینده‌ای فراخوانی می‌شود و کدام پارامترها برای ارسال به متد کالیک نماینده نیاز است. خوشبختانه، **System.Delegate** چند متد ارائه می‌کند که به شما اجازه می‌دهند وقتی تمام اطلاعات لازم درباره‌ی نماینده‌ها را در زمان کامپایل ندارید، یک نماینده ساخته و آن را فراخوانی کنید. متدهای متناظری که **Delegate** تعریف می‌کند:

```

public abstract class Delegate {
    // Construct a 'type' delegate wrapping the specified static method.
    public static Delegate CreateDelegate(Type type, MethodInfo method);
}

```

```

public static Delegate CreateDelegate(Type type, MethodInfo method,
    Boolean throwOnBindFailure);

// Construct a 'type' delegate wrapping the specified instance method.
public static Delegate CreateDelegate(Type type,
    Object firstArgument, MethodInfo method); // firstArgument means 'this'
public static Delegate CreateDelegate(Type type,
    Object firstArgument, MethodInfo method, Boolean throwOnBindFailure);

// Invoke a delegate passing it parameters
public Object DynamicInvoke(params Object[] args);
}

```

تمام متدهای **CreateDelegate** یک شی جدید از یک نوع مشتق شده از **Delegate** که توسط اولین پارامتر، **type**، شناسایی می‌شود، می‌سازند. پارامتر **MethodInfo**، متدی که باید کالک شود را تعیین می‌کند، شما از API های رفلکشن (بحث شده در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن") برای بدست آوردن این مقدار استفاده می‌کنید. اگر شما می‌خواهید نماینده یک متد نمونه را بیوشاند، شما به **CreateDelegate** یک پارامتر **firstArgument** نیز ارسال می‌کنید که شی‌ای که باید به عنوان پارامتر **this** (اولین آرگومان) به متد نمونه ارسال شود را تعیین می‌کند. سرانجام، **CreateDelegate** به صورت عادی اگر نماینده نتواند به متد تعیین شده توسط پارامتر **method** متصل شود، یک **ArgumentException** تولید می‌کند. این می‌تواند رخ دهد اگر امضای متدی که توسط **method** شناسایی می‌شود با امضایی که توسط نماینده که با پارامتر **type** شناسایی می‌شود، تطابق نداشته باشد. هر چند، اگر شما **false** را برای پارامتر **throwOnBindFailure** ارسال کنید، یک **ArgumentException** تولید نخواهد شد، به جای آن، **null** برگشت داده خواهد شد.

مهم کلاس **System.Delegate** تعداد زیادی سربارگذاری از متد **CreateDelegate** دارد که من در اینجا نشان نمی‌دهم. شما نباید هرگز هیچ کدام از این متدها را فراخوانی کنید. در حقیقت، مایکروسافت حتی از تعریف آن‌ها تاسف می‌خورد. علت اینست که این دیگر متدها، متدی که باید متصل شود را به جای یک **MethodInfo** با یک **String** شناسایی می‌کنند. این یعنی یک اتصال مهم امکان دارد باعث شود برنامه شما به صورت غیر قابل پیش‌بینی رفتار کند.

متد **DynamicInvoke** از **System.Delegate** به شما اجازه می‌دهد متد کالک از یک شی نماینده را فراخوانی کرده، یک مجموع از پارامترهای که شما در زمان اجرا تعیین می‌کنند بدان ارسال کنید. وقتی شما **DynamicInvoke** را فراخوانی می‌کنید، در درون آن مطمئن می‌شود پارامترهایی که شما ارسال می‌کنید با پارامترهایی که متد کالک انتظار دارد سازگار باشند. اگر سازگار هستند متد کالک فراخوانی می‌شود. اگر نیستند، یک **ArgumentException** تولید می‌شود. **DynamicInvoke** شی‌ای که متد کالک برمی‌گرداند را برمی‌گرداند. کد زیر نشان می‌دهد چگونه از متدهای **CreateDelegate** و **DynamicInvoke** باید استفاده کرد:

```

using System;
using System.Reflection;
using System.IO;

// Here are some different delegate definitions
internal delegate Object TwoInt32s(Int32 n1, Int32 n2);
internal delegate Object OneString(String s1);

public static class Program {
    public static void Main(String[] args) {
        if (args.Length < 2) {
            String fileName = Path.GetFileNameWithoutExtension(
                Assembly.GetEntryAssembly().Location);

```

```

String usage =
    @"Usage: " +
    "{0}{1} delType methodName [Arg1] [Arg2]" +
    "{0} where delType must be TwoInt32s or OneString"+
    "{0} if delType is TwoInt32s, methodName must be Add or Subtract" +
    "{0} if delType is OneString, methodName must be NumChars or Reverse" +
    "{0}" +
    "{0}Examples:" +
    "{0} {1} TwoInt32s Add 123 321" +
    "{0} {1} TwoInt32s Subtract 123 321" +
    "{0} {1} OneString NumChars \"Hello there\"" +
    "{0} {1} OneString Reverse \"Hello there\"";
Console.WriteLine(usage, Environment.NewLine, fileName);
return;
}

// Convert the delType argument to a delegate type
Type delType = Type.GetType(args[0]);
if (delType == null) {
    Console.WriteLine("Invalid delType argument: " + args[0]);
    return;
}

Delegate d;
try {
    // Convert the Arg1 argument to a method
    MethodInfo mi = typeof(Program).GetMethod(args[1],
        BindingFlags.NonPublic | BindingFlags.Static);

    // Create a delegate object that wraps the static method
    d = Delegate.CreateDelegate(delType, mi);
}
catch (ArgumentException) {
    Console.WriteLine("Invalid methodName argument: " + args[1]);
    return;
}

// Create an array that will contain just the arguments
// to pass to the method via the delegate object
Object[] callbackArgs = new Object[args.Length - 2];
if (d.GetType() == typeof(TwoInt32s)) {
    try {
        // Convert the String arguments to Int32 arguments
        for (Int32 a = 2; a < args.Length; a++)
            callbackArgs[a - 2] = Int32.Parse(args[a]);
    }
    catch (FormatException) {
        Console.WriteLine("Parameters must be integers.");
    }
}

```

```
        return;
    }
}

if (d.GetType() == typeof(OneString)) {
    // Just copy the String argument
    Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
}

try {
    // Invoke the delegate and show the result
    Object result = d.DynamicInvoke(callbackArgs);
    Console.WriteLine("Result = " + result);
}
catch (TargetParameterCountException) {
    Console.WriteLine("Incorrect number of parameters specified.");
}
}

// This callback method takes 2 Int32 arguments
private static Object Add(Int32 n1, Int32 n2) {
    return n1 + n2;
}

// This callback method takes 2 Int32 arguments
private static Object Subtract(Int32 n1, Int32 n2) {
    return n1 - n2;
}

// This callback method takes 1 String argument
private static Object NumChars(String s1) {
    return s1.Length;
}

// This callback method takes 1 String argument
private static Object Reverse(String s1) {
    Char[] chars = s1.ToCharArray();
    Array.Reverse(chars);
    return new String(chars);
}
}
```


فصل ۱۸: صفت های سفارشی

در این فصل، من یکی از خلاقانه ترین ویژگی های دات نت فریمورک میکروسافت را بحث می کنم: صفت های سفارشی `custom attributes`. صفت های سفارشی به شما اجازه می دهند ساخت های کد خود را حاشیه نویسی کنید تا ویژگی های خاصی را فعال کنید. صفت های سفارشی اجازه می دهند تا اطلاعات روی تقریباً هر ورودی جدول متادیتا تعریف و اعمال شود. این اطلاعات متادیتای قابل توسعه در زمان اجرا، می تواند خوانده شود تا به صورت پویا اجرای کد را تغییر دهد.

وقتی از تکنولوژی های مختلف دات نت فریمورک (`XML Web Services`، `Web Forms`، `Windows Forms`) استفاده می کنید شما می بینید که همه ی آن ها از صفت های سفارشی بهره می برند تا به برنامه نویسان اجازه دهند اهداف خود را در کد به آسانی بیان کنند. یک درک عمیق و درست از صفت های سفارشی برای هر برنامه نویس دات نت فریمورک الزامی است.

استفاده از صفت های سفارشی

صفت ها، همانند `public`، `private`، `static` و غیره می توانند بر نوع ها و اعضا اعمال شوند. من فکر کنم همه ی ما در مفید بودن اعمال کردن صفت ها، موافقیم. اما آیا مفیدتر نیست اگر ما بتوانیم صفت های خودمان را تعریف کنیم؟ برای نمونه، اگر من بتوانم نوعی تعریف کنم و به طریقی بیان کنم که نوع می تواند از طریق سریالی سازی، از راه دور استفاده شود، چطور است؟ یا شاید من بتوانم یک صفت را به یک متد اعمال کنم تا نشان دهم اجازه ی امنیتی خاصی باید قبل از آنکه متد بتواند اجرا شود، تامین گردد.

البته، ساخت و اعمال صفت های تعریف شده توسط کاربر بر نوع ها و متدها، بسیار عالی و رایج است اما این نیازمند آن است که کامپایلر از این صفت ها آگاه بوده تا بتواند اطلاعات صفت را در متادیتای خروجی تولید کند. چون سازندگان کامپایلرها معمولاً ترجیح می دهند سورس کد کامپایلرهایشان را عرضه نکنند، میکروسافت راهی دیگر برای صفت های تعریف شده توسط کاربر ارائه کرد. این مکانیزم، که صفت های سفارشی نامیده می شود یک مکانیزم فوق العاده قدرتمند است که در هر دو زمان طراحی برنامه و زمان اجرا سودمند است. هر کسی می تواند صفت های سفارشی را تعریف کرده و استفاده کند، تمام کامپایلرهایی که با CLR کار می کنند باید به گونه ای طراحی شوند که صفت های سفارشی را تشخیص داده و آن ها را در متادیتای خروجی تولید کنند.

اولین چیزی که شما درباره ی صفت های سفارشی باید بدانید اینست که آن ها فقط راهی برای همراه کردن اطلاعات اضافی با یک هدف، هستند کامپایلر این اطلاعات اضافی را در متادیتای مازول مدیریت شده تولید می کند. اغلب صفت ها برای کامپایلر معنی ای ندارند؛ کامپایلر صفت ها را در سورس کد یافته و متادیتای متناظر را تولید می کند.

کتابخانه کلاس دات نت فریمورک (FCL) صدها صفت سفارشی تعریف می کند که می تواند بر آیتم های سورس کد شما اعمال شود. تعدادی مثال در اینجا آمده است:

- اعمال صفت `DllImport` بر یک متد، CLR را آگاه می کند که پیاده سازی متد در حقیقت در کد مدیریت نشده در درون DLL تعیین شده قرار دارد.
- اعمال صفت `Serializable` بر یک نوع، فرمت کننده های سریالی کردن را آگاه می کند که فیلدهای یک نمونه می توانند سریالی و غیرسریالی شوند.
- اعمال صفت `AssemblyVersion` بر یک اسمبلی شماره نسخه اسمبلی را تنظیم می کند.
- اعمال صفت `Flags` به یک نوع شمارشی باعث می شود نوع شمارشی به عنوان یک مجموعه از پرچم های بیتی عمل کند.

در زیر کد سی شارپی را می بینید که صفت های فراوانی بر آن اعمال شده است. در سی شارپ، شما با قراردادن یک صفت در براکت بلافاصله قبل از هدف، صفت سفارشی را بر هدف اعمال می کنید. مهم نیست درک کنید این کد چه کاری انجام می دهد. من فقط می خواهم شما ببینید صفت ها شبیه چیستند:

```
using System;
```

```
using System.Runtime.InteropServices;
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
```

```
internal sealed class OSVERSIONINFO {
```

```

public OSVERSIONINFO() {
    OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
}

public UInt32 OSVersionInfoSize = 0;
public UInt32 MajorVersion = 0;
public UInt32 MinorVersion = 0;
public UInt32 BuildNumber = 0;
public UInt32 PlatformId = 0;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
public string CSDVersion = null;
}

internal sealed class MyClass {
    [DllImport("kernel32", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO ver);
}

```

در این مورد، صفت **StructLayout** بر کلاس **OSVERSIONINFO**، صفت **MarshalAs** بر فیلد **CSDVersion** و صفت **DllImport** بر متد **GetVersionEx** اعمال شده‌اند. هر زبان برنامه‌نویسی، نحوی که کاربر برای اعمال یک صفت سفارشی بر یک هدف باید استفاده کند را تعریف می‌کند. برای نمونه ویژوال بیسیک دات‌نت مایکروسافت، به جای براکت به زاویه دار (علامت‌های کوچکتری و بزرگتری) (> و <) نیاز دارد.

CLR اجازه می‌دهد صفت‌ها تقریباً بر هر چیزی که در متادیتای یک فایل قابل بیان است، اعمال شوند. اغلب رایج است که صفت‌ها بر ورودی‌های جدول-های زیر اعمال شوند: **TypeDef** (کلاس‌ها، ساختارها، نوع‌های شمارشی، رابط‌ها و نماینده‌ها)، **MethodDef** (شامل سازنده‌ها)، **ParamDef**، **FieldDef**، **PropertyDef**، **EventDef**، **AssemblyDef** و **ModuleDef**. به خصوص، سی‌شارپ به شما اجازه می‌دهد یک صفت را فقط به سورس‌کدی اعمال کنید که یکی از هدف‌های زیر را تعریف می‌کند: اسمبلی، ماژول، نوع (کلاس، ساختار، شمارشی، رابط، نماینده)، فیلد، متد (شامل سازنده-ها)، پارامتر متد، مقدار برگشتی متد، ویژگی، رویداد و پارامتر نوع جنریک.

وقتی شما یک صفت اعمال می‌کنید، سی‌شارپ به شما اجازه می‌دهد یک پیشوند تعیین کنید که هدفی که صفت بر آن اعمال می‌شود را نشان دهد. کد زیر تمام پیشوندهای ممکن را نشان می‌دهد. در بسیاری موارد، اگر شما پیشوند را حذف کنید، کامپایلر هنوز هم می‌تواند هدفی که صفت بر آن اعمال می‌شود را تعیین کند، همانگونه که در مثال قبلی نشان داده شد. در برخی موارد، پیشوند باید تعیین گردد تا مقصود شما برای کامپایلر روشن شود. پیشوندهای نمایش داده شده در فرم ایتالیک در مثال زیر اجباری هستند:

```

using System;

[assembly: SomeAttr]           // Applied to assembly
[module: SomeAttr]           // Applied to module

[type: SomeAttr]             // Applied to type
internal sealed class SomeType<[typevar: SomeAttr] T> { // Applied to generic type variable

    [field: SomeAttr]         // Applied to field
    public Int32 SomeField = 0;

    [return: SomeAttr]       // Applied to return value
    [method: SomeAttr]       // Applied to method
    public Int32 SomeMethod(
        [param: SomeAttr]    // Applied to parameter

```

```

Int32 SomeParam) { return SomeParam; }

[property: SomeAttr] // Applied to property
public String SomeProp {
    [method: SomeAttr] // Applied to get accessor method
    get { return null; }
}

[event: SomeAttr] // Applied to event
[field: SomeAttr] // Applied to compiler-generated field
[method: SomeAttr] // Applied to compiler-generated add & remove methods
public event EventHandler SomeEvent;
}

```

حال که می‌دانید چگونه یک صفت دلخواه را اعمال کنید، بگذارید ببینیم یک صفت واقعا چیست. یک صفت سفارشی یک نمونه از یک نوع است. برای تطابق با مشخصات مشترک زبان (CLS)، کلاس‌های صفت سفارشی باید مستقیم یا غیر مستقیم از کلاس عمومی خلاصه **System.Attribute** مشتق شوند. سی‌شارپ تنها اجازه‌ی صفت‌های سازگار با CLS را می‌دهد. با بررسی مستندات SDK دات‌نت فریمورک شما خواهید دید که کلاس‌های زیر (از مثال قبلی) تعریف شده‌اند: **StructLayoutAttribute**، **MarshalAsAttribute**، **DllImportAttribute**، **InAttribute** و **OutAttribute**. تمام این کلاس‌ها در فضای نام **System.Runtime.InteropServices** تعریف شده‌اند اما کلاس‌های صفت می‌توانند در هر فضای نامی تعریف شوند. بعد از بررسی بیشتر، شما خواهید دید که تمام این کلاس‌ها از **System.Attribute** مشتق شده‌اند همانگونه که تمام کلاس‌های صفت که سازگار با CLS هستند، باید باشند.

نکته هنگام اعمال یک صفت بر یک هدف در سورس‌کد، کامپایلر سی‌شارپ اجازه می‌دهد پسوند **Attribute** را برای کاهش تایپ کردن و افزایش خوانایی سورس‌کد حذف کنید. مثال‌های من در این فصل از این ویژگی سی‌شارپ بهره می‌برند. برای نمونه، سورس‌کد من به جای حاوی **[DllImport(...)]** حاوی **[DllImport(...)]** است.

همانطور که قبلا اشاره کردم، یک صفت یک نمونه از یک کلاس است. کلاس باید یک سازنده عمومی داشته باشد تا نمونه‌هایی از آن بتواند ساخته شود. پس وقتی شما یک صفت را به یک هدف اعمال می‌کنید، نحو آن شبیه فراخوانی یکی از سازنده‌های نمونه کلاس است. به علاوه، یک زبان ممکن است نحو ویژه‌ای را اجازه دهد تا شما بتوانید هر یک از فیلد یا ویژگی‌های عمومی همراه با کلاس را تنظیم کنید. کاربردی از صفت **DllImport** که بر متد **GetVersionEx** اعمال شده بود را به خاطر آورید:

```
[DllImport("kernel32", CharSet = CharSet.Auto, SetLastError = true)]
```

نحو این خط باید خیلی برای شما عجیب باشد چون شما هرگز از این نحو برای فراخوانی یک سازنده استفاده نکرده‌اید. اگر شما کلاس **DllImportAttribute** را در مستندات بررسی کنید، خواهید دید که سازنده‌اش نیاز به یک پارامتر **String** دارد. در این مثال، **"Kernel32"** برای پارامتر ارسال شده است. پارامترهای یک سازنده، پارامترهای موضعی **positional parameters** نامیده شده و اجباری هستند. وقتی صفت اعمال می‌شود، پارامتر باید تعیین گردد.

دو "پارامتر" دیگر چه چیزی هستند؟ این نحو ویژه به شما اجازه می‌دهد هر یک از فیلدها یا ویژگی‌های عمومی شی **DllImportAttribute** را بعد از اینکه شی ساخته شد تنظیم کنید. در این مثال، شی **DllImportAttribute** ساخته شده و **"Kernel32"** به سازنده ارسال گردیده، فیلدهای نمونه عمومی شی، **CharSet** و **SetLastError** به ترتیب به **CharSet.Auto** و **true** تنظیم می‌شوند. پارامترهایی که فیلدها یا ویژگی‌ها را تنظیم می‌کنند پارامترهای اسمی **named parameter** نامیده شده و انتخابی هستند چون وقتی یک نمونه از صفت را اعمال می‌کنید مجبور نیستید این پارامترها را تعیین کنید. کمی بعدتر، من توضیح خواهم داد چه چیزی باعث می‌شود یک نمونه از کلاس **DllImportAttribute** ساخته شود.

همچنین توجه کنید که این امکان وجود دارد که چندین صفت را بر یک هدف اعمال کنید. برای نمونه، در اولین برنامه این فصل، پارامتر **ver** از متد **GetVersionEx** هر دو صفت **In** و **Out** بر آن اعمال شده است. وقتی چندین صفت را بر یک هدف اعمال می‌کنید، آگاه باشید که ترتیب صفت‌ها اهمیتی ندارد. همچنین در سی‌شارپ، هر صفت می‌تواند در براکت قرار بگیرد و یا چندین صفت می‌تواند درون یک براکت قرار گرفته و با کاما از هم جدا

شده باشند. اگر سازنده کلاس صفت، پارامتری نگیرد، پراترها انتخابی است. سرانجام، همانطور که قبلا اشاره شد، پسوند **Attribute** انتخابی است. خطوط زیر یکسان عمل کرده و تمام حالت‌های مختلف اعمال صفت‌ها را نشان می‌دهد:

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

تعریف کلاس صفت خودتان

شما می‌دانید یک صفت، یک نمونه از یک کلاس مشتق شده از **System.Attribute** است و همچنین شما می‌دانید چگونه یک صفت را اعمال کنید. حال بگذارید نگاهی به چگونگی تعریف کلاس‌های صفت‌های سفارشی خودتان ببینیم. بگوییم که شما کارمند مایکروسافت بوده و مسئول افزودن پشتیبانی پرچم بیتی برای نوع‌های شمارشی هستید. برای انجام این کار، اولین چیزی که شما باید انجام دهید تعریف کلاس **FlagsAttribute** است:

```
namespace system {
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
        }
    }
}
```

توجه کنید که کلاس **FlagsAttribute** از **Attribute** مشتق می‌شود، این چیزی است که کلاس **FlagsAttribute** را یک صفت سفارشی سازگار با **CLS** می‌کند. به علاوه، نام کلاس پسوندی از **Attribute** دارد؛ این استاندارد رایج را رعایت می‌کند اما آن اجباری نیست. سرانجام، تمام صفت‌های غیر-خالصه (**non-abstract**) باید حداقل یک سازنده عمومی داشته باشند. سازنده ساده **FlagsAttribute** پارامتری نمی‌گیرد و مطلقا کاری نمی‌کند.

مهم شما باید به یک صفت به عنوان یک نگهدارنده‌ی وضعیت منطقی نگاه کنید. یعنی، در حالیکه یک نوع صفت، یک کلاس است، کلاس باید ساده باشد. کلاس باید تنها یک سازنده عمومی ارائه کند که اطلاعات وضعیتی اجباری (موضعی) صفت را دریافت می‌کند و کلاس می‌تواند فیلدها/ویژگی‌های عمومی ارائه کند که اطلاعات وضعیتی انتخابی (یا اسمی) صفت را دریافت می‌کنند. کلاس نباید هیچ متد، رویداد یا دیگر اعضای عمومی را ارائه کند.

به طور کلی، من همیشه استفاده از فیلدهای عمومی را نپسندم و هنوز هم آن‌ها را برای صفت‌ها نپسندم. بسیار بهتر است از ویژگی استفاده کنید چرا که این اجازه‌ی انعطاف پذیری بیشتری می‌دهد اگر شما تصمیم بگیرید نحوه‌ی پیاده‌سازی کلاس را تغییر دهید.

تاکنون، نمونه‌های کلاس **FlagsAttribute** می‌توانند بر هر هدفی اعمال شوند اما این صفت در حقیقت فقط باید بر نوع‌های شمارشی اعمال شود. معنی ندارد که صفت را بر یک ویژگی یا یک متد اعمال کنید. برای آنکه به کامپایلر بگویید این صفت کجا مجاز است که اعمال شود، شما یک نمونه از کلاس **System.AttributeUsageAttribute** را به کلاس صفت اعمال می‌کنید. کد جدید این گونه است:

```
namespace system {
    [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
        }
    }
}
```

در این نسخه جدید، من یک نمونه از **AttributeUsageAttribute** را بر صفت اعمال کرده‌ام. گذشته از این، نوع صفت فقط یک کلاس است و یک کلاس می‌تواند صفت‌هایی بر آن اعمال شود. صفت **AttributeUsage** یک کلاس ساده است که به شما اجازه می‌دهد برای کامپایلر تعیین کنید صفت سفارشی شما کجا مجاز است اعمال شود. تمام کامپایلرها، پشتیبانی درونی برای این صفت دارند و وقتی یک صفت سفارشی تعریف شده توسط کاربر بر یک هدف غیرمعتبر اعمال شود، اعلام خطا می‌کنند. در این مثال، صفت **AttributeUsage** تعیین می‌کند نمونه‌هایی از صفت **Flags** می‌تواند فقط

به اهداف نوع شمارشی اعمال شوند. چون تمام صفت‌ها فقط نوع هستند، شما می‌توانید به راحتی کلاس **AttributeUsageAttribute** را درک کنید. سورس کد FCL برای این کلاس، شبیه به این است:

```
[Serializable]
[AttributeUsage(AttributeTargets.Class, Inherited=true)]
public sealed class AttributeUsageAttribute : Attribute {
    internal static AttributeUsageAttribute Default =
        new AttributeUsageAttribute(AttributeTargets.All);

    internal Boolean m_allowMultiple = false;
    internal AttributeTargets m_attributeTarget = AttributeTargets.All;
    internal Boolean m_inherited = true;

    // This is the one public constructor
    public AttributeUsageAttribute(AttributeTargets validOn) {
        m_attributeTarget = validOn;
    }

    internal AttributeUsageAttribute(AttributeTargets validOn,
        Boolean allowMultiple, Boolean inherited) {
        m_attributeTarget = validOn;
        m_allowMultiple = allowMultiple;
        m_inherited = inherited;
    }

    public Boolean AllowMultiple {
        get { return m_allowMultiple; }
        set { m_allowMultiple = value; }
    }

    public Boolean Inherited {
        get { return m_inherited; }
        set { m_inherited = value; }
    }

    public AttributeTargets validOn {
        get { return m_attributeTarget; }
    }
}
```

همانگونه که می‌بینید، کلاس **AttributeUsageAttribute** یک سازنده عمومی دارد که به شما اجازه می‌دهد پرچم‌های بیتی‌ای ارسال کنید که تعیین می‌کنند صفت شما در کجا مجاز است اعمال شود. نوع شمارشی **System.AttributeTargets** در FCL اینگونه تعریف شده است:

```
[Flags, Serializable]
public enum AttributeTargets {
    Assembly      = 0x0001,
    Module       = 0x0002,
    Class        = 0x0004,
    Struct       = 0x0008,
    Enum         = 0x0010,
```

```

Constructor      = 0x0020,
Method           = 0x0040,
Property         = 0x0080,
Field            = 0x0100,
Event            = 0x0200,
Interface        = 0x0400,
Parameter        = 0x0800,
Delegate         = 0x1000,
ReturnValue      = 0x2000,
GenericParameter = 0x4000,
All              = Assembly | Module | Class | Struct | Enum |
                  Constructor | Method | Property | Field | Event |
                  Interface | Parameter | Delegate | ReturnValue |
                  GenericParameter
}

```

کلاس **AttributeUsageAttribute** دو ویژگی عمومی اضافی نیز ارائه می‌کند که می‌توانند به صورت انتخابی وقتی صفت بر یک کلاس صفت اعمال می‌شود، تنظیم گردند: **Inherited** و **AllowMultiple**. برای اغلب صفت‌ها، معنی ندارد که آن‌ها را بر یک هدف بیش از یکبار اعمال کنید. برای نمونه، با اعمال بیش از یکبار صفت **Flags** یا **Serializable** به یک هدف چیزی حاصل نمی‌شود. در واقع، اگر شما سعی در کامپایل کد زیر داشته باشید، کامپایلر خطای زیر را اعلام می‌کند:

“error CS0579: Duplicate ‘Flags’ attribute.”

```

[Flags][Flags]
internal enum Color {
    Red
}

```

برای تعداد کمی از صفت‌ها، هرچند، این معنی پیدا می‌کند که صفت را چندین بار بر یک هدف اعمال کنید. در **FCL**، کلاس صفت **ConditionalAttribute** اجازه می‌دهد چندین نمونه از خودش بر یک هدف اعمال شود. اگر شما صریحاً **AllowMultiple** را به **true** تنظیم نکنید، صفت شما می‌تواند بیش از یکبار بر یک هدف انتخابی اعمال گردد.

دیگر ویژگی متعلق به صفت **AttributeUsageAttribute**، **Inherited**. بیان می‌کند وقتی صفت به یک کلاس پایه اعمال می‌شود آیا باید بر کلاس‌های مشتق شده و متدهای بازنویسی شده اعمال گردد یا خیر. کد زیر نشان می‌دهد ارث بردن یک صفت به چه معنی است:

```

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited=true)]
internal class TastyAttribute : Attribute {
}

```

```

[Tasty][Serializable]
internal class BaseType {
    [Tasty] protected virtual void DoSomething() { }
}

```

```

internal class DerivedType : BaseType {
    protected override void DoSomething() { }
}

```

در این کد، **DerivedType** و متدهای **DoSomething**، هر دو **Tasty** در نظر گرفته شده‌اند چون کلاس **TastyAttribute** به عنوان ارث برده شده (Inherited=true) علامت زده شده است. هر چند، **DerivedType** قابل سریالی کردن نیست چون کلاس **SerializableAttribute** از **FCL** به عنوان یک صفت غیر ارث برده شده علامت زده شده است.

آگاه باشید که داتنت فریمورک تنها اهدافی از کلاس‌ها، متدها، ویژگی‌ها، رویدادها، فیلدها، مقادیر برگشتی متد و پارامترها را قابل ارث بری در نظر می‌گیرد. پس وقتی شما یک نوع صفت تعریف می‌کنید، تنها اگر اهداف شما یکی از این هدف‌هاست شما باید **Inherited** را به **true** تنظیم کنید. توجه کنید که صفت‌های به ارث برده شده باعث نمی‌شوند متادیتای اضافی برای نوع مشتق شده در مازول مدیریت‌شده تولید شود. من در این باره کمی جلوتر در بخش "شناسایی استفاده از یک صفت سفارشی" بیشتر خواهیم گفت.

نکته اگر شما کلاس صفت خودتان را تعریف کنید و فراموش کنید یک صفت **AttributeUsage** بر کلاستان اعمال کنید، کامپایلر و CLR فرض می‌کنند صفت شما می‌تواند بر تمام اهداف اعمال شود، می‌تواند فقط یکبار بر یک هدف اعمال شود و ارث برده شده است. این فرضیات از مقادیر پیش فرض در کلاس **AttributeUsageAttribute** تقلید می‌کنند.

سازنده صفت و نوع های داده ای فیلد/ویژگی

هنگام تعریف کلاس صفت سفارشی خودتان، شما می‌توانید سازنده‌اش را تعریف کنید تا پارامترهایی بگیرد که برنامه‌نویسان هنگام اعمال یک نمونه از نوع صفت شما، باید تعیین کنند. به علاوه، شما می‌توانید فیلدها و ویژگی‌های عمومی غیر استاتیک در نوعتان تعریف کنید که تنظیماتی که یک برنامه‌نویس می‌تواند برای یک نمونه از کلاس صفت شما انتخاب کند را شناسایی می‌کند.

هنگام تعریف سازنده، فیلدها و ویژگی‌های نمونه از یک کلاس صفت، شما باید خودتان را به زیر مجموعه‌ی کوچکی از نوع‌های داده‌ای محدود کنید. به خصوص، مجموعه‌ی مجاز از نوع‌های داده‌ای به این‌ها محدود است: **Boolean**، **Char**، **Byte**، **SByte**، **Int16**، **UInt16**، **Int32**، **UInt32**، **Int64**، **UInt64**، **String**، **Type**، **Object**، یا یک نوع شمارشی. به علاوه، شما می‌توانید از یک آرایه تک بعدی پایه صفر از هر یک از این نوع‌ها استفاده کنید. هر چند، شما باید از استفاده از آرایه‌ها خودداری کنید چون یک کلاس صفت سفارشی که سازنده‌اش یک آرایه بگیرد سازگار با CLS نیست.

هنگام اعمال یک صفت، شما باید یک عبارت که در زمان کامپایل ثابت است و با نوع تعریف شده توسط کلاس صفت مطابق است ارسال کنید. هر جا که کلاس صفت یک پارامتر **Type**، فیلد **Type** یا ویژگی **Type** تعریف می‌کند شما باید از عملگر **typeof** سی‌شارپ که در کد زیر نشان داده شده است، استفاده کنید. هر جا که کلاس صفت یک پارامتر **Object**، فیلد **Object**، یا ویژگی **Object** تعریف می‌کند، شما می‌توانید یک **String** یا هر عبارت ثابت دیگری (شامل **null**) ارسال کنید. اگر عبارت ثابت، یک نوع مقداری را نمایش دهد، وقتی یک نمونه از صفت ساخته شود، نوع مقداری در زمان اجرا بسته‌بندی خواهد شد.

مثالی از یک صفت و استفاده اش:

```
using System;
```

```
internal enum Color { Red }
```

```
[AttributeUsage(AttributeTargets.All)]
```

```
internal sealed class SomeAttribute : Attribute {
    public SomeAttribute(String name, Object o, Type[] types) {
        // 'name' refers to a String
        // 'o' refers to one of the legal types (boxing if necessary)
        // 'types' refers to a 1-dimension, 0-based array of Types
    }
}
```

```
[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
```

```
internal sealed class SomeType {
}
```

به صورت منطقی، وقتی کامپایلر، یک صفت سفارشی اعمال شده بر یک هدف را می‌یابد، کامپایلر یک نمونه از کلاس صفت با فراخوانی سازنده‌اش و ارسال هر پارامتر تعیین شده‌ای می‌سازد. آنگاه کامپایلر هر فیلد و ویژگی عمومی را با استفاده از مقادیر تعیین شده در نحو سازنده مقداردهی اولیه می‌کند. حال که صفت سفارشی مقداردهی اولیه شد، کامپایلر وضعیت شی صفت را به درون ورودی جدول متادیتای هدف، سریالی می‌کند.

مهم من این را بهترین راه برای برنامه‌نویسان جهت تفکر درباره صفت‌های سفارشی یافته‌ام: نمونه‌هایی از کلاس‌هایی که به یک استریم بایت، سریالی شده و در متادیتا جای می‌گیرند. بعداً در زمان اجرا، یک نمونه از کلاس می‌تواند با غیرسریالی کردن بایت‌های درون متادیتا ساخته شود. در واقعیت، آنچه اتفاق می‌افتد اینست که کامپایلر اطلاعات لازم برای ساخت یک نمونه از کلاس صفت را در متادیتا تولید می‌کند. هر پارامتر سازنده با یک شناسه نوع ۱ بایتی به همراه مقدار نوشته می‌شود. پس از "سریالی کردن" پارامترهای سازنده، کامپایلر هر یک از مقادیر فیلد و ویژگی‌های تعیین شده را با نوشتن نام فیلد/ویژگی به همراه یک شناسه نوع ۱ بایتی و سپس مقدار، تولید می‌کند. برای آرایه‌ها، تعداد عناصر ابتدا ذخیره می‌شود و به دنبال آن تک تک عناصر ذخیره می‌شوند.

شناسایی استفاده از یک صفت سفارشی

تعریف یک کلاس صفت به تنهایی بی‌فایده است. مطمئناً، شما می‌توانید تمام کلاس‌های صفت را که می‌خواهید تعریف کرده و نمونه‌هایی از آن‌ها را که می‌خواهید، اعمال کنید اما این فقط باعث تولید متادیتای اضافی در اسمبلی می‌شود. رفتار برنامه شما تغییر نخواهد کرد.

در فصل ۱۵ "نوع‌های شمارشی و پرچم‌های بیتی" شما دیدید که اعمال صفت **Flags** بر یک نوع شمارشی، رفتار متدهای **ToString** و **Format** از **System.Enum** را تغییر داد. علت این که این متدها متفاوت رفتار می‌کنند اینست که آن‌ها در زمان اجرا بررسی می‌کنند آیا نوع شمارشی که بر روی آن عمل می‌کنند دارای متادیتای صفت **Flags** است یا خیر. کد می‌تواند وجود صفت‌ها را با استفاده از یک تکنولوژی به نام رفلکشن **reflection** تشخیص دهد. من نمایش کوچکی از رفلکشن را در اینجا نشان می‌دهم اما آن‌ها را به صورت کامل در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن" بحث می‌کنم. اگر شما کارمند مایکروسافت و مسئول پیاده‌سازی متد **Format** از **Enum** بودید، شما آن را اینگونه پیاده‌سازی می‌کردید:

```
public static String Format(Type enumType, Object value, String format) {
```

```
    // Does the enumerated type have an instance of
    // the FlagsAttribute type applied to it?
    if (enumType.IsDefined(typeof(FlagsAttribute), false)) {
        // Yes; execute code treating value as a bit flag enumerated type.
        ...
    } else {
        // No; execute code treating value as a normal enumerated type.
        ...
    }
    ...
}
```

این کد، متد **IsDefined** از **Type** را فراخوانی می‌کند و به صورت موثر از سیستم درخواست می‌کند که به دنبال متادیتا برای نوع شمارشی بگردد و ببیند آیا یک نمونه از کلاس **FlagsAttribute** همراه آن هست یا خیر.

اگر **IsDefined**، **true** برگرداند، یک نمونه از **FlagsAttribute** با نوع شمارشی همراه است و متد **Format** می‌داند که با مقدار به گونه‌ای رفتار کند که گویا حاوی یک مجموعه از پرچم‌های بیتی است. اگر **IsDefined**، **false** برگرداند، **Format** با مقدار به عنوان یک نوع شمارشی معمولی رفتار می‌کند. پس اگر شما کلاس‌های صفت خودتان را تعریف کنید، شما باید کدی را هم پیاده‌سازی کنید که وجود یک نمونه از کلاس صفت شما (روی یک هدف) را بررسی کند و آنگاه کد جایگزینی را اجرا کند. این چیزی است که صفت‌های سفارشی را بسیار کاربردی و مفید می‌کند.

FCL راه‌های زیادی برای بررسی وجود یک صفت ارائه می‌کند. اگر شما وجود یک صفت را از طریق یک شی **System.Object** بررسی کنید، شما می‌توانید از متد **IsDefined** که قبلاً نشان داده شد استفاده کنید. هر چند، گاهی اوقات شما می‌خواهید یک صفت را روی یک هدف به جز یک نوع، بررسی کنید، مثل یک اسمبلی، یک ماژول یا یک متد. برای این بحث، بگذارید روی متدهای تعریف شده توسط کلاس **System.Attribute** تمرکز کنیم. شما

بخاطر دارید که تمام صفت‌های سازگار با CLS از **System.Attribute** مشتق شده‌اند. این کلاس سه متد استاتیک برای بدست آوردن صفت‌های همراه با یک هدف تعریف می‌کند: **IsDefined**، **GetCustomAttributes** و **GetCustomAttribute**. هر یک از این سه تابع، چندین نسخه بارگذاری شده دارند. برای مثال، هر متد یک نسخه دارد که روی اعضای نوع (کلاس‌ها، ساختارها، شمارشی‌ها، رابط‌ها، نماینده‌ها، سازنده‌ها، متدها، ویژگی‌ها، فیلدها، رویدادها و نوع‌ها برگشتی)، پارامترها، ماژول‌ها و اسمبلی‌ها کار می‌کند. همچنین نسخه‌هایی وجود دارد که به شما اجازه می‌دهند به سیستم بگوییید در سلسله مراتب وراثت حرکت کند تا صفت‌های به ارث برده شده را در نتایج، شامل کند. جدول ۱-۱۸ به صورت خلاصه آنچه هر یک از متدها انجام می‌دهد را توصیف می‌کند.

جدول ۱-۱۸ متدهای **System.Attribute** که روی متادیتا منعکس می‌شوند (رفلکشن) و به دنبال نمونه‌های صفت‌های سفارشی سازگار با CLS می‌گردند

متد	توضیح
IsDefined	اگر حداقل یک نمونه از کلاس مشتق شده از Attribute همراه با هدف باشد، true برمی‌گرداند. این متد کاراست چون نمونه‌هایی از کلاس صفت را نمی‌سازد (غیر سریالی نمی‌کند).
GetCustomAttributes	یک آرایه که در آن هر عنصر یک نمونه از کلاس صفت تعیین شده می‌باشد که بر هدف اعمال گردیده است را برمی‌گرداند. اگر هیچ کلاس صفتی به متد داده نشود، آرایه حاوی تمام صفت‌های اعمال شده است؛ هر کلاسی می‌خواهند داشته باشند. هر نمونه با استفاده از پارامترها، فیلدها و ویژگی‌های تعیین شده حین کامپایل، ساخته (غیر سریالی) می‌شود. اگر هدف، هیچ نمونه‌ای از کلاس صفت تعیین شده را نداشته، یک آرایه خالی برمی‌گردد. این متد نوعاً با صفت‌هایی که AllowMultiple برای آن‌ها true شده باشد یا برای لیست کردن تمام صفت‌های اعمال شده، استفاده می‌شود.
GetCustomAttribute	یک نمونه از کلاس صفت تعیین شده که بر هدف اعمال گردیده را برمی‌گرداند. نمونه با استفاده از پارامترها، فیلدها و ویژگی‌های تعیین شده حین کامپایل، ساخته (غیرسریالی) می‌شود. اگر هدف، نمونه‌ای از کلاس صفت تعیین شده را نداشته باشد، null برمی‌گردد. اگر هدف چندین نمونه از صفت تعیین شده بر آن اعمال گردیده باشد، یک اکسپشن System.Reflection.AmbiguousMatchException تولید می‌شود. این متد نوعاً با صفت‌هایی که AllowMultiple برای آن‌ها false است استفاده می‌شود.

اگر شما بخواهید فقط ببینید آیا یک صفت بر یک هدف اعمال شده است، شما باید **IsDefined** را فراخوانی کنید چون از دو متد دیگر کارا تر است. هر چند، شما می‌دانید که وقتی یک صفت بر یک هدف اعمال می‌شود، شما می‌توانید، پارامترهایی برای سازنده صفت تعیین کرده و به صورت انتخابی فیلدها و ویژگی‌ها را تنظیم کنید. استفاده از **IsDefined** یک شی صفت نخواهد ساخت که سازنده‌اش را فراخوانی کند یا فیلد و ویژگی‌هایش را تنظیم کند.

اگر شما می‌خواهید یک شی صفت بسازید، شما باید **GetCustomAttributes** یا **GetCustomAttribute** را فراخوانی کنید. هر بار یکی از این متدها فراخوانی می‌شود، نمونه‌های جدیدی از نوع صفت تعیین شده می‌سازد و هر یک از فیلدها و ویژگی‌های متعلق به نمونه را بر اساس مقادیر تعیین شده در سورس کد تنظیم می‌کنند. این متدها ارجاع‌هایی به نمونه‌های کاملاً ساخته شده از کلاس‌های صفت‌های اعمال شده را برمی‌گرداند.

وقتی شما یکی از این متدها را فراخوانی می‌کنید، در درون، آن‌ها باید متادیتای ماژول مدیریت شده را اسکن کنند، مقایسه‌های رشته‌ای انجام دهند تا کلاس صفت سفارشی تعیین شده را بیابند. واضح است، این عملیات‌ها زمان می‌برد. اگر شما نگران عملکرد هستید، شما باید به دنبال ذخیره نتایج این متدها به جای آنکه مکرراً همان اطلاعات را درخواست کنید، باشید. فضای نام **System.Reflection** چندین کلاس تعریف می‌کند که به شما اجازه می‌دهند محتویات متادیتای یک ماژول را بررسی کنید: **Assembly**، **Module**، **ParameterInfo**، **MemberInfo**، **Type**، **MethodInfo**، **ConstructorInfo**، **FieldInfo**، **EventInfo**، **PropertyInfo** و کلاس‌های ***Builder**. متناظر آن‌ها، تمام این کلاس‌ها هم متدهای **IsDefined** و **GetCustomAttributes** را ارائه می‌کنند. تنها **System.Attribute**، متد بسیار رایج **GetCustomAttribute** را ارائه می‌کند.

نسخه‌ای از **GetCustomAttributes** که توسط کلاس‌های رفلکشن تعریف شده‌اند یک آرایه از نمونه‌های **Object[] (Object[])** به جای یک آرایه از نمونه‌های **Attribute[] (Attribute[])** برمی‌گرداند. این بدین خاطر است که کلاس‌های رفلکشن قادرند اشیاء کلاس‌های صفت غیرسازگار با CLS را برگردانند. شما نباید درباره‌ی این ناهماهنگی نگران باشید چون صفت‌های سازگار با CLS بسیار نادرند. در واقع، در تمام زمانی که من با دات‌نت فریم‌ورک کار می‌کنم، تاکنون حتی یکی از آن‌ها را هم ندیده‌ام.

نکته آگاه باشید که تنها کلاس‌های **Attribute**، **Type** و **MethodInfo** متدهای رفلکشنی را پیاده سازی می‌کنند که برای پارامتر **inherit**، **Boolean**، ارزش قائلند. تمام دیگر متدها که صفت‌ها را جستجو می‌کنند پارامتر **inherit** را نادیده می‌گیرند و سلسله مراتب وراثت را بررسی نمی‌کنند. اگر شما نیاز به بررسی وجود یک صفت به ارث برده شده برای رویدادها، ویژگی‌ها، فیلدها، سازنده‌ها یا پارامترها دارید، شما باید یکی از متدهای **Attribute** را فراخوانی کنید.

یک چیز دیگر هست که شما باید از آن آگاه باشید: وقتی شما یک کلاس به **GetCustomAttribute.IsDefined** یا **GetCustomAttributes** ارسال می‌کنید، این متدها به دنبال موارد استفاده از کلاس صفت تعیین شده توسط شما یا هر کلاس صفت مشتق شده از کلاس تعیین شده، می‌گردند. اگر کد شما به دنبال یک کلاس صفت خاص است، شما باید یک بررسی اضافی روی مقدار برگشتی انجام دهید تا مطمئن شوید که آنچه این متدها برمی‌گردانند دقیقاً آن چیزی است که شما به دنبال آن هستید. شاید هم شما بخواهید کلاس صفت خود را **sealed** تعریف کنید تا سردرگمی احتمالی را کاهش داده و این بررسی اضافی را حذف کنید.

در اینجا کد نمونه‌ای آمده است که تمام متدهای تعریف شده درون یک نوع را لیست کرده و صفت اعمال شده به هر متد را نشان می‌دهد. کد فقط برای نمایش است، به صورت عادی شما این صفت‌های سفارشی را به این اهداف آنگونه که من انجام داده‌ام، اعمال نمی‌کنید.

```
using System;
using System.Diagnostics;
using System.Reflection;

[assembly: CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name = "Jeff", Target = typeof(Program))]
public sealed class Program {
    [Conditional("Debug")]
    [Conditional("Release")]
    public void DoSomething() { }

    public Program() {
    }

    [CLSCompliant(true)]
    [STAThread]
    public static void Main() {
        // Show the set of attributes applied to this type
        ShowAttributes(typeof(Program));

        // Get the set of methods associated with the type
        MemberInfo[] members = typeof(Program).FindMembers(
            MemberTypes.Constructor | MemberTypes.Method,
            BindingFlags.DeclaredOnly | BindingFlags.Instance |
            BindingFlags.Public | BindingFlags.Static,
            Type.FilterName, "*");

        foreach (MemberInfo member in members) {
            // Show the set of attributes applied to this member
            ShowAttributes(member);
        }
    }
}
```

```

    }
}

private static void ShowAttributes(MemberInfo attributeTarget) {
    Attribute[] attributes = Attribute.GetCustomAttributes(attributeTarget);

    Console.WriteLine("Attributes applied to {0}: {1}",
        attributeTarget.Name, (attributes.Length == 0 ? "None" : String.Empty));
    foreach (Attribute attribute in attributes) {
        // Display the type of each applied attribute
        Console.WriteLine(" {0}", attribute.GetType().ToString());
        if (attribute is DefaultMemberAttribute)
            Console.WriteLine(" MemberName={0}",
                ((DefaultMemberAttribute) attribute).MemberName);

        if (attribute is ConditionalAttribute)
            Console.WriteLine(" ConditionString={0}",
                ((ConditionalAttribute) attribute).ConditionString);

        if (attribute is CLSCompliantAttribute)
            Console.WriteLine(" IsCompliant={0}",
                ((CLSCompliantAttribute) attribute).IsCompliant);

        DebuggerDisplayAttribute dda = attribute as DebuggerDisplayAttribute;
        if (dda != null) {
            Console.WriteLine(" Value={0}, Name={1}, Target={2}",
                dda.Value, dda.Name, dda.Target);
        }
    }
    Console.WriteLine();
}
}

```

ساخت و اجرای این برنامه خروجی زیر را تولید می‌کند:

```

Attributes applied to Program:
System.SerializableAttribute
System.Diagnostics.DebuggerDisplayAttribute
    Value=Richter, Name=Jeff, Target=Program
System.Reflection.DefaultMemberAttribute
    MemberName=Main

Attributes applied to DoSomething:
System.Diagnostics.ConditionalAttribute
    ConditionString=Release
System.Diagnostics.ConditionalAttribute
    ConditionString=Debug

Attributes applied to Main:
System.CLSCompliantAttribute

```

```
IsCompliant=True
System.STAThreadAttribute
```

Attributes applied to .ctor: None

بررسی تطابق دو نمونه صفت در مقابل هم

حال که کد شما می‌داند چگونه بررسی کند آیا یک نمونه از یک صفت بر یک هدف اعمال شده است، اکنون شاید بخواهد فیلدهای صفت را بررسی کند تا ببیند چه مقادیری دارند. یک راه انجام چنین کاری نوشتن کدی است که صریحاً مقادیر فیلدهای کلاس صفت را بررسی کند. به هر حال، **System.Attribute**، متد **Equals** از **Object** را بازنویسی می‌کند و در درون، این متد، نوع‌های دو شی را بررسی می‌کند. اگر آن‌ها یکی نبودند، **Equals**، **false** برمی‌گرداند. اگر نوع‌ها یکی بودند، سپس **Equals** از رفلکشن برای مقایسه مقادیر فیلدهای دو شی صفت (با فراخوانی **Equals** بر روی هر فیلد) استفاده می‌کند. اگر تمام فیلدها مطابق بودند، آنگاه **true** برگردانده می‌شود، در غیر این صورت **false** برگردانده می‌شود. شاید شما **Equals** را در کلاس صفت خودتان بازنویسی کنید تا استفاده از رفلکشن را حذف کرده و عملکرد را بهبود بخشید.

System.Attribute یک متد مجازی **Match** نیز ارائه می‌کند که شما می‌توانید برای فراهم کردن معانی غنی‌تر، آن را بازنویسی کنید. پیاده‌سازی پیش فرض از **Match** به سادگی **Equals** را فراخوانی کرده و نتیجه‌اش را برمی‌گرداند. کد زیر نشان می‌دهد چگونه **Match** و **Equals** (که اگر یک صفت یک زیرمجموعه از دیگری را نمایش دهد، **true** برمی‌گرداند) را بازنویسی کنید و سپس نشان می‌دهد چگونه **Match** استفاده می‌شود:

```
using System;
```

```
[Flags]
```

```
internal enum Accounts {
    Savings = 0x0001,
    Checking = 0x0002,
    Brokerage = 0x0004
}
```

```
[AttributeUsage(AttributeTargets.Class)]
```

```
internal sealed class AccountsAttribute : Attribute {
    private Accounts m_accounts;

    public AccountsAttribute(Accounts accounts) {
        m_accounts = accounts;
    }

    public override Boolean Match(Object obj) {
        // If the base class implements Match and the base class
        // is not Attribute, then uncomment the line below.
        // if (!base.Match(obj)) return false;

        // Since 'this' isn't null, if obj is null,
        // then the objects can't match
        // NOTE: This line may be deleted if you trust
        // that the base type implemented Match correctly.
        if (obj == null) return false;

        // If the objects are of different types, they can't match
        // NOTE: This line may be deleted if you trust
        // that the base type implemented Match correctly.
```

```

    if (this.GetType() != obj.GetType()) return false;

    // Cast obj to our type to access fields. NOTE: This cast
    // can't fail since we know objects are of the same type
    AccountsAttribute other = (AccountsAttribute) obj;

    // Compare the fields as you see fit
    // This example checks if 'this' accounts is a subset
    // of others' accounts
    if ((other.m_accounts & m_accounts) != m_accounts)
        return false;

    return true; // Objects match
}

public override Boolean Equals(Object obj) {
    // If the base class implements Equals, and the base class
    // is not Object, then uncomment the line below.
    // if (!base.Equals(obj)) return false;

    // Since 'this' isn't null, if obj is null,
    // then the objects can't be equal
    // NOTE: This line may be deleted if you trust
    // that the base type implemented Equals correctly.
    if (obj == null) return false;

    // If the objects are of different types, they can't be equal
    // NOTE: This line may be deleted if you trust
    // that the base type implemented Equals correctly.
    if (this.GetType() != obj.GetType()) return false;

    // Cast obj to our type to access fields. NOTE: This cast
    // can't fail since we know objects are of the same type
    AccountsAttribute other = (AccountsAttribute) obj;

    // Compare the fields to see if they have the same value
    // This example checks if 'this' accounts is the same
    // as other's accounts
    if (other.m_accounts != m_accounts)
        return false;

    return true; // Objects are equal
}

// Override GetHashCode since we override Equals
public override Int32 GetHashCode() {
    return (Int32) m_accounts;
}

```

```

}

[Accounts(Accounts.Savings)]
internal sealed class ChildAccount { }

[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
internal sealed class AdultAccount { }

public sealed class Program {
    public static void Main() {
        CanWriteCheck(new ChildAccount());
        CanWriteCheck(new AdultAccount());

        // This just demonstrates that the method works correctly on a
        // type that doesn't have the AccountsAttribute applied to it.
        CanWriteCheck(new Program());
    }

    private static void CanWriteCheck(Object obj) {
        // Construct an instance of the attribute type and initialize it
        // to what we are explicitly looking for.
        Attribute checking = new AccountsAttribute(Accounts.Checking);

        // Construct the attribute instance that was applied to the type
        Attribute validAccounts = Attribute.GetCustomAttribute(
            obj.GetType(), typeof(AccountsAttribute), false);

        // If the attribute was applied to the type AND the
        // attribute specifies the "Checking" account, then the
        // type can write a check
        if ((validAccounts != null) && checking.Match(validAccounts)) {
            Console.WriteLine("{0} types can write checks.", obj.GetType());
        } else {
            Console.WriteLine("{0} types can NOT write checks.", obj.GetType());
        }
    }
}
}

```

ساخت و اجرای این برنامه خروجی زیر را تولید می‌کند:

```

ChildAccount types can NOT write checks.
AdultAccount types can write checks.
Program types can NOT write checks.

```

شناسایی استفاده از یک صفت سفارشی بدون ساخت اشیاء مشتق شده از Attribute

در این بخش، من یک راه جایگزین برای شناسایی صفت‌های سفارشی اعمال شده بر یک ورودی متادیتا را بحث می‌کنم. در برخی سناریوهایی که امنیت مهم است، این تکنیک جایگزین، اطمینان حاصل می‌کند هیچ کدی در یک کلاس مشتق شده از **Attribute**، اجرا نخواهد شد. گذشته از این، وقتی شما متدهای **GetCustomAttribute(s)** از **Attribute** را فراخوانی می‌کنید، در درون، این متدها سازنده کلاس صفت را فراخوانی کرده و می‌توانند

متدهای دستیابی **set** ویژگی را نیز فراخوانی کنند. به علاوه، اولین دسترسی به یک نوع باعث می‌شود CLR سازنده‌ی نوع (type constructor) متعلق به نوع را (در صورت وجود) فراخوانی کند. سازنده، متد دستیابی **set** و متدهای سازنده نوع می‌توانند حاوی کدی باشند که هرگاه کدی فقط به دنبال یک صفت می‌گردد، اجرا شوند. این اجازه می‌دهد کد ناشناخته در **AppDomain** اجرا شود و این آسیب پذیری امنیتی احتمالی را به دنبال دارد.

برای یافتن صفت‌ها بدون اجازه دادن به اجرای کد کلاس صفت، شما از کلاس **System.Reflection.CustomAttributeData** استفاده می‌کنید. این کلاس یک متد استاتیک برای بدست آوردن صفت‌های همراه با یک هدف ارائه می‌کند: **GetCustomAttributes**. این متد چهار سربارگذاری دارد: یکی از آن‌ها یک **Assembly** می‌گیرد، یکی از آن‌ها یک **Module**، یکی از آن‌ها یک **ParameterInfo** و یکی از آن‌ها یک **MemberInfo** می‌گیرد. این کلاس در فضای نام **System.Reflection** تعریف شده است که در فصل ۲۳ بحث خواهد شد. نوعاً، شما از کلاس **CustomAttributeData** برای آنالیز صفت‌های درون متادیتا برای یک اسمبلی که از طریق متد استاتیک **ReflectionOnlyLoad** از **Assembly**، بارگذاری شده است استفاده می‌کنید (این هم در فصل ۲۳ بحث می‌شود). به طور خلاصه، **ReflectionOnlyLoad** یک اسمبلی را به نحوی بارگذاری می‌کند که مانع از آن می‌شود CLR هر گونه کدی درون آن را اجرا کند، این شامل سازنده‌های نوع نیز می‌شود.

متد **GetCustomAttributes** از **CustomAttributeData**، مثل یک کارخانه عمل می‌کند. یعنی، وقتی شما آن را فراخوانی می‌کنید، یک مجموعه از اشیاء **CustomAttributeData** در یک شی از نوع **IList<CustomAttributeData>** برمی‌گرداند. مجموعه حاوی یک عنصر به ازای هر صفت سفارشی اعمال شده بر هدف تعیین شده می‌باشد. برای هر شی **CustomAttributeData**، شما می‌توانید تعدادی ویژگی‌های فقط-خواندنی آن را بخوانید تا تعیین کنید صفت چگونه ساخته و مقداردهی اولیه می‌شود. به خصوص، ویژگی **Constructor** بیان می‌کند کدام متد سازنده می‌بایست فراخوانی شود، ویژگی **ConstructorArguments** آرگومان‌هایی که می‌بایست به این سازنده ارسال شوند را در قالب یک نمونه از **IList<CustomAttributeTypedArgument>** برمی‌گرداند، و ویژگی **NamedArguments** فیلد/ویژگی‌هایی که می‌بایست تنظیم شوند را به عنوان یک نمونه از **IList<CustomAttributeNamedArgument>** برمی‌گرداند. توجه کنید من در جملات قبلی گفتم "می‌بایست" چون سازنده و متدهای دستیابی **set** در حقیقت فراخوانی نمی‌شوند – ما امنیت بیشتری با جلوگیری از اجرای متدهای کلاس صفت، بدست آورده‌ایم.

نسخه تغییر یافته نمونه کد قبلی که از کلاس **CustomAttributeData** برای بدست آوردن امن صفت‌های اعمالی بر اهداف مختلف استفاده می‌کند:

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name="Jeff", Target=typeof(Program))]
public sealed class Program {
    [Conditional("Debug")]
    [Conditional("Release")]
    public void DoSomething() { }

    public Program() {
    }

    [CLSCompliant(true)]
    [STAThread]
    public static void Main() {
        // Show the set of attributes applied to this type
        ShowAttributes(typeof(Program));

        // Get the set of methods associated with the type
```

```

MemberInfo[] members = typeof(Program).FindMembers(
    MemberTypes.Constructor | MemberTypes.Method,
    BindingFlags.DeclaredOnly | BindingFlags.Instance |
    BindingFlags.Public | BindingFlags.Static,
    Type.FilterName, "*");

foreach (MemberInfo member in members) {
    // Show the set of attributes applied to this member
    ShowAttributes(member);
}
}

private static void ShowAttributes(MemberInfo attributeTarget) {
    IList<CustomAttributeData> attributes =
        CustomAttributeData.GetCustomAttributes(attributeTarget);

    Console.WriteLine("Attributes applied to {0}: {1}",
        attributeTarget.Name, (attributes.Count == 0 ? "None" : String.Empty));

    foreach (CustomAttributeData attribute in attributes) {
        // Display the type of each applied attribute
        Type t = attribute.Constructor.DeclaringType;
        Console.WriteLine(" {0}", t.ToString());
        Console.WriteLine(" Constructor called={0}", attribute.Constructor);

        IList<CustomAttributeTypedArgument> posArgs = attribute.ConstructorArguments;
        Console.WriteLine(" Positional arguments passed to constructor:" +
            ((posArgs.Count == 0) ? " None" : String.Empty));
        foreach (CustomAttributeTypedArgument pa in posArgs) {
            Console.WriteLine(" Type={0}, Value={1}", pa.ArgumentType, pa.Value);
        }

        IList<CustomAttributeNamedArgument> namedArgs = attribute.NamedArguments;
        Console.WriteLine("Named arguments set after construction:" +
            ((namedArgs.Count == 0) ? " None" : String.Empty));
        foreach(CustomAttributeNamedArgument na in namedArgs) {
            Console.WriteLine(" Name={0}, Type={1}, Value={2}",
                na.MemberInfo.Name, na.TypedValue.ArgumentType, na.TypedValue.Value);
        }

        Console.WriteLine();
    }
    Console.WriteLine();
}
}
}

```

ساخت و اجرای این برنامه خروجی زیر را تولید می‌کند:

Attributes applied to Program:

System.SerializableAttribute
Constructor called=Void .ctor()
Positional arguments passed to constructor: None
Named arguments set after construction: None

System.Diagnostics.DebuggerDisplayAttribute
Constructor called=Void .ctor(System.String)
Positional arguments passed to constructor:
Type=System.String, Value=Richter
Named arguments set after construction:
Name=Name, Type=System.String, Value=Jeff
Name=Target, Type=System.Type, Value=Program

System.Reflection.DefaultMemberAttribute
Constructor called=Void .ctor(System.String)
Positional arguments passed to constructor:
Type=System.String, Value=Main
Named arguments set after construction: None

Attributes applied to DoSomething:

System.Diagnostics.ConditionalAttribute
Constructor called=Void .ctor(System.String)
Positional arguments passed to constructor:
Type=System.String, Value=Release
Named arguments set after construction: None

System.Diagnostics.ConditionalAttribute
Constructor called=Void .ctor(System.String)
Positional arguments passed to constructor:
Type=System.String, Value=Debug
Named arguments set after construction: None

Attributes applied to Main:

System.CLSCompliantAttribute
Constructor called=Void .ctor(Boolean)
Positional arguments passed to constructor:
Type=System.Boolean, Value=True
Named arguments set after construction: None

System.STAThreadAttribute
Constructor called=Void .ctor()
Positional arguments passed to constructor: None
Named arguments set after construction: None

Attributes applied to .ctor: None

کلاس های صفت شرطی

در طول زمان، سادگی تعریف، اعمال و رفلکشن روی صفت‌ها باعث شد برنامه‌نویسان از آن‌ها بیشتر و بیشتر استفاده کنند. استفاده از صفت‌ها یک راه بسیار آسان برای حاشیه نویسی کدتان است در حالیکه ویژگی‌های بسیار غنی را همزمان پیاده سازی می‌کنید. اخیراً، برنامه‌نویسان از صفت‌ها برای کمک به طراحی و خطایابی استفاده می‌کنند. برای نمونه، ابزار آنالیز کد ویژوال استودیو (FxCopCmd.exe) یک مانع از گزارش نقض یک قانون خاص، توسط ابزار آنالیز شوید. این صفت تنها توسط کد ابزار آنالیز جستجو می‌شود، صفت هرگز وقتی یک برنامه به صورت عادی اجرا می‌شود، جستجو نمی‌گردد. وقتی از آنالیز کد استفاده نمی‌شود، وجود صفت‌های **SuppressMessage** در متادیتا منجر به بزرگ شدن آن می‌شود و در نتیجه بر عملکرد برنامه شما ضربه می‌زند. خیلی خوب است اگر راهی آسان باشد که کامپایلر، صفت‌های **SuppressMessage** را تنها در صورتی که شما قصد دارید از ابزار آنالیز کد استفاده کنید، تولید کند. خوشبختانه، راهی برای انجام این کار با استفاده از کلاس‌های صفت‌های شرطی وجود دارد.

یک کلاس صفت که **System.Diagnostics.ConditionalAttribute** بر آن اعمال شده باشد یک کلاس صفت شرطی `conditional attribute class` نامیده می‌شود. یک نمونه ببینید:

```
//#define TEST
#define VERIFY

using System;
using System.Diagnostics;

[Conditional("TEST")][Conditional("VERIFY")]
public sealed class CondAttribute : Attribute {
}

[Cond]
public sealed class Program {
    public static void Main() {
        Console.WriteLine("CondAttribute is {0}applied to Program type.",
            Attribute.IsDefined(typeof(Program),
                typeof(CondAttribute)) ? "" : "not ");
    }
}
```

وقتی کامپایلر می‌بیند یک نمونه از **CondAttribute** بر یک هدف اعمال شده است، کامپایلر تنها در صورتی اطلاعات صفت را در متادیتا تولید می‌کند که نماد **TEST** یا **VERIFY** وقتی کد حاوی هدف کامپایل می‌شود تعریف شده باشند. هرچند، متادیتا، تعریف کلاس صفت و پیاده‌سازی هنوز هم در اسمبلی وجود دارند.

فصل ۱۹: نوع های مقداری تهی پذیر

همانگونه که می دانید، یک متغیر نوع مقداری هرگز نمی تواند **null** باشد و آن همیشه خودش حاوی مقدار نوع مقداری است. در واقع، به همین علت به آن ها نوع مقداری می گویند. متاسفانه، سناریوهایی وجود دارد که این یک مشکل است. برای نمونه، هنگام طراحی یک پایگاه داده، این امکان وجود دارد که نوع داده ای یک ستون را یک عدد ۳۲ بیتی تعریف کنید که به نوع داده **Int32** در کتابخانه کلاس فریمورک (FCL) منطبق شود. اما یک ستون در یک پایگاه داده می تواند یک مقدار تهی پذیر را نشان دهد. یعنی نداشتن مقدار در یک ردیف و ستون، مشکلی ندارد. کار با داده های پایگاه داده با استفاده از دات نت فریمورک می تواند بسیار مشکل باشد. در CLR راهی وجود ندارد که یک مقدار **Int32** را به عنوان **null** نمایش دهید.

نکته Table Adapter های ADO.NET از نوع های تهی پذیر پشتیبانی می کنند. اما متاسفانه، نوع های درون فضای نام **System.Data.SqlTypes** با نوع های تهی پذیر جایگزین نمی شوند، چون تطابق یک به یک بین دو نوع وجود ندارد. برای نمونه، نوع **SqlDecimal** ماکزیمم ۳۸ رقم دارد در حالیکه نوع معمولی **Decimal** فقط به ۲۹ رقم می رسد. به علاوه نوع **SqlString** دارای اطلاعات محل و مقایسه در درون خودش می باشد که توسط نوع عادی **String** پشتیبانی نمی شود.

یک مثال دیگر: در جاوا، کلاس **java.util.Date** یک نوع ارجاعی است و بنابراین، یک متغیر از نوع می تواند به **null** تنظیم شود اما در CLR، یک **System.DateTime** یک نوع مقداری است و یک متغیر **DateTime** هرگز نمی تواند **null** شود. اگر یک برنامه نوشته شده در جاوا بخواهد با یک وب سرویس که با CLR کار می کند یک تاریخ/زمان را تبادل کند، اگر برنامه جاوا یک **null** ارسال کند مشکلی پیش می آید چون CLR راهی برای نمایش و عمل روی آن ندارد.

برای بهبود این وضعیت، مایکروسافت مفهوم نوع های مقداری تهی پذیر را به CLR افزود. برای درک چگونگی عملکرد آن ها، ابتدا باید به کلاس **System.Nullable<T>** که در FCL تعریف شده است نگاه کنیم. نمایش منطقی نوع **System.Nullable<T>** اینگونه است:

```
[Serializable, StructLayout(LayoutKind.Sequential)]
```

```
public struct Nullable<T> where T : struct {
```

```
    // These 2 fields represent the state
    private Boolean hasValue = false;    // Assume null
    internal T value = default(T);      // Assume all bits zero
    public Nullable(T value) {
        this.value = value;
        this.hasValue = true;
    }
```

```
    public Boolean HasValue { get { return hasValue; } }
```

```
    public T value {
        get {
            if (!hasValue) {
                throw new InvalidOperationException(
                    "Nullable object must have a value.");
            }
            return value;
        }
    }
}
```

```
public T GetValueOrDefault() { return value; }
```

```

public T GetValueOrDefault(T defaultValue) {
    if (!HasValue) return defaultValue;
    return value;
}

public override Boolean Equals(Object other) {
    if (!HasValue) return (other == null);
    if (other == null) return false;
    return value.Equals(other);
}

public override int GetHashCode() {
    if (!HasValue) return 0;
    return value.GetHashCode();
}

public override string ToString() {
    if (!HasValue) return "";
    return value.ToString();
}

public static implicit operator Nullable<T>(T value) {
    return new Nullable<T>(value);
}

public static explicit operator T(Nullable<T> value) {
    return value.Value;
}
}

```

همانگونه که می‌بینید، این کلاس مفهوم یک نوع مقداری که **null** هم می‌تواند باشد را کپسوله می‌کند. چون **Nullable<T>** خودش یک نوع مقداری است، نمونه‌های آن بسیار سبک هستند یعنی، نمونه‌ها می‌توانند هنوز در پشته باشند و یک نمونه از آن، هم اندازه‌ی نوع مقداری اصلی به اضافه‌ی اندازه‌ی یک فیلد **Boolean** است. توجه کنید که پارامتر نوع از **Nullable.T**، به **struct** محدود شده است. این بدین خاطر است که نوع‌های ارجاعی از قبل می‌توانند **null** باشند.

پس اکنون، اگر شما بخواهید از یک **Int32** تهی پذیر در کدتان استفاده کنید، شما می‌توانید چیزی شبیه به این بنویسید:

```

Nullable<Int32> x = 5;
Nullable<Int32> y = null;
Console.WriteLine("x: HasValue={0}, Value={1}", x.HasValue, x.Value);
Console.WriteLine("y: HasValue={0}, Value={1}", y.HasValue, y.GetValueOrDefault());

```

وقتی من این کد را کامپایل و اجرا می‌کنم، خروجی زیر را می‌گیرم:

```

x: HasValue=True, Value=5
y: HasValue=False, Value=0

```

پشتیبانی سی‌شارپ برای نوع‌های مقداری تهی پذیر

در کد توجه کنید که سی‌شارپ به شما اجازه داد از نحو بسیار ساده‌ای برای مقداردهی دو متغیر **Nullable<Int32>** **x** و **y**، استفاده کنید. در حقیقت، تیم سی‌شارپ می‌خواهند نوع‌های مقداری تهی پذیر را در زبان سی‌شارپ نهادینه کرده، آن‌ها را به شهروندان درجه اول تبدیل کنند. تا آن زمان، سی‌شارپ

یک نحو تمیزتر برای کار با نوع‌های مقداری تهی پذیر ارائه می‌کند. سی‌شارپ به کد اجازه می‌دهد متغیرهای **x** و **y** را همراه با علامت سوال، اینگونه تعریف کنند:

```
Int32? x = 5;
Int32? y = null;
```

در سی‌شارپ **Int32?** مترادف **Nullable<Int32>** است. اما سی‌شارپ از این هم فراتر می‌رود. سی‌شارپ به شما اجازه می‌دهد تا تبدیل‌ها را روی نمونه‌های تهی پذیر انجام دهید و سی‌شارپ همچنین اعمال عملگر بر نمونه‌های تهی پذیر را پشتیبانی می‌کند. کد زیر نمونه‌هایی از این را نشان می‌دهد:

```
private static void ConversionsAndCasting() {
    // Implicit conversion from non-nullable Int32 to Nullable<Int32>
    Int32? a = 5;

    // Implicit conversion from 'null' to Nullable<Int32>
    Int32? b = null;

    // Explicit conversion from Nullable<Int32> to non-nullable Int32
    Int32 c = (Int32) a;

    // Casting between nullable primitive types
    Double? d = 5; // Int32->Double? (d is 5.0 as a double)
    Double? e = b; // Int32?->Double? (e is null)
}
```

سی‌شارپ همچنین به شما اجازه می‌دهد عملگرها را روی نمونه‌های تهی پذیر اعمال کنید. کد زیر نمونه‌هایی از این را نشان می‌دهد:

```
private static void Operators() {
    Int32? a = 5;
    Int32? b = null;

    // Unary operators (+ ++ - -- ! ~)
    a++; // a = 6
    b = -b; // b = null

    // Binary operators (+ - * / % & | ^ << >>)
    a = a + 3; // a = 9
    b = b * 3; // b = null;

    // Equality operators (== !=)
    if (a == null) { /* no */ } else { /* yes */ }
    if (b == null) { /* yes */ } else { /* no */ }
    if (a != b) { /* yes */ } else { /* no */ }

    // Comparison operators (<> <= >=)
    if (a < b) { /* no */ } else { /* yes */ }
}
```

سی‌شارپ این عملگرها را اینگونه تفسیر می‌کند:

- **عملگرهای یکانی (+, ++, -, --, !, ~)** اگر عملوند **null** باشد حاصل **null** است.
- **عملگرهای باینری (+, -, *, /, %, &, |, ^, <<, >>)** اگر هر یک از عملوندها **null** باشد حاصل **null** است. هر چند، یک اکسپشن تولید می‌شود وقتی **&** و **|** روی عملوندهای **Boolean?** عمل می‌کنند تا رفتار این دو عملگر همان رفتاری باشد که توسط منطق سه مقداری

SQL ارائه می‌شود. برای این دو عملگر، اگر هیچ کدام از عملوندها **null** نباشند، عملگر طبق انتظار عمل می‌کند و اگر هر دو عملوند **null** باشند، حاصل **null** است. رفتار بخصوصی ایجاد می‌شود وقتی فقط یکی از عملوندها **null** باشد. جدول زیر نتایج تولید شده توسط این دو عملگر را برای تمام ترکیبهای **true**، **false** و **null** نشان می‌دهد:

عملگر ۱ ←	true	false	null
عملگر ۲ ↓	& = true	& = false	& = null
	= true	= true	= true
	& = false	& = false	& = false
	= true	= false	= null
	& = null	& = false	& = null
	= true	= null	= null

- **عملگرهای برابری (== و !=)** اگر هر دو عملوند **null** باشند، آن‌ها برابرند. اگر یکی از عملوندها **null** باشد، آن‌ها برابر نیستند. اگر هیچ کدام از عملوندها **null** نباشند، مقادیر را مقایسه می‌کند که آیا برابرند.
- **عملگرهای رابطه‌ای (< و > و <= و >=)** اگر هر کدام از عملوندها **null** باشند، حاصل **false** است. اگر هیچ کدام از عملوندها **null** نباشند مقادیر را مقایسه می‌کند.

شما باید آگاه باشید که دستکاری نمونه‌های تهی پذیر کد زیادی تولید می‌کند. برای مثال، متد زیر را ببینید:

```
private static Int32? NullableCodeSize(Int32? a, Int32? b) {
    return a + b;
}
```

وقتی من این متد را کامپایل می‌کنم، کد IL زیادی تولید می‌شود که عمل روی نوع‌های تهی پذیر را از انجام همان عمل روی نوع‌های غیرتهی پذیر کندتر می‌کند. معادل سی‌شارپ برای کد IL تولید شده توسط کامپایلر اینست:

```
private static Nullable<Int32> NullableCodeSize(Nullable<Int32> a, Nullable<Int32> b) {
    Nullable<Int32> nullable1 = a;
    Nullable<Int32> nullable2 = b;
    if (!(nullable1.HasValue & nullable2.HasValue)) {
        return new Nullable<Int32>();
    }
    return new Nullable<Int32>(nullable1.GetValueOrDefault() +
        nullable2.GetValueOrDefault());
}
```

سرانجام، بگذارید اشاره کنم که شما می‌توانید نوع‌های مقداری خود را تعریف کنید که عملگرهای مختلفی که در بالا ذکر شدند را سربارگذاری می‌کنند. من چگونگی انجام این کار را در بخش "متدهای سربارگذاری عملگر" در فصل ۸ "متدها" بحث کردم. اگر شما سپس از یک نمونه تهی پذیر از نوع مقداری خودتان استفاده کنید، کامپایلر کار صحیح را انجام داده و عملگر سربارگذاری شده‌ی شما را فراخوانی می‌کند. برای مثال، تصور کنید شما یک نوع مقداری **Point** دارید که سربارگذاری‌هایی برای عملگرهای **==** و **!=** را اینگونه تعریف می‌کند:

```
using System;
```

```
internal struct Point {
    private Int32 m_x, m_y;
    public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }

    public static Boolean operator==(Point p1, Point p2) {
        return (p1.m_x == p2.m_x) && (p1.m_y == p2.m_y);
    }
}
```

```

    }

    public static Boolean operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }
}

```

در این لحظه، شما می‌توانید از نمونه‌های تهی پذیر از نوع **Point** استفاده کرده و کامپایلر عملگرهای سربارگذاری شده‌ی شما را فراخوانی خواهد کرد:

```

internal static class Program {
    public static void Main() {
        Point? p1 = new Point(1, 1);
        Point? p2 = new Point(2, 2);

        Console.WriteLine("Are points equal? " + (p1 == p2).ToString());
        Console.WriteLine("Are points not equal? " + (p1 != p2).ToString());
    }
}

```

وقتی من کد فوق را ساخته و اجرا می‌کنم، خروجی زیر را می‌گیرم:

```

Are points equal? False
Are points not equal? True

```

عملگر ترکیب گر تهی سی شارپ

سی‌شارپ یک عملگر به نام عملگر ترکیب گر تهی **null-coalescing operator** (**??**) دارد که دو عملوند می‌گیرد. اگر عملوند سمت چپ **null** نباشد، مقدار عملوند برگردانده می‌شود. اگر عملوند سمت چپ **null** باشد، مقدار عملوند راست برگردانده می‌شود. عملگر ترکیب گر تهی یک روش خیلی آسان برای تنظیم مقدار پیش فرض یک متغیر فراهم می‌کند.

یک ویژگی عالی از عملگر ترکیب گر تهی اینست که آن می‌تواند با نوع‌های ارجاعی و همچنین نوع‌های مقداری تهی پذیر استفاده شود. کد زیر استفاده از عملگر ترکیب گر تهی را نشان می‌دهد:

```

private static void NullCoalescingOperator() {
    Int32? b = null;

    // The line below is equivalent to:
    // x = (b.HasValue) ? b.Value : 123
    Int32 x = b ?? 123;
    Console.WriteLine(x); // "123"

    // The line below is equivalent to:
    // String temp = GetFilename();
    // filename = (temp != null) ? temp : "Untitled";
    String filename = GetFilename() ?? "Untitled";
}

```

برخی افراد بحث می‌کنند که عملگر ترکیب گر تهی فقط یک شکر نحوی (syntactical sugar) برای عملگر **?:** است و اینکه تیم کامپایلر سی‌شارپ نمی‌بایست این عملگر را به زبان اضافه کند. هر چند، عملگر ترکیب گر تهی دو بهبود نحوی مهم ارائه می‌کند. اول اینکه عملگر **??** با عبارت‌ها بهتر عمل می‌کند:

```

Func<String> f = () => SomeMethod() ?? "Untitled";

```

این کد از کد زیر که نیاز به انتساب متغیر و چندین عبارت دارد ساده تر خوانده و درک می‌شود:

```

Func<String> f = () => { var temp = SomeMethod();

```

```
return temp != null ? temp : "Untitled";};
```

بهبود دوم اینست که ?? در سناریوهای ترکیبی بهتر عمل می‌کند، برای مثال تک خط زیر:

```
String s = SomeMethod1() ?? SomeMethod2() ?? "Untitled";
```

از تکه کد زیر بسیار آسانتر خوانده و درک می‌شود:

```
String s;
var sm1 = SomeMethod1();
if (sm1 != null) s = sm1;
else {
    var sm2 = SomeMethod2();
    if (sm2 != null) s = sm2;
    else s = "Untitled";
}
```

CLR پشتیبانی ویژه برای نوع‌های مقداری تهی پذیر دارد

CLR برای نوع‌های مقداری تهی پذیر پشتیبانی درونی دارد. این پشتیبانی ویژه برای بسته‌بندی، باز کردن، فراخوانی **GetType** و فراخوانی متدهای رابط است و بدین علت به نوع‌های تهی پذیر داده شده است که آن‌ها راحت تر در CLR جای بگیرند. این همچنین باعث می‌شود آن‌ها عادی تر رفتار کنند همانند آنچه اغلب برنامه‌نویسان انتظار دارند. بگذارید نگاهی دقیق تر به پشتیبانی ویژه CLR برای نوع‌های تهی پذیر بیاندازیم.

بسته بندی نوع‌های مقداری تهی پذیر

یک متغیر **Nullable<Int32>** را تصور کنید که به صورت منطقی به **null** تنظیم شده است. اگر این متغیر به یک متد که انتظار یک **Object** را دارد ارسال شود، متغیر باید بسته‌بندی گردد و یک ارجاع به **Nullable<Int32>** بسته‌بندی شده به متد ارسال می‌شود. این ایده آل نیست چون به متد اکنون یک مقدار غیر **null** ارسال شده است اگرچه متغیر **Nullable<Int32>** منطقی حاوی مقدار **null** است. برای حل این، CLR هنگام بسته‌بندی یک متغیر تهی پذیر کد خاصی را اجرا می‌کند تا این تصور که نوع‌های تهی پذیر شهروندان درجه یک در محیط هستند را حفظ کند.

مخصوصاً، وقتی CLR یک نمونه **Nullable<T>** را بسته‌بندی می‌کند بررسی می‌کند ببیند آیا آن **null** است و اگر هست، CLR در حقیقت چیزی را بسته‌بندی نمی‌کند و **null** برگردانده می‌شود. اگر نمونه تهی پذیر **null** نیست، CLR مقدار را از نمونه تهی پذیر بدست آورده و آن را بسته‌بندی می‌کند. به بیان دیگر، یک **Nullable<Int32>** با یک مقدار **5** به یک **Int32** بسته‌بندی شده با یک مقدار **5** تبدیل می‌شود.

کد زیر این رفتار را نشان می‌دهد:

```
// Boxing Nullable<T> is null or boxed T
Int32? n = null;
Object o = n; // o is null
Console.WriteLine("o is null={0}", o == null); // "True"

n = 5;
o = n; // o refers to a boxed Int32
Console.WriteLine("o's type={0}", o.GetType()); // "System.Int32"
```

باز کردن نوع‌های مقداری تهی پذیر

CLR اجازه می‌دهد یک نوع مقداری بسته‌بندی شده **T** به یک **T** یا یک **Nullable<T>** باز شود. اگر ارجاع به نوع مقداری بسته‌بندی شده **null** باشد و شما آن را به یک **Nullable<T>** باز می‌کنید، CLR مقدار **Nullable<T>** را **null** می‌کند. کد زیر این رفتار را نشان می‌دهد:

```
// Create a boxed Int32
Object o = 5;

// Unbox it into a Nullable<Int32> and into an Int32
Int32? a = (Int32?) o; // a = 5
```



```

Int32 b = (Int32) o; // b = 5

// Create a reference initialized to null
o = null;

// "Unbox" it into a Nullable<Int32> and into an Int32
a = (Int32?) o; // a = null
b = (Int32) o; // NullReferenceException

```

فراخوانی GetType از طریق یک نوع مقداری تهی پذیر

هنگام فراخوانی GetType روی یک شیء `Nullable<T>`، CLR در حقیقت دروغ گفته و به جای نوع `Nullable<T>`، نوع `T` را برمی‌گرداند. کد زیر این رفتار را نشان می‌دهد:

```

Int32? x = 5;

// The line below displays "System.Int32"; not "System.Nullable<Int32>"
Console.WriteLine(x.GetType());

```

فراخوانی متدهای رابط از طریق یک نوع مقداری تهی پذیر

در کد زیر، من `n`، یک `Nullable<Int32>` را به `IComparable<Int32>`، یک نوع رابط، تبدیل (cast) می‌کنم. هرچند، نوع `Nullable<T>`، رابط `IComparable<Int32>` را پیاده‌سازی نمی‌کند در حالیکه `Int32` می‌کند. کامپایلر سی‌شارپ اجازه می‌دهد به هر حال این کد کامپایل شود و بررسی‌کننده‌ی CLR این کد را قابل بررسی در نظر می‌گیرد تا برای شما نحو ساده‌تری فراهم کند.

```

Int32? n = 5;
Int32 result = ((IComparable) n).CompareTo(5); // Compiles & runs OK
Console.WriteLine(result); // 0

```

اگر CLR این پشتیبانی ویژه را فراهم نمی‌کرد، برای شما نوشتن کدی که یک متد رابط را بر روی یک نوع مقداری تهی پذیر فراخوانی کند، بسیار زحمت‌آور بود. شما مجبور بودید قبل از تبدیل به یک رابط، نوع مقداری باز شده را تبدیل کنید تا بتوانید فراخوانی را انجام دهید:

```

Int32 result = ((IComparable) (Int32) n).CompareTo(5); // Cumbersome

```

فصل ۲۰: اکسپشن ها و مدیریت وضعیت

این فصل تماما درباره‌ی مدیریت خطاست. اما فقط در مورد آن نیست، چندین بخش برای مدیریت خطا وجود دارد. اول، ما تعریف می‌کنیم یک خطا در حقیقت چیست. سپس، بحث می‌کنیم چگونه شما کشف کنید چه موقع کدتان با یک خطا مواجه می‌شود و چگونه برنامه را از این خطا احیا کنید. در این لحظه، وضعیت به یک مسئله تبدیل می‌شود چون خطاها تمایل دارند در زمان‌های نامناسبی رخ دهند. این احتمال وجود دارد که کد شما در وسط یک تغییر وضعیت باشد و با خطایی مواجه شود، و کد شما احتمالا مجبور است به چند وضعیت عقب تا قبل از آنکه آن را تغییر دهد، برگردد. البته، ما همچنین بحث می‌کنیم چگونه کد شما، به فراخوانی کننده‌هایش اطلاع دهد که یک خطا را یافت کرده است.

به نظر من، مدیریت اکسپشن ضعیف ترین بخش اجرایی زبان مشترک (CLR) است و بنابراین باعث مشکلات فراوانی برای برنامه‌نویسان در نوشتن کد مدیریت شده می‌شود. در طول سال‌ها، میکروسافت بهبودهای مهمی انجام داده تا به برنامه‌نویسان کمک کند با خطا روبرو شوند اما من اعتقاد دارم هنوز کارهای زیادی مانده تا واقعا ما یک سیستم خوب و قابل اطمینان داشته باشیم. من درباره‌ی این بهبودها که برای کار با اکسپشن‌های مدیریت نشده، نواحی اجرایی محدود شده، قراردادهای کد، اکسپشن‌های پوشانده شده‌ی زمان اجرا، اکسپشن‌های غیر قابل گرفتن و غیره فراهم شده اند، بسیار بحث می‌کنم.

تعریف "اکسپشن"

هنگام طراحی یک نوع، شما ابتدا وضعیت‌های مختلف که نوع در آن‌ها استفاده می‌شود را تصور می‌کنید. نام نوع معمولا یک اسم است مثل **FileStream** یا **StringBuilder**. سپس شما ویژگی‌ها، متدها، رویدادها و غیره را برای نوع تعریف می‌کنید. روشی که شما این اعضا را تعریف می‌کنید (نوع‌های داده‌ای و ویژگی، پارامترهای متد، مقادیر برگشتی و غیره) تبدیل واسط برنامه‌نویسی برای نوع شما می‌شوند. این اعضا عمل‌هایی که می‌تواند توسط خود نوع یا روی یک نمونه از نوع انجام شود را تعیین می‌کنند. این اعضای عملیاتی معمولا فعل‌هایی مثل **Read, Write, Flush, Append, Insert, Remove** و غیره است. وقتی یک عضو عملیاتی نتواند وظیفه‌اش را انجام دهد، عضو باید یک اکسپشن^{۵۷} تولید کند.^{۵۸}

مهم وقتی که یک عضو نتواند وظیفه‌ای که از روی نامش بر می‌آید را کامل کند، این یک اکسپشن است.

به تعریف کلاس زیر نگاه کنید:

```
internal sealed class Account {
    public static void Transfer(Account from, Account to, Decimal amount) {
        from -= amount;
        to += amount;
    }
}
```

متد **Transfer** دو شی **Account** و یک مقدار **Decimal** که مقدار پولی که باید بین حساب‌ها انتقال یابد را معین می‌کند، دریافت می‌نماید. واضح است که هدف متد **Transfer**، کسر پول از یک حساب و افزودن پول به حساب دیگر است. متد **Transfer** می‌تواند به دلایل مختلف شکست بخورد: آرگومان **from** یا **to** **null** باشند، آرگومان **from** یا **to** شاید به یک حساب باز اشاره نکنند، حساب **from** شاید پول کافی نداشته باشد، حساب **to** شاید پول بسیار زیادی داشته باشد که افزودن به آن باعث سرریز شود، یا آرگومان مقدار، شاید **0**، منفی یا بیش از دو رقم اعشار داشته باشد.

وقتی متد **Transfer** فراخوانی می‌شود، کد شی باید تمام این احتمالات را بررسی کند و اگر هر یک از آن‌ها یافت شد، آن نمی‌تواند پول را انتقال دهد و باید فراخوانی کننده را از شکست خود با تولید یک اکسپشن مطلع کند. در حقیقت توجه کنید که نوع برگشتی متد **Transfer**، **void** است. این بدین خاطر است که متد **Transfer** مقدار معناداری برای برگرداندن ندارد. اگر برگردد، پس آن موفق بوده است. اگر شکست بخورد یک اکسپشن معنادار تولید می‌کند.

برنامه‌نویسی شی گرا به برنامه‌نویسان اجازه می‌دهد که بسیار بهره‌ور باشند چون شما می‌توانید چنین کدی بنویسید:

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().Endswith("E"); // true
```

^{۵۷} Exception به معنی استثنا نیز معنی می‌شود.

^{۵۸} فعل اصلی رایج در برنامه‌نویسی **Throw** به معنای پرتاب کردن، برای اکسپشن‌ها استفاده می‌شود، معنی تحت الفظی عبارت اینست: "... یک اکسپشن پرتاب کند".

در اینجا من هدفم را با زنجیر کردن چندین عملیات به همدیگر تشکیل می‌دهم.^{۵۹} نوشتن این کد برای من و خواندن و نگهداری آن برای دیگران آسان است چون هدف مشخص است: یک رشته بگیر، بخشی از آن را جدا کن، آن بخش را به حروف بزرگ تبدیل کن، و بین آیا با یک "E" خاتمه یافته است. این عالی است اما یک فرض بزرگ در اینجا در نظر گرفته شده: هیچ عملیاتی شکست نمی‌خورد. اما، البته، خطا همیشه ممکن است، پس ما به راهی برای مدیریت این خطاها نیاز داریم. در حقیقت، ساخت‌های شی‌گرای بسیاری - سازنده‌ها، خواندن/نوشتن یک ویژگی، افزودن/حذف یک رویداد، فراخوانی سربرگذاری یک عملگر، فراخوانی یک عملگر تبدیل - وجود دارند که راهی برای برگرداندن کدهای خطا ندارند، اما این ساخت‌ها هنوز هم باید بتوانند یک خطا را گزارش کنند. مکانیزم فراهم شده توسط دات‌نت فریمورک مایکروسافت و تمام زبان‌های برنامه‌نویسی که آن را پشتیبانی می‌کنند مدیریت اکسپشن exception handling است.

مهم بسیاری از برنامه‌نویسان به اشتباه باور دارند که یک اکسپشن به این ربط دارد که چیزی چند بار تکراری رخ دهد. برای مثال، یک برنامه‌نویس که یک متد **Read** برای فایل‌ها می‌نویسد، احتمالاً این را می‌گوید، "هنگام خواندن از یک فایل، شما سرانجام به انتهای داده‌ها می‌رسید. چون رسیدن به انتها همیشه رخ می‌دهد، من متد **Read** خود را طوری طراحی می‌کنم که رسیدن به انتها را با برگرداندن یک مقدار خاص گزارش کند. من یک اکسپشن تولید نمی‌کنم." مشکل این عبارت این است که توسط برنامه‌نویسی که متد **Read** را طراحی کرده، گفته شده است نه توسط برنامه‌نویسی که متد **Read** را فراخوانی می‌کند.

هنگام طراحی متد **Read**، این غیرممکن است که برنامه‌نویس تمام حالت‌هایی که متد فراخوانی می‌شود را بداند. بنابراین، برنامه‌نویس احتمالاً نخواهد دانست چقدر مکرر فراخوانی‌کننده‌ی متد **Read** سعی می‌کند پس از انتهای فایل را بخواند. در حقیقت، چون اغلب فایل‌ها حاوی داده ساخت یافته هستند، تلاش برای خواندن پس از انتهای فایل چیزیست که به ندرت رخ می‌دهد.

مکانیک مدیریت اکسپشن

در این بخش، من مکانیک و ساخت‌های سی‌شارپ که برای مدیریت اکسپشن نیاز است را معرفی می‌کنم اما هدفم این است که با جزئیات آن‌ها را مطرح کنم. هدف این فصل ارائه راهنمایی برای این است که شما چه موقع و چگونه از مدیریت اکسپشن در کدتان استفاده کنید. اگر اطلاعات بیشتری درباره‌ی مکانیک و ساخت‌های زبان برای استفاده از مدیریت اکسپشن می‌خواهید، مستندات دات‌نت فریمورک و مشخصات زبان سی‌شارپ را نگاه کنید. همچنین، مکانیزم مدیریت اکسپشن دات‌نت فریمورک با استفاده از مکانیزم مدیریت اکسپشن ساخت یافته **Structured Exception Handling (SEH)** که توسط ویندوز مایکروسافت ارائه می‌شود، ساخته شده است. **SEH** در منابع فراوانی بحث شده است، شامل کتاب خودم، **Windows via C/C++** و ویرایش پنجم (انتشارات مایکروسافت ۲۰۰۷) که سه فصل آن به **SEH** اختصاص یافته است.

کد سی‌شارپ زیر یک استفاده استاندارد از مکانیزم مدیریت خطا را نشان می‌دهد. این کد به شما ایده‌ای می‌دهد که بلوک‌های مدیریت اکسپشن شبیه چیستند و هدفشان چیست. در بخش‌های بعد از کد، من رسماً بلوک‌های **try**، **catch** و **finally** و اهدافشان را توضیح می‌دهم و درباره‌ی استفاده‌ی آن‌ها نکاتی را می‌گویم.

```
private void SomeMethod() {
    try {
        // Put code requiring graceful recovery and/or cleanup operations here...
    }
    catch (InvalidOperationException) {
        // Put code that recovers from an InvalidOperationException here...
    }
    catch (IOException) {
        // Put code that recovers from an IOException here...
    }
    catch {
        // Put code that recovers from any kind of exception other than those above here...
    }
}
```

^{۵۹} در حقیقت، ویژگی متدهای گسترشی سی‌شارپ برای این در زبان هست که به شما اجازه دهد چندین متد را به هم زنجیر کنید که از طریق دیگر غیرممکن است.

```

// when catching any exception, you usually re-throw the exception.
// I explain re-throwing later in this chapter.
throw;
}
finally {
// Put code that cleans up any operations started within the try block here...
// The code in here ALWAYS executes, regardless of whether an exception is thrown.
}
// Code below the finally block executes if no exception is thrown within the try
// block or if a catch block catches the exception and doesn't throw or re-throw an
// exception.
}

```

این کد یک روش ممکن برای استفاده از بلوک‌های مدیریت اکسپشن را نشان می‌دهد. نگذارید کد شما را بترساند – اغلب متدها یک بلوک **try** ساده با یک تک بلوک **finally** یا یک بلوک **try** با یک تک بلوک **catch** دارند. غیرمعمول است که این تعداد بلوک **catch** داشته باشند. من آن‌ها را فقط برای نمایش گذاشته‌ام.

بلوک try

یک بلوک **try** حاوی کدی است که نیاز به عملیات‌های پاک‌سازی، عملیات‌های احیای اکسپشن^{6۰} یا هر دو را دارد. کد پاک‌سازی باید در یک تک بلوک **finally** قرار بگیرد. یک بلوک **try** می‌تواند همچنین شامل کدی باشد که احتمالاً یک اکسپشن تولید می‌کند. کد احیای اکسپشن باید در یک یا بیشتر بلوک **catch** قرار بگیرد. شما یک بلوک **catch** به ازای هر گونه از اکسپشن که برنامه تان بتواند به صورت امن از آن احیا شود، درست می‌کنید. یک بلوک **try** باید با حداقل یک بلوک **catch** یا **finally** همراه شود. معنی ندارد که یک بلوک **try** به تنهایی داشته باشید، و سی‌شارپ مانع از این کار می‌شود.

مهم گاهی برنامه‌نویسان می‌پرسند چه مقدار کد باید درون یک بلوک **try** جای بگیرد. پاسخ به این سوال به مدیریت وضعیت بستگی دارد. اگر، درون یک بلوک **try**، شما چندین عملیات را اجرا می‌کنید که می‌توانند منجر به تولید یک نوع اکسپشن شوند و روشی که شما از این نوع اکسپشن، احیا می‌شوید بسته به عملیات، متفاوت است، آنگاه شما باید هر عملیات را در بلوک **try** مجزای خودش جای دهید تا بتوانید از وضعیت به درستی احیا شوید.

بلوک catch

یک بلوک **catch** حاوی کدی است که باید در پاسخ به یک اکسپشن اجرا شود. یک بلوک **try** می‌تواند صفر یا بیشتر بلوک **catch** همراهش باشد. اگر کد درون یک بلوک **try** منجر به تولید یک اکسپشن نشود، **CLR** هرگز کد درون هر کدام از بلوک‌های **catch** آن را اجرا نمی‌کند. ترد به سادگی از روی بلوک‌های **catch** رد می‌شود و کد درون بلوک **finally** را (در صورت وجود) اجرا می‌کند. پس از آنکه کد درون بلوک **finally** اجرا شد، اجرا با عبارت بعد از بلوک **finally** ادامه می‌یابد.

آنچه بعد از کلمه کلیدی **catch** درون پرانتز قرار گرفته، نوع "گرفتن" **catch type** نامیده می‌شود. در سی‌شارپ، شما باید یک نوع گرفتن **System.Exception** یا یک نوع مشتق شده از **System.Exception** را تعیین کنید. برای نمونه، کد قبلی حاوی بلوک‌های **catch** برای مدیریت یک **InvalidOperationException** (یا هر اکسپشن مشتق شده از آن) و یک **IOException** (یا هر اکسپشن مشتق شده از آن) است. آخرین بلوک **catch** (که یک نوع گرفتن تعیین نمی‌کند) هر اکسپشنی که از نوع اکسپشن‌های تعیین شده توسط بلوک‌های **catch** قبلی را مدیریت می‌کند، این معادل آنست که یک بلوک **catch** داشته باشید که یک نوع گرفتن **System.Exception** تعیین کند، با این فرق که شما نمی‌توانید به اطلاعات اکسپشن از طریق کد درون آکولات‌های بلوک **catch** دسترسی پیدا کنید.

⁶⁰ Exception Recovery

نکته هنگام خطایابی یک بلوک **catch** با استفاده از ویژوال استودیو مایکروسافت، شما می‌توانید شی اکسپشن تولید شده‌ی کنونی را با افزودن متغیر ویژه‌ی **\$exception** به یک پنجره‌ی **watch**، نگاه کنید.

CLR از بالا به پایین به دنبال یک نوع **catch** مطابق می‌گردد و بنابراین شما باید نوع‌های اکسپشن خاص تر را در بالا قرار دهید. نوع بیشتر مشتق شده ابتدا باید قرار گیرد و به دنبال آن نوع‌های پایه‌اش (اگر باشند)، و به سمت **System.Exception** (یا هر بلوک اکسپشن که یک نوع گرفتن تعیین نکند)، پایین می‌آییم. در حقیقت، کامپایلر سی‌شارپ اگر شما بلوک‌های خاص تر **catch** را نزدیک تر به پایین، بگذارید اعلام خطا می‌کند چون بلوک **catch** غیرقابل دستیابی خواهد شد.

اگر یک اکسپشن توسط کدی درون بلوک **try** (یا هر متدی که از درون بلوک **try** فراخوانی شده است) تولید شود، CLR به دنبال بلوک‌های **catch** می‌گردد که نوع گرفتن آن‌ها از همان نوع یا یک نوع پایه‌ی اکسپشن تولید شده باشد. اگر هیچ نوع گرفتگی با نوع اکسپشن مطابق نباشد، CLR پشت‌پشتی فراخوانی (call stack) را برای یافتن یک نوع گرفتن که با اکسپشن مطابق باشد، جستجو می‌کند. اگر پس از رسیدن به بالای پشت‌پشتی فراخوانی، هیچ بلوک **catch** ای با یک نوع گرفتن مطابق، یافت نشد، یک اکسپشن مدیریت نشده تولید می‌شود. من بعداً در این فصل بیشتر درباره‌ی اکسپشن مدیریت نشده صحبت می‌کنم.

به محضی که CLR یک بلوک **catch** با یک نوع مطابق را می‌یابد، کد درون تمام بلوک‌های **finally** درونی را با شروع از بلوک **try** ای که کدش، اکسپشن تولید کرده و توقف در بلوک **catch** که با اکسپشن مطابق است، اجرا می‌کند. توجه کنید هر بلوک **finally** که با بلوک **catch** ای که با اکسپشن مطابق شده، همراه است، هنوز اجرا نشده است. کد درون این بلوک **finally** تا زمانی که کد درون بلوک **catch** مدیریت کننده، اجرا نشود، اجرا نمی‌گردد.

پس از آنکه تمام بلوک‌های **finally** درونی اجرا شدند، کد درون بلوک **catch** مدیریت کننده اجرا می‌شود. این کد نوعاً عملیات‌هایی برای مواجه شدن با اکسپشن انجام می‌دهد. پس از پایان بلوک **catch**، شما سه انتخاب دارید:

- همان اکسپشن را تولید کنید تا کدی که در پشت‌پشتی فراخوانی بالاتر قرار دارد، با خبر شود.
- یک اکسپشن متفاوت تولید کنید تا به کد بالاتر در پشت‌پشتی فراخوانی، اطلاعات اکسپشن غنی‌تری بدهید.
- بگذارید ترد از انتهای بلوک **catch** عبور کند.

بعداً در این فصل، من راهنمایی‌هایی را برای اینکه چه وقت از هر کدام از این تکنیک‌ها استفاده کنید، ارائه می‌کنم. اگر شما هر یک از دو تکنیک اول را انتخاب کنید، شما دارید یک اکسپشن تولید می‌کنید و CLR همانند گذشته عمل می‌کند؛ در پشت‌پشتی فراخوانی بالا می‌رود و به دنبال یک بلوک **catch** که نوعش مطابق نوع اکسپشن تولید شده است می‌گردد.

اگر شما آخرین تکنیک را انتخاب کنید، وقتی ترد از انتهای بلوک **catch** خارج می‌شود، بلافاصله شروع به اجرای کد درون بلوک **finally** (در صورت وجود) می‌کند. بعد از آنکه همه‌ی کد درون بلوک **finally** اجرا شد، ترد از بلوک **finally** خارج شده و شروع به اجرای عبارت‌های بلافاصله بعد از بلوک **finally** می‌کند. اگر هیچ بلوک **finally** موجود نباشد، ترد اجرا را با عبارتی که بلافاصله بعد از آخرین بلوک **catch** آمده است ادامه می‌دهد.

در سی‌شارپ، شما می‌توانید نام یک متغیر را بعد از یک نوع گرفتن تعیین کنید. وقتی یک اکسپشن گرفته می‌شود^{۶۱}، این متغیر به شی مشتق شده از **System.Exception** که تولید شده است اشاره می‌کند. کد بلوک **catch** می‌تواند این متغیر را ارجاع کند تا به اطلاعات مخصوص به اکسپشن (مثل ردیابی در پشت‌پشتی تا رسیدن به اکسپشن) دسترسی پیدا کنید. اگرچه این امکان وجود دارد که این شی را تغییر دهید، شما نباید این کار را بکنید و شی را فقط خواندنی در نظر بگیرید. من نوع **Exception** و آنچه می‌توانید با آن انجام دهید را بعداً در این فصل خواهیم گفت.

نکته کد شما می‌تواند با رویداد **FirstChanceException** از **AppDomain** خودش را ثبت کند تا به محض رخداد یک اکسپشن در یک **AppDomain** از آن مطلع گردد. این اطلاع رسانی قبل از آنکه CLR به دنبال بلوک‌های **catch** بگردد رخ می‌دهد. برای اطلاعات بیشتر درباره این رویداد، فصل ۲۲ "میزبانی CLR و AppDomain ها" را ببینید.

^{۶۱} "Exception is caught" این عبارت را هنگام خطایابی اکسپشن‌ها خواهید دید.

بلوک finally

یک بلوک finally حاوی کدی است که تضمین می‌شود اجرا گردد.^{۶۴} نوعاً، کد درون یک بلوک finally عملیات پاکسازی که توسط عملیات‌های انجام شده در بلوک try، مورد نیاز است را انجام می‌دهد.

برای نمونه، اگر شما یک فایل را در یک بلوک try باز کنید، کد بستن فایل را در یک بلوک finally قرار می‌دهید:

```
private void ReadData(String pathname) {
    FileStream fs = null;
    try {
        fs = new FileStream(pathname, FileMode.Open);
        // Process the data in the file...
    }
    catch (IOException) {
        // Put code that recovers from an IOException here...
    }
    finally {
        // Make sure that the file gets closed.
        if (fs != null) fs.Close();
    }
}
```

اگر کد درون بلوک try بدون تولید یک اکسپشن اجرا شود، فایل به صورت تضمینی بسته می‌شود. اگر کد درون بلوک try یک اکسپشن تولید کند، کد درون بلوک finally هنوز هم اجرا می‌شود و تضمین می‌شود که فایل بسته شود؛ بدون توجه به اینکه آیا اکسپشن گرفته شده است یا خیر. نادرست است که عبارت بستن فایل را بعد از بلوک finally قرار دهید، اگر یک اکسپشن تولید شود و گرفته نشود، عبارت اجرا نخواهد شد که باعث می‌شود فایل (تا جمع آوری زباله بعدی) باز باقی بماند.

یک بلوک try نیاز ندارد یک بلوک finally همراهش باشد، گاهی کد درون یک بلوک try به هیچ کد پاکسازی نیاز ندارد. هرچند، اگر شما یک بلوک finally دارید، آن باید پس از تمام بلوک‌های catch قرار بگیرد. یک بلوک try می‌تواند صفر یا یک بلوک finally همراهش باشد.

وقتی یک ترد به انتهای کد درون یک بلوک finally می‌رسد، ترد شروع به اجرای عبارت بلافاصله پس از عبارت finally می‌کند. به خاطر داشته باشید کد درون بلوک finally کد پاکسازی است. این کد تنها باید آنچه برای پاکسازی عملیاتی که در بلوک try آغاز شده، نیاز است را اجرا کند. کد درون بلوک‌های catch و finally باید بسیار کوتاه بوده و با احتمال فراوان بدون اینکه خودشان یک اکسپشن تولید کنند اجرا شوند.

همیشه ممکن است که کد احیای اکسپشن یا پاکسازی، شکست خورده و یک اکسپشن تولید کند. در حالیکه امکان آن وجود دارد، خیلی بعید است و اگر اتفاق افتد معمولاً یعنی مشکل بسیار بزرگی در جایی وجود دارد. به احتمال زیاد وضعیتی در جایی دچار خرابی شده است. اگر سهواً یک اکسپشن درون یک بلوک اکسپشن یا finally تولید شود، دنیا به پایان نرسیده – مکانیزم اکسپشن CLR، اجرا خواهد شد، گویا اکسپشن بعد از بلوک finally رخ داده است. هرچند CLR اطلاعات اولین اکسپشن که در بلوک try متناظر (اگر باشد) را نگه نمی‌دارد و تمام اطلاعات (مثل ردیابی در پشته) را برای اولین اکسپشن از دست می‌دهد. احتمالاً (و امیدوارانه) این اکسپشن جدید توسط کد شما مدیریت نخواهد شد و اکسپشن تبدیل به یک اکسپشن مدیریت نشده خواهد شد. آنگاه CLR پردازش شما را از بین می‌برد، که خوب است چون وضعیت خراب، اکنون از بین می‌رود. این بسیار بهتر از آن است که برنامه شما با پاسخ‌های غیرقابل پیش‌بینی و حفره‌های امنیتی احتمالی به اجرای خود ادامه دهد.

شخصاً، من فکر می‌کنم تیم سی‌شارپ می‌بایست کلمات کلیدی متفاوتی برای مکانیزم مدیریت اکسپشن انتخاب می‌کرد. آنچه برنامه‌نویسان می‌خواهند انجام دهند، سعی (try) در اجرا تکه کدی است. و سپس، اگر چیزی شکست بخورد، یا از شکست احیا شود یا با برگشت به چندین وضعیت عقب آن را جبران کند (compensate) و با گزارش شکست به فراخوانی کننده به کار خود ادامه دهد. برنامه‌نویسان همچنین می‌خواهند بدون توجه به آنچه رخ می‌-

^{۶۴} از بین بردن یک ترد یا تخلیه یک AppDomain باعث می‌شود CLR یک ThreadAbortException تولید کند که به بلوک finally اجازه اجرا می‌دهد.

اگر یک ترد از طریق تابع Win32 TerminateThread کشته شود، یا اگر پردازش از طریق تابع Win32 TerminateProcess یا متد FailFast از System.Environment کشته شود، آنگاه بلوک finally اجرا نمی‌شود. البته وقتی یک پردازش از بین می‌رود، ویندوز تمام منابعی که پردازش مصرف می‌کرده است را پاکسازی می‌کند.

دهد، پاکسازی (cleanup) تضمینی داشته باشند. کد سمت چپ آن چیزی است که شما باید بنویسید تا کامپایلر سی شارپ خوشحال شود، اما کد سمت راست روشی است که من ترجیح می‌دهم به آن فکر کنم:

```
void Method() {
    try{
        ...
    }
    catch (XxxException) {
        ...
    }
    catch (YyyException) {
        ...
    }
    catch {
        ...; throw;
    }
    finally {
        ...
    }
}
```

```
void Method() {
    try{
        ...
    }
    handle (XxxException) {
        ...
    }
    handle (YyyException) {
        ...
    }
    compensate {
        ...; throw;
    }
    cleanup {
        ...
    }
}
```

اکسپشن‌های CLS و غیر CLS

تمام زبان‌های برنامه‌نویسی برای CLR باید تولید (throwing) اشیاء مشتق شده از **Exception** را پشتیبانی کنند چون مشخصات مشترک زبان (CLS) این را اجبار می‌کند. هر چند، در حقیقت CLR به یک نمونه از هر نوعی اجازه‌ی تولید شدن می‌دهد و برخی زبان‌های برنامه‌نویسی اجازه‌ی تولید اشیاء غیر از اکسپشن سازگار با CLS، مثل **String**، **Int32** یا **DateTime** را می‌دهند. کامپایلر سی شارپ تنها اجازه‌ی تولید اشیاء مشتق شده از **Exception** را می‌دهد در حالی‌که کد نوشته شده در زبان دیگر اجازه‌ی تولید اشیاء مشتق شده از **Exception** و همچنین اشیایی که از **Exception** مشتق نشده‌اند را می‌دهد.

بسیاری از برنامه‌نویسان نمی‌دانند که CLR اجازه می‌دهد هر شی‌ای تولید شود (throw) تا یک اکسپشن را گزارش کند. اغلب برنامه‌نویسان باور دارند تنها اشیاء مشتق شده از **Exception** می‌توانند تولید شوند. تا قبل از نسخه 2.0 از CLR وقتی برنامه‌نویسان بلوک‌های **catch** برای گرفتن اکسپشن‌ها را می‌نوشتند، آن بلوک‌ها فقط اکسپشن‌های سازگار با CLS را می‌گرفتند. اگر یک متد سی شارپ، یک متد نوشته شده در زبان برنامه‌نویسی دیگری را فراخوانی می‌کرد و آن متد یک اکسپشن جز اکسپشن‌های سازگار با CLS تولید می‌کرد، کد سی شارپ، این اکسپشن را اصلاً نمی‌گرفت که منجر به آسیب پذیری امنیتی می‌شد.

در نسخه 2.0 از CLR، مایکروسافت یک کلاس جدید **RuntimeWrappedException** (تعریف شده در فضای نام **System.Runtime.CompilerServices**) را معرفی کرد. این کلاس از **Exception** مشتق شده است پس یک نوع

اکسپشن سازگار با CLS است. کلاس **RuntimeWrappedException** حاوی یک فیلد از نوع **Object** است (که توسط ویژگی فقط-خواندنی **WrappedException** از **RuntimeWrappedException** قابل دسترسی است). در نسخه 2.0 از CLR، وقتی یک اکسپشن غیر از اکسپشن‌های سازگار با CLS تولید می‌شود، CLR به صورت خودکار یک نمونه از کلاس **RuntimeWrappedException** ساخته و فیلد خصوصی‌اش را برای اشاره به شی‌ای که در حقیقت تولید شده (پرتاب شده) مقداردهی اولیه می‌کند. در عمل، CLR اکنون تمام اکسپشن‌های غیر سازگار با CLS را به اکسپشن‌های سازگار با CLS تبدیل کرده است. هر کدی که اکنون یک نوع **Exception** بگیرد، اکسپشن‌های غیرسازگار با CLS را نیز می‌گیرد که مشکل آسیب پذیری احتمالی را حل می‌کند.

اگرچه کامپایلر سی‌شارپ به برنامه‌نویسان اجازه می‌دهد فقط اشیاء مشتق شده از **Exception** را بگیرند، تا قبل از نسخه 2.0 از سی‌شارپ، کامپایلر سی‌شارپ به برنامه‌نویسان اجازه می‌داد اکسپشن‌های غیرسازگار با CLS را با استفاده از کدی شبیه به این بگیرند:

```
private void SomeMethod() {
    try {
        // Put code requiring graceful recovery and/or cleanup operations here...
    }
    catch (Exception e) {
        // Before C# 2.0, this block catches CLS-compliant exceptions only
        // Now, this block catches CLS- & non-CLS-compliant exceptions
        throw; // Re-throws whatever got caught
    }
    catch {
        // In all versions of C#, this block catches CLS- & non-CLS-compliant
        // exceptions
        throw; // Re-throws whatever got caught
    }
}
```

حال، بعضی برنامه‌نویسان می‌دانند که CLR از هر دوی اکسپشن‌های سازگار و غیرسازگار با CLS پشتیبانی می‌کند و این برنامه‌نویسان ممکن است دو بلوک **catch** (مثل بالا) بنویسند تا هر دو گونه‌ی اکسپشن‌ها را بگیرند. اگر کد فوق برای CLR 2.0 یا جدیدتر مجدداً کامپایل شوند، بلوک **catch** دوم هرگز اجرا نخواهد شد و کامپایلر سی‌شارپ این را با یک هشدار اعلام می‌کند:

“CS1058: A previous catch clause already catches all exceptions. All non-exceptions thrown will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException.”

دو راه برای برنامه‌نویسان وجود دارد که کدی از نسخه قبل از 2.0 دانتنت فریمورک را به نسخه‌های جدید بروزرسانی کنند. راه اول اینکه شما می‌توانید کد دو بلوک **catch** را در یک بلوک **catch** ادغام کرده و یکی از بلوک‌های **catch** را حذف کنید. این دیدگاه توصیه می‌شود و به جای آن شما می‌توانید به CLR بگویید که کد درون اسمبلی شما می‌خواهد با قوانین قدیمی بازی کند. یعنی، به CLR بگویید بلوک‌های **catch (Exception)** شما نباید یک نمونه از کلاس جدید **RuntimeWrappedException** را بگیرند. و به جای آن، CLR باید شی غیر سازگار با CLS را واپسچی کند و تنها اگر شما یک بلوک **catch** که هیچ نوعی تعیین نکرده است، دارید، کد شما را فراخوانی کند. شما به کامپایلر می‌گویید که رفتار قدیمی را می‌خواهید با اعمال یک نمونه از **RuntimeCompatibilityAttribute** بر اسمبلی خود، شبیه به این:

```
using System.Runtime.CompilerServices;
[assembly:RuntimeCompatibility(WrapNonExceptionThrows = false)]
```


نکته این صفت اثری در سطح اسمبلی دارد: راهی نیست که هر دو سبک اکسپشن پوشانده شده و واپیچی شده را با هم در یک اسمبلی داشته باشید. هنگام افزودن کد جدید (که انتظار دارد CLR اکسپشن‌ها را بپوشاند) به یک اسمبلی حاوی کد قدیمی (که CLR در آن اکسپشن‌ها را نمی‌پوشاند) مراقب باشید.

کلاس System.Exception

CLR اجازه می‌دهد یک نمونه از هر نوعی برای یک اکسپشن، تولید شود - از یک **Int32** گرفته تا یک **String** و فرای آن. هر چند، مایکروسافت تصمیم گرفته است تمام زبان‌های برنامه‌نویسی را مجبور نکند اکسپشن‌هایی از هر نوعی را تولید کنند و بگیرند، پس آن‌ها نوع **System.Exception** را تعریف کردند و اعلام کردند تمام زبان‌های برنامه‌نویسی سازگار با **CLS** باید قادر باشند اکسپشن‌هایی که نوعشان از این نوع مشتق شده است را تولید کرده و بگیرند. نوع‌های اکسپشن که از **System.Exception** مشتق شده‌اند، سازگار با **CLS** نامیده می‌شوند. سی‌شارپ و بسیاری زبان‌های برنامه‌نویسی دیگر اجازه می‌دهند کد شما تنها اکسپشن‌های سازگار با **CLS** را تولید کند. نوع **System.Exception** یک نوع بسیار ساده است که حاوی ویژگی‌هایی است که در جدول ۱-۲۰ توضیح داده شده اند. معمولاً، شما کدی به هیچ طریقی برای خواندن یا دسترسی به این ویژگی‌ها نمی‌نویسید. به جای آن، وقتی برنامه شما به خاطر یک اکسپشن مدیریت نشده از کار می‌افتد، شما به این ویژگی‌ها در دیباگر، یا در یک گزارش که در **Windows Application event log** نوشته شده، نگاه می‌کنید.

جدول ۱-۲۰ ویژگی‌های عمومی از نوع System.Exception

ویژگی	دسترسی	نوع	توضیح
Message	فقط-خواندنی	String	شامل متن مفیدی است که بیان می‌کند چرا اکسپشن تولید شده است. وقتی یک اکسپشن تولید شده، مدیریت نشده باشد، نوعا پیام در یک گزارش نوشته می‌شود. چون کاربران نهایی این پیام را نمی‌بینند، پیام باید تا قدر ممکن فنی باشد تا برنامه‌نویسانی که گزارش را می‌بینند بتوانند از اطلاعات پیام برای حل مشکل کد هنگام تولید نسخه‌ی جدید استفاده کنند.
Data	فقط-خواندنی	IDictionary	یک ارجاع به یک مجموعه از جفت‌های کلید-مقدار. معمولاً، کدی که اکسپشن تولید می‌کند قبل از تولید آن ورودی‌هایی به این مجموعه اضافه می‌کند، کدی که این اکسپشن را می‌گیرد می‌تواند ورودی‌ها را خوانده و از اطلاعات در فرآیند احیای اکسپشن استفاده کند.
Source	خواندنی-نوشتنی	String	حاوی نام اسمبلی‌ای است که اکسپشن را تولید کرده است.
StackTrace	فقط-خواندنی	String	حاوی نام‌ها و امضاهای متدهایی است که فراخوانی شده‌اند تا این اکسپشن تولید شده است. این ویژگی برای خطایابی بی ارزش است.
TargetSite	فقط-خواندنی	MethodBase	حاوی متدی است که اکسپشن را تولید کرده است.
HelpLink	فقط-خواندنی	String	حاوی یک URL (مثل <code>file:///C:\MyApp\Help.htm#MyExceptionHelp</code>) است که به کاربر کمک می‌کند اکسپشن را درک کند. به خاطر داشته باشید تجربه‌های امنیتی مانع از این می‌شوند که کاربران حتی بتوانند اکسپشن‌های مدیریت نشده را ببینند، بنابراین مگر آنکه شما سعی کنید اطلاعات را به برنامه‌نویس دیگری برسانید، این ویژگی به ندرت استفاده خواهد شد.
InnerException	فقط-خواندنی	Exception	اکسپشن قبلی را نشان می‌دهد اگر اکسپشن کنونی در هنگام مدیریت یک اکسپشن رخ داده باشد. این ویژگی فقط-خواندنی معمولاً null است. نوع Exception همچنین یک متد عمومی GetBaseException ارائه می‌کند که از لیست پیوندی اکسپشن‌های درونی گذر کرده و اکسپشن اصلی تولید شده را برمی‌گرداند.

من دوست دارم کمی درباره‌ی ویژگی فقط-خواندنی **StackTrace** از **System.Exception** بگویم. یک بلوک **catch** می‌تواند این ویژگی را برای ردیابی در پشته، بخواند که بیان می‌کند چه متدهایی فراخوانی شده‌اند که منجر به این اکسپشن شده است. وقتی شما سعی در یافتن علت یک اکسپشن دارید، این اطلاعات می‌تواند بسیار با ارزش باشند تا شما بتوانید کدتان را تصحیح کنید. وقتی شما به این ویژگی دسترسی پیدا می‌کنید، شما در واقع کد درون CLR را فراخوانی می‌کنید، ویژگی فقط به سادگی یک رشته برمی‌گرداند. وقتی شما یک شی جدید از یک نوع مشتق شده از **Exception** می‌سازید، ویژگی **StackTrace** به **null** مقداردهی اولیه می‌شود. اگر شما بخواهید ویژگی را بخوانید، شما ردیابی در پشته را بدست نخواهید آورد، بلکه شما یک **null** دریافت می‌کنید.

وقتی یک اکسپشن تولید می‌شود، CLR در درون، مکانی که دستور **throw** (پرتاب شدن، تولید شدن) اتفاق افتاده است را ضبط می‌کند. وقتی یک بلوک **catch**، اکسپشن را می‌پذیرد، CLR مکانی که اکسپشن گرفته شده است را ضبط می‌کند. اگر درون یک بلوک **catch**، شما به ویژگی **StackTrace** از شی اکسپشن تولید شده، دسترسی پیدا کنید، کدی که ویژگی را پیاده‌سازی می‌کند به درون CLR فراخوانی می‌کند که یک رشته که تمام متدها بین جاییکه اکسپشن تولید شده و فیلتری که اکسپشن را گرفته، شناسایی کند را می‌سازد.

مهم وقتی شما یک اکسپشن تولید می‌کنید، CLR نقطه شروع برای اکسپشن را مجدداً تنظیم می‌کند یعنی، CLR تنها مکانی که جدیدترین شی اکسپشن در آن تولید شده است را به خاطر می‌آورد.

کد زیر همان شی اکسپشن که گرفته شده است را تولید می‌کند و باعث می‌شود CLR نقطه شروع برای اکسپشن را مجدداً تنظیم کند:

```
private void SomeMethod() {
    try { ... }
    catch (Exception e) {
        ...
        throw e; // CLR thinks this is where exception originated.
                // FxCop reports this as an error
    }
}
```

برخلاف آن، اگر شما یک شی اکسپشن را با استفاده از خود کلمه کلیدی **throw** مجدداً تولید کنید، CLR نقطه شروع پشته را مجدداً تنظیم نمی‌کند. کد زیر همان شی اکسپشن گرفته شده را مجدداً تولید می‌کند که باعث می‌شود CLR نقطه شروع اکسپشن را مجدداً تنظیم نکند:

```
private void SomeMethod() {
    try { ... }
    catch (Exception e) {
        ...
        throw; // This has no effect on where the CLR thinks the exception
              // originated. FxCop does NOT report this as an error
    }
}
```

در حقیقت، تنها تفاوت بین این دو تکه کد اینست که CLR فکر می‌کند مکان اصلی که اکسپشن تولید شده است، کجاست. متأسفانه، وقتی شما یک اکسپشن را تولید یا مجدداً تولید می‌کنید، ویندوز نقطه شروع پشته را مجدداً تنظیم (ریست) می‌کند. پس اگر اکسپشن، مدیریت نشده باشد، موقعیت پشته که به **Windows Error Reporting** گزارش می‌شود موقعیت آخرین تولید یا تولید مجدد است، اگرچه CLR موقعیت پشته که اکسپشن اصلی تولید شده است را می‌داند. این مایه تأسف است چون این کار، خطایابی برنامه‌ای که شکست خورده است را بسیار مشکل تر می‌کند. بعضی برنامه‌نویسان این را بسیار غیر قابل تحمل می‌دانند و آن‌ها راهی متفاوت برای پیاده‌سازی کدشان انتخاب کرده‌اند تا مطمئن شوند که ردیابی پشته، موقعیت یک اکسپشن که در اصل تولید شده است به درستی نشان می‌دهد:

```
private void SomeMethod() {
    Boolean trySucceeds = false;
    try {
        ...
    }
}
```

```

        trySucceeds = true;
    }
    finally {
        if (!trySucceeds) { /* catch code goes in here */ }
    }
}

```

رشته برگشتی از ویژگی **StackTrace** شامل هیچ متدی نیست که در پشته‌ی فراخوانی، بالاتر از نقطه‌ای قرار گیرد که بلوک **catch** که شی اکسپشن را گرفته، قرار دارد. اگر شما ردیابی پشته کامل، از شروع ترد تا مدیریت کننده‌ی اکسپشن را بخواهید، شما می‌توانید از نوع **System.Diagnostics.StackTrace** استفاده کنید. این نوع چند ویژگی و متد تعریف می‌کند که به برنامه‌نویسان اجازه می‌دهد با استفاده از برنامه‌نویسی، یک ردیابی پشته و فریم‌هایی که ردیابی پشته را می‌سازند، دستکاری کنند.

شما می‌توانید با استفاده از سازنده‌های مختلف، یک شی **StackTrace** بسازید. بعضی سازنده‌ها فریم‌ها را از شروع ترد تا نقطه‌ای که شی **StackTrace** ساخته می‌شود را می‌سازند. دیگر سازنده‌ها فریم‌های شی **StackTrace** را با استفاده از یک شی مشتق شده از **Exception** که به عنوان آرگومان ارسال می‌شود، مقداردهی اولیه می‌کنند.

اگر CLR بتواند نمادهای خطایابی (که در فایل **.pdb** قرار دارد) را برای اسمبلی‌های شما پیدا کند رشته برگشتی از ویژگی **StackTrace** از **System.Exception** یا متد **ToString** از **System.Diagnostics.StackTrace** شامل مسیر فایل‌های سورس کد و شماره خطوط خواهد بود. این اطلاعات برای خطایابی بسیار مفیدند.

هرگاه شما یک ردیابی پشته بدست می‌آورید، شاید شما ببینید که بعضی متدها که واقعا در پشته فراخوانی هستند در رشته ردیابی پشته ظاهر نمی‌شوند. دو دلیل برای این وجود دارد. دلیل اول اینست که پشته در حقیقت یک رکورد از مکانی است که ترد باید به آن برگردد، نه جاییکه ترد از آنجا آمده است. دلیل دوم اینکه کامپایلر فقط-در-لحظه (JIT) می‌تواند متدها را خطی کند تا از بار اضافی فراخوانی و برگشت از یک متد مجزا، خودداری کند. بسیاری از کامپایلرها (شامل کامپایلر سی‌شارپ) یک سویچ خط فرمان **/debug** ارائه می‌کنند. وقتی این سویچ استفاده می‌شود، این کامپایلرها اطلاعات را در اسمبلی حاصل تعبیه می‌کنند تا به کامپایلر JIT بگویند هیچ یک از متدهای اسمبلی را خطی نکنند و ردیابی‌های پشته را برای برنامه‌نویسی که کد را خطایابی می‌کند، کامل تر و معنادار تر نماید.

نکته کامپایلر JIT صفت سفارشی **System.Diagnostics.DebuggableAttribute** را که بر اسمبلی اعمال شده، بررسی می‌کند. کامپایلر سی‌شارپ این صفت را به صورت خودکار اعمال می‌کند. اگر این صفت، پرچم **DisableOptimizations** آن، تعیین شده باشد، کامپایلر JIT متدهای اسمبلی را خطی نمی‌کند. استفاده از سویچ **/debug** کامپایلر سی‌شارپ، این پرچم را تنظیم می‌کند. با اعمال صفت سفارشی **System.Runtime.CompilerServices.MethodImplAttribute** بر یک متد، شما می‌توانید مانع از آن شوید که کامپایلر JIT متد را برای هر دو ساخت خطایابی و نهایی، خطی کند. تعریف متد زیر نشان می‌دهد چگونه می‌توان مانع از خطی شدن متد شد:

```

using System;
using System.Runtime.CompilerServices;

internal sealed class SomeType {

    [MethodImpl(MethodImplOptions.NoInlining)]
    public void SomeMethod() {
        ...
    }
}

```

کلاس های اکسپشن تعریف شده در FCL

کتابخانه کلاس فریمورک (FCL) چندین نوع اکسپشن تعریف می‌کند (که تمام آن‌ها سرانجام از **System.Exception** مشتق می‌شوند). سلسله مراتب زیر نوع‌های اکسپشن تعریف شده در اسمبلی **mscorlib.dll** را نشان می‌دهد. دیگر اسمبلی‌ها، نوع‌های اکسپشن بیشتری هم تعریف می‌کنند. (برنامه‌ای که برای بدست آوردن این سلسله مراتب نیاز است در فصل ۲۳ "بارگذاری اسمبلی و رفلکشن" نشان داده شده است.)

System.Exception

System.AggregateException

System.ApplicationException

System.Reflection.InvalidFilterCriteriaException

System.Reflection.TargetException

System.Reflection.TargetInvocationException

System.Reflection.TargetParameterCountException

System.Threading.WaitHandleCannotBeOpenedException

System.InvalidTimeZoneException

System.IO.IsolatedStorage.IsolatedStorageException

System.Runtime.CompilerServices.RuntimeWrappedException

System.SystemException

System.AccessViolationException

System.AppDomainUnloadedException

System.ArgumentException

System.ArgumentNullException

System.ArgumentOutOfRangeException

System.DuplicateWaitObjectException

System.Globalization.CultureNotFoundException

System.Text.DecoderFallbackException

System.Text.EncoderFallbackException

System.ArithmeticException

System.DivideByZeroException

System.NotFiniteNumberException

System.OverflowException

System.ArrayTypeMismatchException

System.BadImageFormatException

System.CannotUnloadAppDomainException

System.Collections.Generic.KeyNotFoundException

System.ContextMarshalException

System.DataMisalignedException

System.FormatException

System.Reflection.CustomAttributeFormatException

System.IndexOutOfRangeException

System.InsufficientExecutionStackException

System.InvalidCastException

System.InvalidOperationException

System.ObjectDisposedException

System.InvalidProgramException

System.IO.IOException

System.IO.DirectoryNotFoundException

System.IO.DriveNotFoundException

System.IO.EndOfStreamException
System.IO.FileLoadException
System.IO.FileNotFoundException
System.IO.PathTooLongException
System.MemberAccessException
System.FieldAccessException
System.MethodAccessException
System.MissingMemberException
 System.MissingFieldException
 System.MissingMethodException
System.MulticastNotSupportedException
System.NotImplementedException
System.NotSupportedException
 System.PlatformNotSupportedException
System.NullReferenceException
System.OperationCanceledException
 System.Threading.Tasks.TaskCanceledException
System.OutOfMemoryException
 System.InsufficientMemoryException
System.RankException
System.Reflection.AmbiguousMatchException
System.Reflection.ReflectionTypeLoadException
System.Resources.MissingManifestResourceException
System.Resources.MissingSatelliteAssemblyException
System.Runtime.InteropServices.ExternalException
 System.Runtime.InteropServices.COMException
 System.Runtime.InteropServices.SEHException
System.Runtime.InteropServices.InvalidComObjectException
System.Runtime.InteropServices.InvalidOleVariantTypeException
System.Runtime.InteropServices.MarshalDirectiveException
System.Runtime.InteropServices.SafeArrayRankMismatchException
System.Runtime.InteropServices.SafeArrayTypeMismatchException
System.Runtime.Remoting.RemotingException
 System.Runtime.Remoting.RemotingTimeoutException
System.Runtime.Remoting.ServerException
System.Runtime.Serialization.SerializationException
System.Security.Cryptography.CryptographicException
 System.Security.Cryptography.CryptographicUnexpectedOperationException
System.Security.HostProtectionException
System.Security.Policy.PolicyException
System.Security.Principal.IdentityNotMappedException
System.Security.SecurityException
System.Security.VerificationException
System.Security.XmlSyntaxException
System.StackOverflowException
System.Threading.AbandonedMutexException
System.Threading.SemaphoreFullException
System.Threading.SynchronizationLockException

```

System.Threading.ThreadAbortException
System.Threading.ThreadInterruptedException
System.Threading.ThreadStartException
System.Threading.ThreadStateException
System.TimeoutException
System.TypeInitializationException
System.TypeLoadException
    System.DllNotFoundException
    System.EntryPointNotFoundException
System.TypeUnloadedException
System.UnauthorizedAccessException
    System.Security.AccessControl.PrivilegeNotHeldException
System.Threading.BarrierPostPhaseException
System.Threading.LockRecursionException
System.Threading.Tasks.TaskSchedulerException
System.TimeZoneNotFoundException

```

ایده‌ی اصلی مایکروسافت این بود که **System.Exception** نوع پایه برای تمام اکسپشن‌ها باشد و دو نوع دیگر، **System.SystemException** و **ApplicationException** تنها دو نوعی باشند که بلافاصله از **Exception** مشتق می‌شوند. علاوه بر این، اکسپشن‌هایی که توسط CLR تولید می‌شوند مشتق شده از **SystemException** باشند و تمام اکسپشن‌های تولید شده توسط برنامه مشتق شده از **ApplicationException** باشند. به این روش، برنامه‌نویسان می‌توانند یک بلوک **catch** بنویسند که تمام اکسپشن‌های تولیدی توسط CLR یا تمام اکسپشن‌های تولیدی توسط برنامه را بگیرد. هر چند، همانگونه که می‌بینید، این قاعده بسیار خوب پیروی نشده است، برخی نوع‌های اکسپشن مستقیماً از **Exception** مشتق شده‌اند (**IsolatedStorageException**)، برخی اکسپشن‌های تولیدی توسط CLR از **ApplicationException** مشتق شده‌اند (**TargetInvocationException**) و برخی اکسپشن‌های تولیدی توسط برنامه از **SystemException** مشتق شده‌اند (**FormatException**). بنابراین بسیار شلوغ و بهم ریخته است و نتیجه اینست که نوع‌های **SystemException** و **ApplicationException** اصلاً معنایی ندارند. در این لحظه، مایکروسافت دوست دارد آن‌ها را از سلسله مراتب کلاس اکسپشن حذف کند اما نمی‌تواند چون این کار هر کدی که قبلاً این دو نوع را ارجاع می‌کرده، از کار خواهد انداخت.

تولید یک اکسپشن

هنگام پیاده‌سازی متدهای خود، وقتی یک متد نمی‌تواند وظیفه‌ای که نامش نشان می‌دهد را کامل انجام دهد، شما یک اکسپشن تولید می‌کنید. وقتی شما می‌خواهید یک اکسپشن تولید کنید، دو مسئله وجود دارد که شما واقعاً نیاز دارید به آن‌ها توجه کنید. اولین مسئله درباره‌ی اینست که تصمیم بگیرید چه نوع مشتق شده از **Exception** را قصد دارید تولید کنید. شما واقعاً می‌خواهید یک نوع که در اینجا معنی‌دار است انتخاب کنید. به کدی که در پشته فراخوانی بالاتر قرار دارد توجه کنید و چگونه آن کد می‌تواند تعیین کند که یک متد شکست خورده است تا کد احیای مناسبی را اجرا نماید. شما می‌توانید از یک نوع که قبلاً در **FCL** تعریف شده است استفاده کنید اما ممکن است آن چیزی که با معنای دقیق شما مطابق باشد در **FCL** وجود نداشته باشد. پس شما احتمالاً نیاز به تعریف نوع خودتان دارید، که سرانجام از **System.Exception** مشتق می‌شود.

اگر شما بخواهید یک سلسله مراتب نوع اکسپشن تعریف کنید، قویاً توصیه می‌شود که سلسله مراتب، کم عمق و عریض باشد که تا حد ممکن کلاس‌های پایه کمتری ساخته شود. علت اینست که کلاس‌های پایه به روشی عمل می‌کنند که با چندین خطا به عنوان یک خطا رفتار می‌کنند که این معمولاً خطرناک است. در طول این خطوط، شما هرگز نباید یک شی **System.Exception**^{۶۳} تولید (پرتاب) کنید، و اگر شما یک کلاس پایه نوع اکسپشن دیگر را تولید می‌کنید باید با احتیاط بسیار زیاد، از آن استفاده کنید.

^{۶۳} در حقیقت، کلاس **System.Exception** باید **abstract** علامت زده می‌شد که حتی مانع از کامپایل کدی شود که سعی در تولید آن دارد.

مهم انشعابات نسخه‌بندی در اینجا نیز وجود دارد. اگر شما یک نوع اکسپشن جدید مشتق شده از یک نوع اکسپشن موجود تعریف کنید، آنگاه تمام کدهایی که نوع پایه کنونی را می‌گیرند اکنون نوع جدید شما را نیز می‌گیرند. در برخی سناریوها، شاید این مطلوب باشد و در برخی سناریوها، شاید این مطلوب نباشد. مشکل اینست که این واقعا بستگی به این دارد کدی که کلاس پایه را می‌گیرد، چگونه به نوع اکسپشن و نوع‌های مشتق شده از آن پاسخ می‌دهد. کدی که هرگز اکسپشن جدید را پیش‌بینی نمی‌کرده، شاید اکنون غیرقابل پیش‌بینی رفتار کرده و حفره‌های امنیتی باز شود. فردی که نوع اکسپشن جدید را تعریف می‌کند نمی‌تواند درباره‌ی تمام مکان‌هایی که اکسپشن گرفته می‌شود و اینکه چگونه آن مدیریت می‌شود، بدانند. و بنابراین، در عمل، این غیرممکن است که یک تصمیم هوشمندانه خوب بگیرید.

مسئله دوم درباره‌ی اینست که تصمیم بگیرید چه رشته پیامی می‌خواهید به سازنده‌ی نوع اکسپشن ارسال کنید. وقتی شما یک اکسپشن تولید می‌کنید، شما باید یک رشته پیام با اطلاعات جزئی که نشان دهد چرا متد وظیفه‌اش را کامل انجام نداده، همراه آن کنید. اگر اکسپشن گرفته و مدیریت شده باشد، رشته پیام دیده نمی‌شود. هرچند، اگر اکسپشن به یک اکسپشن مدیریت نشده تبدیل شود، پیام معمولاً گزارش می‌شود. یک اکسپشن مدیریت نشده یک خطا را در برنامه نشان می‌دهد و برنامه‌نویس باید برای رفع خطا درگیر کد شود. یک کاربر نهایی، دسترسی به سورس کد یا توانایی رفع مشکل کد و کامپایل مجدد آن را ندارد. در حقیقت، رشته پیام نباید به یک کاربر نهایی نشان داده شود. پس این رشته‌های پیام می‌توانند بسیار جزئی باشند تا به برنامه‌نویسان در رفع مشکل کد کمک کنند.

علاوه بر این، چون تمام برنامه‌نویسان مجبورند انگلیسی صحبت کنند (حداقل تا حدی، چون زبان‌های برنامه‌نویسی و کلاس‌های FCL و متدها به زبان انگلیسی هستند)، معمولاً نیازی به محلی کردن رشته پیام‌های اکسپشن نیست. هر چند، شاید شما بخواهید رشته‌ها را محلی کنید اگر شما یک کتابخانه کلاس می‌سازید که توسط برنامه‌نویسانی که به زبانی متفاوت صحبت می‌کنند، استفاده می‌شود. مایکروسافت پیام‌های اکسپشن تولید شده توسط FCL را محلی می‌کند، چون برنامه‌نویسان سرتاسر دنیا از این کتابخانه کلاس استفاده می‌کنند.

تعریف کلاس اکسپشن خودتان

متأسفانه، طراحی نوع اکسپشن خودتان، خسته کننده بوده و با خطا همراه است. اصلی ترین علت اینست که تمام نوع‌های مشتق شده از **Exception** باید قابل سریالی شدن باشند تا بتوانند از محدوده یک AppDomain عبور کرده یا به یک گزارش یا پایگاه داده نوشته شوند. چندین مسئله مرتبط با سریالی کردن هست که در فصل ۲۴ "سریالی کردن زمان اجرا" بحث می‌شود. پس برای ساده کردن کارها، من کلاس جنریک **Exception<TExceptionArgs>** را درست کرده ام که اینگونه تعریف شده است:

```
[Serializable]
public sealed class Exception<TExceptionArgs> : Exception, ISerializable
    where TExceptionArgs : ExceptionArgs {

    private const String c_args = "Args"; // For (de)serialization
    private readonly TExceptionArgs m_args;

    public TExceptionArgs Args { get { return m_args; } }
    public Exception(String message = null, Exception innerException = null)
        : this(null, message, innerException) { }

    public Exception(TExceptionArgs args, String message = null,
        Exception innerException = null): base(message, innerException) { m_args = args; }

    // The constructor is for deserialization; since the class is sealed, the constructor
    // is private. If this class were not sealed, this constructor should be protected
    [SecurityPermission(SecurityAction.LinkDemand,
        Flags=SecurityPermissionFlag.SerializationFormatter)]
    private Exception(SerializationInfo info, StreamingContext context)
        : base(info, context) {
```

```

        m_args = (TExceptionArgs)info.GetValue(c_args, typeof(TExceptionArgs));
    }

    // The method for serialization; it's public because of the ISerializable interface
    [SecurityPermission(SecurityAction.LinkDemand,
        Flags=SecurityPermissionFlag.SerializationFormatter)]
    public override void GetObjectData(SerializationInfo info, StreamingContext context) {
        info.AddValue(c_args, m_args);
        base.GetObjectData(info, context);
    }

    public override String Message {
        get {
            String baseMsg = base.Message;
            return (m_args == null) ? baseMsg : baseMsg + " (" + m_args.Message + ")";
        }
    }

    public override Boolean Equals(Object obj) {
        Exception<TExceptionArgs> other = obj as Exception<TExceptionArgs>;
        if (obj == null) return false;
        return Object.Equals(m_args, other.m_args) && base.Equals(obj);
    }

    public override int GetHashCode() { return base.GetHashCode(); }
}

```

و کلاس پایه **ExceptionArgs** که **TExceptionArgs** به آن محدود شده، بسیار ساده و شبیه به این است:

```

[Serializable]
public abstract class ExceptionArgs {
    public virtual String Message { get { return String.Empty; } }
}

```

اکنون، با تعریف این دو کلاس، من می‌توانم کلاس‌های اکسپشن بیشتری تعریف کنم وقتی به آن‌ها نیاز دارم. برای تعریف یک نوع اکسپشن که بیان کند دیسک پر است، من به سادگی این کار را می‌کنم:

```

[Serializable]
public sealed class DiskFullExceptionArgs : ExceptionArgs {
    private readonly String m_diskpath; // private field set at construction time

    public DiskFullExceptionArgs(String diskpath) { m_diskpath = diskpath; }

    // Public read-only property that returns the field
    public String DiskPath { get { return m_diskpath; } }

    // Override the Message property to include our field (if set)
    public override String Message {
        get {
            return (m_diskpath == null) ? base.Message : "DiskPath=" + m_diskpath;
        }
    }
}

```



```

    }
}

```

و اگر من داده اضافی که بخواهم درون کلاس بگذارم نداشته باشم، به سادگی این کار را می‌کنم:

```
[Serializable]
```

```
public sealed class DiskFullExceptionArgs : ExceptionArgs { }
```

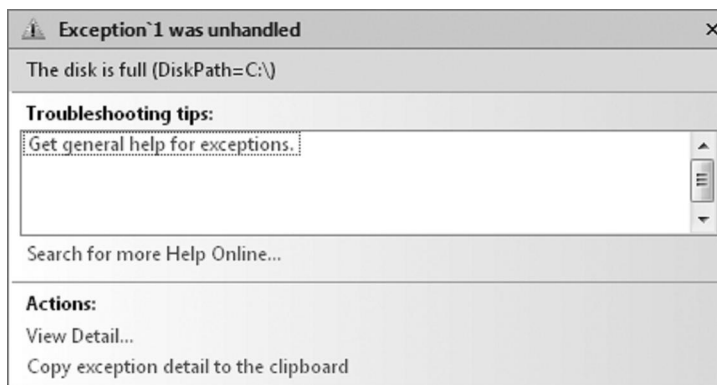
و اکنون من می‌توانم کدی بنویسم که یک از این‌ها را تولید کند و بگیرد:

```

public static void TextException() {
    try {
        throw new Exception<DiskFullExceptionArgs>(
            new DiskFullExceptionArgs(@"C:\\"), "The disk is full");
    }
    catch (Exception<DiskFullExceptionArgs> e) {
        Console.WriteLine(e.Message);
    }
}
}

```

نکته دو مسئله وجود دارد که درباره کلاس **Exception<TExceptionArgs>** خودم، اشاره می‌کنم. اولین مسئله اینست که هر نوع اکسپشن که شما تعریف می‌کنید همیشه از **System.Exception** مشتق می‌شود. در اغلب سناریوها، اصلاً این مسئله نیست و در واقع داشتن یک سلسله مراتب نوع اکسپشن کم عمق و عریض، ترجیح داده می‌شود. مسئله دوم اینست که پنجره‌ی دیالوگ اکسپشن مدیریت نشده در ویژوال استودیو، پارامتر نوع جنریک از نوع **Exception<T>** را نشان نمی‌دهد، همانگونه که در اینجا می‌بینید:



معامله‌ی قابلیت اطمینان در برابر بهره‌وری

من نوشتن نرم افزار را در سال ۱۹۷۵ شروع کردم. مقداری برنامه‌نویسی BASIC انجام دادم و وقتی به سخت افزار بیشتر علاقه مند شدم، به زبان اسمبلی سویچ کردم. در طول زمان، من به زبان برنامه‌نویسی C سویچ کردم چون به من اجازه می‌داد با سطح بالاتری از انتزاع به سخت افزار دسترسی پیدا کنم که برنامه‌نویسی مرا آسانتر می‌کرد. پیشینه‌ی من در نوشتن کدهای سیستم عامل و کدهای پلتفرم/کتابخانه است، بنابراین من همیشه سعی می‌کردم کدم تا حد ممکن کوچک و سریع باشد چون برنامه‌ها می‌توانند تنها به اندازه‌ی سیستم عامل یا کتابخانه‌ای که از آن استفاده می‌کنند، خوب باشند.

علاوه بر ساخت کدهای کوچک و سریع، من همیشه بر احیای خطا تمرکز می‌کردم. هنگام تخصیص حافظه (با استفاده از عملگر **new** از **C++** یا با فراخوانی **VirtualAlloc**، **HeapAlloc**، **malloc** و غیره)، من همیشه مقدار برگشتی را بررسی می‌کردم تا مطمئن شوم حافظه‌ای که درخواست کرده‌ام واقعا به من داده شده باشد. و اگر درخواست حافظه شکست می‌خورد، من همیشه یک مسیر کد جایگزین داشتم تا مطمئن شوم وضعیت بقیه برنامه

دست نخورده باقی می ماند و به هر یک از فراخوانی کننده های من اجازه می دهد که بدانند من شکست خورده ام تا کد فراخوانی کننده هم بتواند اصلاحاتی را داشته باشد.

به دلایلی که نمی توانم کاملا توضیح دهم، این توجه به جزئیات هنگام نوشتن کد برای دانت فریمورک وجود ندارد. دسترسی به وضعیت خارج از حافظه همیشه امکان دارد و من تقریباً هرگز کدی حاوی یک بلوک **catch** که از یک **OutOfMemoryException** احیا شود را نمی بینم. در واقع، من حتی برنامه نویسانی را دیده ام که به من می گویند CLR به یک برنامه اجازه نمی دهد یک **OutOfMemoryException** را بگیرد. این مطلقاً غلط است، شما می توانید این اکسپشن را بگیرید. در حقیقت خطاهای زیادی وجود دارد که هنگام اجرای کد مدیریت شده ممکن است رخ بدهد و من به سختی برنامه نویسانی را دیده ام که کدی بنویسند که سعی کند از این خطاهای احتمالی احیا شود. در این بخش، من می خواهم برخی شکست های احتمالی و اینکه چرا پذیرفته می شود آن ها را نادیده بگیریم را اشاره کنم. من همچنین می خواهم برخی مسائل مهم که هنگام نادیده گرفتن این شکست ها رخ می دهد و راه هایی برای کم کردن این مسائل را توضیح دهم.

برنامه نویسی شی گرا به برنامه نویسان اجازه می دهد بسیار بهره ور باشند. یک بخش مهم از این، قابلیت ترکیب است که نوشتن، خواندن و نگهداری کد را آسان می کند. برای مثال، این خط کد را در نظر بگیرید:

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().Endswith("E");
```

یک فرض بزرگ در اینجا وجود دارد: هیچ خطایی رخ نمی دهد، اما البته خطا همیشه ممکن است و بنابراین ما به راهی برای مدیریت این خطاها نیاز داریم. این تمام چیزی است که مکانیزم ها و ساخت های مدیریت اکسپشن درباره ی آن است و چرا در مقابل داشتن متدهایی که **true/false** برمی گردانند تا مانند توابع Win32 و COM، موفقیت/شکست را بیان کنند، ما به آن ها نیاز داریم.

علاوه بر قابلیت ترکیب کد، ما به خاطر ویژگی های عالی که توسط کامپایلرهایمان ارائه می شود نیز بهره ور هستیم. برای نمونه، کامپایلر به صورت ضمنی:

- هنگام فراخوانی یک متد، آرگومان های انتخابی را درج می کند.
- نمونه های نوع مقداری را بسته بندی می کند.
- پارامترهای آرایه را می سازد/مقداردهی اولیه می کند.
- به اعضای متغیرها و عبارت های **dynamic** متصل می کند.
- به متدهای گسترشی متصل می کند.
- به عملگرهای سربارگذاری متصل می کند/آن ها را فراخوانی می کند.
- اشیاء نماینده را می سازد.
- هنگام فراخوانی متدهای جنریک، تعریف یک متغیر و استفاده از عبارت های لامبدا، نوع ها را استنتاج می کند.
- کلاس های بستر برای عبارت های لامبدا و حرکت کننده در حلقه (iterator) را تعریف می کند/می سازد.
- نوع های ناشناس و نمونه های آن را تعریف می کند/می سازد/مقدار دهی اولیه می کند.
- کدی را بازنویسی می کند تا از پرس و جوهای یکپارچه زبان (LINQ) Language Integrated Queries، عبارت های پرس و جو و درخت های عبارت (پشتیبانی کند).

و CLR همه نوع کارهای عالی برای برنامه نویسان انجام می دهد تا زندگی ما را آسانتر کند. برای نمونه، CLR به صورت ضمنی:

- متدهای مجازی و متدهای رابط را فراخوانی می کند.
- اسمبلی ها را بارگذاری کرده و متدهایی که ممکن است اکسپشن های زیر را تولید کنند، JIT کامپایل می کند: **FileLoadException**، **FieldAccessException**، **InvalidImageException**، **BadImageFormatException**، **MethodAccessException**، **MissingFieldException**، **MissingMethodException** و **VerificationException**.
- بین مرزهای AppDomain هنگام دسترسی به یک شی از یک نوع مشتق شده از **MarshalByRefObject** که احتمالاً می تواند **AppDomainUnloadedException** تولید کند، انتقال انجام می دهد.
- هنگام عبور از مرز یک **AppDomain**، اشیاء را سریالی و غیرسریالی می کند.

- باعث می‌شود ترد(ها) یک **ThreadAbortException** تولید کنند وقتی **Thread.Abort** یا **AppDomain.Unload** فراخوانی می‌شود.
- پس از جمع آوری، قبل از آنکه حافظه اشیاء باز پس گرفته شود، متدهای **Finalize** را فراخوانی می‌کند.
- هنگام استفاده از نوع‌های جنریک، اشیاء نوع را در هیپ بارگذاری کننده، می‌سازد.
- سازنده‌ی استاتیک یک نوع را که احتمال تولید **TypeInitializationException** را دارد، فراخوانی می‌کند.
- اکسپشن‌های مختلفی تولید می‌کند، شامل **DividedByZeroException**، **OutOfMemoryException**، **TargetInvocationException**، **RuntimeWrappedException**، **NullReferenceException**، **OverflowException**، **NotFiniteNumberException**، **ArrayTypeMismatchException**، **RankException**، **InvalidCastException**، **IndexOutOfRangeException**، **DataMisalignedException** و دیگر اکسپشن‌ها.

و البته دانتنت فریمورک با یک کتابخانه کلاس عظیم ارائه می‌شود که شامل ده‌ها هزار نوع می‌باشد که هر نوع یک کاربرد رایج و قابل استفاده مجدد را کپسوله می‌کند. نوع‌هایی برای ساخت برنامه‌های فرم وب، وب سرویس، برنامه‌های گرافیکی، کار با امنیت، دستکاری تصاویر، تشخیص صدا، و غیره وجود دارد. هر کدام از این کدها می‌توانند یک اکسپشن تولید کنند تا شکست را نشان دهند و نسخه‌های آتی، نوع‌های اکسپشن جدید که از نوع‌های اکسپشن کنونی مشتق شده‌اند را معرفی می‌کنند و اکنون بلوک **catch** شما نوع‌های اکسپشنی را می‌گیرد که قبلاً اصلاً وجود نداشته‌اند.

تمام این‌ها – برنامه‌نویسی شی گرا، ویژگی‌های کامپایلر، ویژگی‌های CLR و کتابخانه کلاس عظیم – آن چیزی است که دانتنت فریمورک را به یک پلتفرم برنامه‌نویسی قدرتمند مبدل می‌سازد.^{۶۴} نظر من اینست که تمام این موارد، نقاط شکستی برای کد شما معرفی می‌کنند که شما کنترل اندکی روی آن دارید. تا زمانی که همه چیز عالی کار می‌کند، خوب است: ما به راحتی کد می‌نویسیم، کد به راحتی خوانده و نگهداری می‌شود. اما وقتی چیزی اشتباه می‌شود، تقریباً غیرممکن است کاملاً بفهمیم چه چیزی و چرا اشتباه شده است. یک مثال حرف من را بهتر نشان می‌دهد:

```
private static Object OneStatement(Stream stream, Char charToFind) {
    return (charToFind + ": " + stream.GetType() + String.Empty
        + (stream.Position + 512M)).Where(c=>c == charToFind).ToArray();
}
```

متد، حاوی فقط یک عبارت سی‌شارپ است اما این عبارت کار زیادی انجام می‌دهد. در حقیقت، کد زبان میانی (IL) که کامپایلر سی‌شارپ برای این متد تولید می‌کند در زیر آمده است. (من برخی خطوط را با ایتالیک **Bold** نشان داده‌ام که این خطوط نقاط شکست احتمالی به خاطر عملیات‌های ضمنی‌ای است که صورت می‌گیرد.)

```
.method private hidebySig static object OneStatement(
    class [mscorlib]System.IO.Stream stream, char charToFind) cil managed {
    .maxstack 5
    .locals init (
        [0] class Program/<>c__DisplayClass1 CS$<>8__locals2,
        [1] object[] CS$0$0000)
    L_0000: newobj instance void Program/<>c__DisplayClass1::.ctor()
    L_0005: stloc.0
    L_0006: ldloc.0
    L_0007: ldarg.1
    L_0008: stfld char Program/<>c__DisplayClass1::charToFind
    L_000d: ldc.i4.5
    L_000e: newarr object
    L_0013: stloc.1
    L_0014: ldloc.1
```

^{۶۴} من باید ویرایشگر ویژوال استودیو، پشتیبانی **IntelliSense**، پشتیبانی تکه کدهای کوچک (code snippet)، قالب‌ها، سیستم قابلیت توسعه، سیستم خطایابی و دیگر ابزارها که پلتفرم را قدرتمند می‌سازند اشاره کنم. اما من این‌ها را از بحث اصلی حذف کردم چون اثری بر رفتار کد در زمان اجرا ندارند.

```

L_0015: ldc.i4.0
L_0016: ldloc.0
L_0017: ldfld char Program/<>c__DisplayClass1::charToFind
L_001c: box char
L_0021: stelem.ref
L_0022: ldloc.1
L_0023: ldc.i4.1
L_0024: ldstr ": "
L_0029: stelem.ref
L_002a: ldloc.1
L_002b: ldc.i4.2
L_002c: ldarg.0
L_002d: callvirt instance class [mscorlib]System.Type
    [mscorlib]System.Object::GetType()
L_0032: stelem.ref
L_0033: ldloc.1
L_0034: ldc.i4.3
L_0035: ldsgfld string [mscorlib]System.String::Empty
L_003a: stelem.ref
L_003b: ldloc.1
L_003c: ldc.i4.4
L_003d: ldc.i4 0x200
L_0042: newobj instance void [mscorlib]System.Decimal::.ctor(int32)
L_0047: ldarg.0
L_0048: callvirt instance int64 [mscorlib]System.IO.Stream::get_Position()
L_004d: call valuetype [mscorlib]System.Decimal
    [mscorlib]System.Decimal::op_Implicit(int64)
L_0052: call valuetype [mscorlib]System.Decimal [mscorlib]System.Decimal::op_Addition
    (valuetype [mscorlib]System.Decimal, valuetype [mscorlib]System.Decimal)
L_0057: box [mscorlib]System.Decimal
L_005c: stelem.ref
L_005d: ldloc.1
L_005e: call string [mscorlib]System.String::Concat(object[])
L_0063: ldloc.0
L_0064: ldftn instance bool Program/<>c__DisplayClass1::<M>b__0(char)
L_006a: newobj instance
    void [mscorlib]System.Func`2<char, bool>::.ctor(object, native int)
L_006f: call class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>
    [System.Core]System.Linq.Enumerable::Where<char>(
        class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>,
        class [mscorlib]System.Func`2<!!0, bool>)
L_0074: call !!0[] [System.Core]System.Linq.Enumerable::ToArray<char>
    (class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>)
L_0079: ret
}

```

همانگونه که می‌توانید ببینید هنگام ساخت کلاس `<>__DisplayClass1` (یک نوع تولیدی توسط کامپایلر)، آرایه `Object[]`، نماینده `Func` و بسته-بندی `Char` و `Decimal` احتمال تولید یک `OutOfMemoryException` وجود دارد. وقتی `Concat`، `Where` و `ToArray` فراخوانی می-

شوند، حافظه در درون تخصیص می‌یابد. ساخت نمونه **Decimal** می‌تواند باعث شود سازنده نوع، فراخوانی شود که آن هم منجر به یک **TypeInitializationException** گردد.^{۶۵} و سپس، فراخوانی‌های ضمنی به متدهای عملگر **op_implicit** و عملگر **op_addition** از **Decimal** وجود دارد که می‌توانند هر کاری شامل تولید یک **OverflowException** انجام دهند.

خواندن ویژگی **Position** از **Stream** جالب است. اول اینکه آن یک ویژگی مجازی است، پس متد **OneStatement** من هیچ نظری درباره اینکه چه کدی اجرا خواهد شد که می‌تواند یک اکسپشن تولید کند، ندارد. از سوی دیگر **Stream** از **MarshalByRefObject** مشتق شده است و بنابراین آرگومان استریم می‌تواند به یک شی پروکسی که خودش به یک شی در **AppDomain** دیگری اشاره دارد، اشاره کند. **AppDomain** دیگر می‌تواند تخلیه شده باشد و بنابراین تولید یک **AppDomainUnloadedException** در اینجا ممکن است.

البته، تمام متدهایی که فراخوانی شدند متدهایی هستند که من شخصا کنترلی روی آن‌ها ندارم چون آن‌ها توسط مایکروسافت تولید شده‌اند. و این امکان کاملا وجود دارد که مایکروسافت پیاده‌سازی آن‌ها را در آینده تغییر دهد. پس آن‌ها می‌توانند نوع‌های اکسپشن جدیدی تولید کنند که من در روزی که متد **OneStatement** را نوشتم درباره‌ی آن‌ها چیزی نمی‌دانستم. چگونه احتمالا من می‌توانم متد **OneStatement** را بنویسم که در مقابل تمام این شکست‌های احتمالی مقاوم باشد؟ ضمناً، عکس آن هم یک مشکل است، یک بلوک **catch** می‌تواند یک نوع اکسپشن مشتق شده از نوع اکسپشن تعیین شده را بگیرد و اکنون من کد احیا را برای گونه‌ای متفاوت از شکست دارم اجرا می‌کنم. پس اکنون که شما اطلاعاتی درباره‌ی همه شکست‌های ممکن دارید، احتمالا می‌دانید چرا از نظر جامعه پذیرفته شده است که کد کاملا قابل اطمینان و قدرتمند ننویسیم؛ خیلی ساده، غیر عملی است. گذشته از این، یک نفر می‌تواند بگوید که اصلاً غیرممکن است. حقیقت اینکه خطاها مکرراً رخ نمی‌دهند دلیل دیگری است که چرا این قضیه پذیرفته شده است. چون خطاها (مثل **OutOfMemoryException**) خیلی به ندرت رخ می‌دهند، جامعه تصمیم گرفته است که کد قابل اطمینان را در ازای بهره‌وری برنامه‌نویس معامله کند. یکی از چیزهای جالب درباره‌ی اکسپشن‌ها اینست که یک اکسپشن مدیریت نشده باعث می‌شود برنامه شما از کار بیافتد. این در هنگام آزمایش خوب است، شما مشکلات را به سرعت پیدا می‌کنید و اطلاعاتی (شامل پیام خطا و ردیابی پشته) که با یک اکسپشن مدیریت نشده بدست می‌آورد معمولاً برای رفع مشکل کد کافی است. البته، شرکت‌های زیادی هستند که نمی‌خواهند پس از اینکه برنامه‌شان آزمایش و نصب شد، آن از کار بیافتد. و بنابراین بسیاری از برنامه‌نویسان کدی می‌نویسند که **System.Exception**، کلاس پایه تمام نوع‌های اکسپشن را بگیرد. هر چند، مسئله با گرفتن **System.Exception** و اجازه دادن به برنامه برای ادامه اجرا اینست که وضعیت ممکن است خراب شده باشد.

قبلاً در این فصل، من یک کلاس **Account** نشان دادم که یک متد **Transfer** تعریف می‌کرد و کارش انتقال پول از یک حساب به دیگری بود. وقتی این متد **Transfer** فراخوانی می‌شود، اگر با موفقیت پول را از حساب **from** خارج کرد و آنگاه قبل از افزودن آن به حساب **to** یک اکسپشن تولید کرد، آنگاه چه می‌شود؟ اگر کد فراخوانی کننده **System.Exception** را بگیرد و به اجرا ادامه دهد، آنگاه وضعیت برنامه خراب شده است: هر دو حساب **from** و **to** پول کمتری از آنچه باید داشته باشند دارند. چون در اینجا ما درباره‌ی پول حرف می‌زنیم، این خراب شدن وضعیت فقط به سادگی یک خطا نیست، آن یک خطای امنیتی محسوب می‌شود. اگر برنامه به اجرا ادامه دهد، آن سعی می‌کند انتقال‌ها بیشتری را بین حساب‌ها انجام دهد و اکنون وضعیت خراب درون برنامه گسترده‌تر می‌شود.

یک نفر می‌تواند بگوید که متد **Transfer** خودش باید **System.Exception** را بگیرد و پول را به حساب **from** برگرداند. و این در صورتی که متد **Transfer** به اندازه‌ی کافی ساده باشد، به خوبی کار می‌کند. اما اگر متد **Transfer** یک رکورد حساسی از پول برداشت شده تولید کند یا اگر دیگر تردها همان حساب را در همان زمان دستکاری کنند، آنگاه سعی در خنثی کردن عملیات هم شکست می‌خورد که یک اکسپشن دیگر هم تولید می‌کند. و اکنون وضعیت خراب، بدتر شده است نه بهتر.

^{۶۵} ضمناً، **System.Char**، **System.String**، **System.Type**، **System.IO.Stream** همه سازنده‌های کلاس تعریف می‌کنند که می‌توانند در جایی از برنامه باعث تولید یک **TypeInitializationException** گردند.

نکته یک نفر می‌تواند استدلال کند که دانستن اینکه کجا چیزی خراب شده است از دانستن اینکه چه خطایی رخ داده است مفیدتر است. برای نمونه، دانستن اینکه انتقال پول به خارج از یک حساب شکست خورده به جای دانستن اینکه **Transfer** به خاطر یک **SecurityException** یا **OutOfMemoryException** شکست خورده، مفیدتر است. در حقیقت، مدل خطای Win32 به این روش کار می‌کند. متدها **true/false** برمی‌گردانند تا موفقیت/شکست را نشان دهند تا شما بدانید چه متدی شکست خورده است. آنگاه، اگر برنامه شما به اینکه چرا شکست خورده اهمیت دهد، برنامه، متد **GetLastError Win32** را فراخوانی می‌کند. **System.Exception** یک ویژگی **Source** دارد که به شما متدی که شکست خورده را نشان می‌دهد. اما این ویژگی یک رشته است که شما باید آن را تجزیه کنید و اگر دو متد، در درون، متد یکسانی را فراخوانی کنند، شما فقط از روی ویژگی **Source** نمی‌توانید بگویید کدام متد را کد شما فراخوانی کرده که به شکست منجر شده است. به جای آن، شما مجبورید **String** برگشتی از ویژگی **StackTrace** از **Exception** را برای بدست آوردن این اطلاعات تجزیه کنید. چون این کار بسیار مشکل است، من هرگز کسی را ندیدم که واقعا چنین کاری کند.

چندین چیز وجود دارد که شما می‌توانید انجام دهید تا به کاهش خرابی وضعیت کمک کنید:

- CLR اجازه نمی‌دهد یک ترد درحالی‌که کد درون یک بلوک **catch** یا بلوک **finally** را اجرا می‌کند، بسته شود. بنابراین، ما می‌توانیم به سادگی متد **Transfer** را قدرتمندتر کنیم:

```
public static void Transfer(Account from, Account to, Decimal amount) {
    try { /* do nothing in here */ }
    finally {
        from -= amount;
        // Now, a thread abort (due to Thread.Abort/AppDomain.Unload) can't happen here
        to += amount;
    }
}
```

هر چند، مطلقا توصیه نمی‌شود که همه کدتان را در بلوک **finally** بنویسید، شما باید از این تکنیک فقط برای تغییر وضعیت‌های فوق العاده حساس استفاده کنید.

- شما می‌توانید از کلاس **System.Diagnostics.Contracts.Contract** برای اعمال قراردادهای کد به متدهایتان استفاده کنید. قراردادهای کد به شما راهی می‌دهند که آرگومان و دیگر متغیرها را قبل از آنکه سعی در تغییر وضعیت با استفاده از این آرگومان‌ها/متغیرها داشته باشید، ارزیابی کنید. اگر آرگومان‌ها/متغیرها قرارداد را رعایت کنند، شانس وضعیت خراب، حداقل می‌شود (کاملا حذف نمی‌شود). اگر یک قرارداد شکست بخورد، آنگاه یک اکسپشن قبل از آنکه وضعیتی تغییر کند، تولید می‌شود. من در انتهای فصل درباره‌ی قراردادهای کد صحبت می‌کنم.

- شما می‌توانید از نواحی اجرایی محدود شده (CER ها) استفاده کنید که به شما راهی می‌دهند تا شک و تردید CLR را کم کنید. برای نمونه، قبل از ورود به یک بلوک **try**، شما می‌توانید CLR را مجبور کنید هر اسمبلی که توسط کد درون بلوک‌های **catch** و **finally** همراه آن نیاز است را بارگذاری کند. به علاوه، CLR تمام کد درون بلوک‌های **catch** و **finally**، شامل تمام متدهایی که از درون این بلوک‌ها فراخوانی شده‌اند را کامپایل می‌کند. این کار تولید بخشی از اکسپشن‌های احتمالی (شامل **BadImageFormatException**، **FileLoadException**، **MethodAccessException**، **FieldAccessException**، **InvalidProgramException**، **MissingFieldException** و **MissingMethodException**) را وقتی سعی دارد کد احیای خطا (در بلوک **catch**) یا کد پاکسازی (در بلوک **finally**) را اجرا کند، حذف می‌کند. این همچنین احتمال **OutOfMemoryException** و دیگر اکسپشن‌ها را هم کاهش می‌دهد. من بعدا درباره‌ی CER ها در این فصل صحبت می‌کنم.

- بسته به اینکه وضعیت در کجا زندگی می‌کند، شما می‌توانید از تراکنش‌ها استفاده کنید که مطمئن می‌شوند تمام وضعیت تغییر کرده یا هیچ وضعیتی تغییر نکرده باشد. برای نمونه اگر داده در یک پایگاه داده است، تراکنش خوب کار می‌کند. ویندوز اکنون از رجیستری و عمیات‌های فایل (فقط روی یک درایو NTFS) به صورت تراکنشی، پشتیبانی می‌کند و شاید شما بتوانید از این استفاده کنید؛ اگرچه اکنون دات‌نت فریمورک این

کاربرد را ارائه نمی‌کند. شما مجبورید برای آزاد سازی آن، کد اصلی را `P/Invoke` کنید. کلاس `System.Transactions.TransactionScope` را برای جزئیات در این باره نگاه کنید.

شما می‌توانید متدهایتان را به گونه‌ای طراحی کنید که صریح تر باشند. برای نمونه، کلاس `Monitor` معمولاً برای گرفتن/آزاد کردن قفل همزمانی یک ترد مثل زیر استفاده می‌شود:

```
public static class SomeType {
    private static Object s_myLockObject = new Object();
    public static void SomeMethod () {
        Monitor.Enter(s_myLockObject); // If this throws, did the lock get taken or
                                        // not? If it did, then it won't get released!

        try {
            // Do thread-safe operation here...
        }
        finally {
            Monitor.Exit(s_myLockObject);
        }
    }
    // ...
}
```

به خاطر مشکلاتی که در بالا نشان داده شد، متد `Enter` از `Monitor` که در بالا استفاده می‌شود دیگر توصیه نمی‌گردد و توصیه می‌شود کد فوق را اینگونه بنویسید:

```
public static class SomeType {
    private static Object s_myLockObject = new Object();
    public static void SomeMethod () {
        Boolean lockTaken = false; // Assume the lock was not taken
        try {
            // This works whether an exception is thrown or not!
            Monitor.Enter(s_myLockObject, ref lockTaken);

            // Do thread-safe operation here...
        }
        finally {
            // If the lock was taken, release it
            if (lockTaken) Monitor.Exit(s_myLockObject);
        }
    }
    // ...
}
```

در حالیکه صراحت در این کد یک پیشرفت محسوب می‌شود، در مورد قفل‌های همزمانی ترد توصیه اکنون اینست که از آن‌ها با مدیریت اکسپشن اصلاً استفاده نشود. فصل ۲۹ "ساخت‌های ترکیبی همزمانی ترد" را برای اطلاعات بیشتر در این باره ببینید.

اگر در کدتان، شما تعیین نموده اید که فرای تعمیر، وضعیت از قبل خراب بوده است، آنگاه شما باید هر وضعیت خراب شده را از بین ببرید تا باعث صدمه‌های بیشتری نشود. سپس، برنامه تان را مجدداً اجرا کنید تا وضعیت شما، خودش را به یک شرایط خوب مقداردهی اولیه کنید و با امیدواری، خراب شدن وضعیت مجدداً رخ نخواهد داد. چون وضعیت مدیریت‌شده نمی‌تواند به بیرون از یک `AppDomain` درز پیدا کند، شما می‌توانید هر وضعیت خراب که درون یک `AppDomain` زندگی می‌کند را با تخلیه کردن (`unload`) کل `AppDomain` بوسلیه فراخوانی متد `Unload` از `AppDomain` (فصل ۲۲ را برای جزئیات ببینید) از بین ببرید.

و اگر شما احساس می‌کنید وضعیت شما آنقدر بد است که کل فرآیند باید متوقف شود، آنگاه شما می‌توانید متد استاتیک **FailFast** از **Environment** را فراخوانی کنید:

```
public static void FailFast(String message);
public static void FailFast(String message, Exception exception);
```

این متد پردازش را بدون اجرای هر بلوک **try/finally** فعال، یا متدهای **Finalize** از بین می‌برد. این خوب است چون اجرای کد بیشتر، وقتی وضعیت خراب است می‌تواند اوضاع را بدتر کند. هر چند، **FailFast** تنها به اشیاء مشتق شده از **CriticalFinalizerObject** که در فصل ۲۱ "مدیریت حافظه اتوماتیک (جمع آوری زباله)" بحث می‌شوند، شانس پاکسازی را می‌دهد. این معمولا درست است چون آن‌ها معمولا تمایل دارند منابع خودی (native) را ببندند و وضعیت ویندوز حتی اگر وضعیت CLR یا وضعیت برنامه خراب شده باشد، احتمالا خوب است. متد **FailFast**، رشته پیام و اکسپشن انتخابی (معمولا اکسپشنی که در یک بلوک **catch** گرفته می‌شود) را در **Windows Application event log** نوشته، یک گزارش خطای ویندوز تولید می‌کند، یک کپی از حافظه برنامه تان می‌سازد و آنگاه پردازش کنونی را از بین می‌برد.

مهم اکثر کدهای **FCL** مایکروسافت مطمئن نمی‌شوند که در صورت بروز یک اکسپشن مدیریت نشده، وضعیت خوب باقی بماند. اگر کد شما یک اکسپشن که از کد **FCL** رد شده است را بگیرد و سپس به استفاده از اشیاء **FCL** ادامه دهد، این احتمال وجود دارد که این اشیاء غیرقابل پیش‌بینی رفتار کنند. شرم آور است که بیشتر اشیاء **FCL** وضعیت شان در مواجهه با اکسپشن‌های مدیریت نشده را بهتر نگهداری نمی‌کنند یا اگر وضعیت آن‌ها قابل برگشت نباشد، **FailFast** را فراخوانی نمی‌کنند.

هدف این بحث این است که شما را از مسائل احتمالی که به کار با مکانیزم مدیریت اکسپشن **CLR** مربوط است، آگاه کند. اغلب برنامه‌ها نمی‌توانند با اجرا با وضعیت خراب، مدارا کنند چون منجر به داده‌های غلط و حفره‌های امنیتی احتمالی می‌گردد. اگر شما برنامه‌ای می‌نویسید که نمی‌تواند از بین رفتن را تحمل کند (مثل یک سیستم عامل یا موتور پایگاه داده)، آنگاه کد مدیریت شده تکنولوژی مناسبی برای استفاده نیست. و در حالیکه **Microsoft Exchange Server** بیشتر در کد مدیریت شده نوشته شده است از پایگاه داده اصلی (native) برای ذخیره پیام‌های ایمیل استفاده می‌کند. پایگاه داده اصلی **Extensible Storage Engine** نامیده می‌شود که با ویندوز عرضه شده و معمولا در **C:\Windows\System32\EseNT.dll** وجود دارد. برنامه شما می‌تواند اگر بخواهید از این موتور نیز استفاده کند؛ در وب سایت **MSDN** مایکروسافت به دنبال "Extensible Storage Engine" بگردید.

کد مدیریت شده انتخاب خوبی برای برنامه‌هایی است که وقتی وضعی برنامه خراب می‌شود، بتوانند از بین رفتن (termination) را تحمل کنند. برنامه‌های فراوانی هستند که در این دسته جای می‌گیرند. همچنین، منابع و مهارت بیشتری برای نوشتن یک کتابخانه کلاس یا برنامه اصلی (native) لازم است. برای بسیاری برنامه‌ها، کد مدیریت شده انتخاب بهتری است چون بهره‌وری برنامه‌نویس را بسیار زیاد افزایش می‌دهد.

راهنمایی‌ها و بهترین تجربه‌ها

درک مکانیزم اکسپشن قطعا مهم است. به همان قدر مهم است که بدانید چگونه عاقلانه اکسپشن‌ها را به کار ببرید. بسیار زیاد من برنامه‌نویسان کتابخانه‌ای را دیده‌ام که همه نوع اکسپشن را می‌گیرند و مانع از آن می‌شوند که برنامه‌نویس برنامه بدانند مشکلی رخ داده است. در این بخش، من راهنمایی‌هایی برای برنامه‌نویسان ارائه می‌کنم تا بدانند چه موقع از اکسپشن‌ها استفاده کنند.

مهم اگر شما یک برنامه‌نویس کتابخانه کلاس (class library developer) هستید که نوع‌هایی طراحی می‌کنید که توسط دیگر برنامه‌نویسان استفاده می‌شوند، این راهنمایی‌ها را خیلی جدی بگیرید. شما مسئولیت بزرگی دارید: شما سعی دارید نوعی را در کتابخانه کلاس طراحی کنید که برای برنامه‌های مختلفی کاربرد دارد. به خاطر داشته باشید که شما دانش دقیقی از کدی که (از طریق نماینده‌ها، متدهای مجازی یا متدهای رابط) کالیک (فراخوانی) می‌کنید، ندارید. و شما نمی‌دانید چه کدی شما را فراخوانی می‌کند. پیش بینی تمام حالت‌هایی که نوع شما در آن‌ها استفاده می‌شود عملی نیست پس هیچ‌گونه تصمیم‌گیری نگیرید. کد شما نباید تصمیم بگیرد که چه شرایطی یک خطا را تشکیل می‌دهد؛ بگذارید فراخوانی کننده این تصمیم را بگیرد.

به علاوه، خیلی دقیق مراقب وضعیت باشید و سعی نکنید آن را خراب کنید. آرگومان‌هایی که به متد شما ارسال شده‌اند را با استفاده از قراردادهای کد ارزیابی کنید (که در انتهای این فصل بحث می‌شود). سعی کنید اصلاً وضعیت را تغییر ندهید. اگر شما وضعیت را تغییر دهید، آنگاه برای یک شکست آماده باشید و سپس سعی کنید آن وضعیت را برگردانید. اگر شما از راهنمایی‌های این فصل پیروی کنید، برنامه‌نویسان برنامه در استفاده از نوع‌های کتابخانه کلاس شما اوقات سختی را نخواهند گذراند.

اگر شما یک برنامه‌نویس برنامه (application developer) هستید، هر سیاستی که فکر می‌کنید درست است را تعریف کنید. پیروی از راهنمایی‌های این فصل به شما کمک می‌کند مشکلات را در کدتان پیش از آنکه آزاد شوند کشف کنید که به شما اجازه می‌دهد آن‌ها را برطرف کنید تا برنامه‌تان را قدرتمندتر نمایید. هر چند، پس از ملاحظات کافی آزادید که دیگر این راهنمایی‌ها را رها کنید. شما سیاست را تنظیم می‌کنید. برای نمونه، کد برنامه می‌تواند در گرفتن اکسپشن‌ها از کد کتابخانه کلاس خشن تر باشد.

از بلوک های finally آزدانه استفاده کنید

من فکر می‌کنم بلوک‌های finally عالی هستند! آن‌ها به شما اجازه می‌دهند یک بلوک کد تعیین کنید که بدون تفاوت به اینکه چه نوع اکسپشنی تولید می‌کند، تضمینی اجرا می‌شوند. شما باید از بلوک‌های finally برای پاکسازی هر گونه عملیات قبل از آنکه به فراخوانی کننده برگردید یا به کد پس از بلوک finally اجازه اجرا دهید، استفاده کنید. شما همچنین مکرراً از بلوک‌های finally برای از بین بردن صریح هر شی استفاده می‌کنید تا از کمبود منابع جلوگیری کنید. یک مثال که تمام کد پاکسازی (بستن فایل) را در یک بلوک finally قرار می‌دهد:

```
using System;
using System.IO;
```

```
public sealed class SomeType {
    private void SomeMethod() {
        FileStream fs = new FileStream(@"C:\Data.bin ", FileMode.Open);
        try {
            // Display 100 divided by the first byte in the file.
            Console.WriteLine(100 / fs.ReadByte());
        }
        finally {
            // Put cleanup code in a finally block to ensure that the file gets closed
            // regardless of whether or not an exception occurs (for example, the first
            // byte was 0).
            if (fs != null) fs.Dispose();
        }
    }
}
```

اطمینان از اینکه کد پاکسازی همیشه اجرا می‌شود آنقدر مهم است که بسیاری از زبان‌های برنامه‌نویسی ساخت‌هایی ارائه می‌کنند که نوشتن کد پاکسازی را آسانتر کند. برای نمونه، زبان سی‌شارپ به صورت خودکار بلوک‌های try/finally را وقتی شما از عبارتهای lock، using و foreach استفاده می-

کنید، تولید می‌کند. کامپایلر سی‌شارپ همچنین وقتی شما مخرب (destructor) یک کلاس (متد **Finalize**) را بازنویسی می‌کنید، بلوک‌های **try/finally** را تولید می‌کند. هنگام استفاده از این ساختارها، کامپایلر کدی که شما نوشته اید را درون بلوک **try** قرار می‌دهد و به صورت خودکار کد پاکسازی را درون بلوک **finally** قرار می‌دهد. به خصوص:

- وقتی شما از عبارت **lock** استفاده می‌کنید، قفل درون یک بلوک **finally** آزاد می‌شود.
 - وقتی شما از عبارت **using** استفاده می‌کنید، متد **Dispose** از شی، درون یک بلوک **finally** فراخوانی می‌شود.
 - وقتی از عبارت **foreach** استفاده می‌کنید، متد **Dispose** از شی **IEnumerator** درون یک بلوک **finally** فراخوانی می‌شود.
 - وقتی شما یک متد مخرب تعریف می‌کنید، متد **Finalize** از کلاس پایه درون یک بلوک **finally** فراخوانی می‌شود.
- برای نمونه، کد سی‌شارپ زیر از عبارت **using** بهره می‌برد. این کد از کدی که در مثال قبلی نشان داده شد کوتاهتر است، اما کدی که کامپایلر تولید می‌کند معادل کدی تولید شده در مثال قبلی است:

```
using System;
using System.IO;

internal sealed class SomeType {
    private void SomeMethod() {
        using (FileStream fs = new FileStream(@"C:\Data.bin", FileMode.Open)) {
            // Display 100 divided by the first byte in the file.
            Console.WriteLine(100 / fs.ReadByte());
        }
    }
}
```

برای اطلاعات بیشتر درباره عبارت **using**، به فصل ۲۱ و برای اطلاعات بیشتر در مورد عبارت **lock** به فصل ۲۹ "ساخت‌های ترکیبی همزمانی ترد" مراجعه کنید.

هرچیزی را نگیرید

یک اشتباه همیشگی که برنامه‌نویسانی که برای استفاده صحیح از اکسپشن‌ها آموزش ندیده‌اند، مرتکب می‌شوند استفاده از بلوک‌های **catch** فراوان و به اشتباه است. وقتی شما یک اکسپشن را می‌گیرید، شما بیان می‌کنید که انتظار یک اکسپشن را داشته‌اید، شما می‌دانید چرا رخ داده است، و شما می‌دانید چگونه با آن برخورد کنید. به بیان دیگر، شما یک سیاست برای برنامه تعریف می‌کنید. تمام این به بخش "معامله قابلیت امنیت در مقابل بهره‌وری" در این فصل برمی‌گردد.

بسیار زیاد من کدی اینگونه می‌بینم:

```
try {
    // try to execute code that the programmer knows might fail...
}
catch (Exception) {
    ...
}
```

این کد بیان می‌کند که انتظار هر و همه‌ی اکسپشن‌ها را داشته و می‌داند چگونه از هر و همه‌ی وضعیت‌ها احیا شود. این چگونه ممکن است؟ یک نوع که بخشی از یک کتابخانه کلاس است نباید هرگز و تحت هیچ شرایطی تمام اکسپشن‌ها را گرفته و ببلعد چرا که برای نوع راهی نیست که دقیقاً بدانند برنامه قصد دارد چگونه به یک اکسپشن پاسخ دهد. به علاوه، نوع، مدام به کد برنامه از طریق یک نماینده، متد مجازی یا متد رابط فراخوانی می‌کند. اگر کد برنامه یک اکسپشن تولید کند، بخش دیگری از برنامه احتمالاً انتظار گرفتن این اکسپشن را دارد. اکسپشن باید اجازه داده شود تا راهش را به بالای پشته فراخوانی فیلتر کند و به کد برنامه اجازه‌ی مدیریت اکسپشن به محضی که آن را می‌بیند، بدهد.

اگر اکسپشن مدیریت نشده باشد، CLR پردازش را از بین می‌برد. من اکسپشن‌های مدیریت نشده را در انتهای فصل خواهیم گفت. اغلب اکسپشن‌های مدیریت نشده در حین تست کد پیدا می‌شوند. برای حل این اکسپشن‌های مدیریت نشده، شما باید یا کد را تغییر دهید تا به دنبال یک اکسپشن خاص بگردد یا کد را

بازنویسی کنید تا شرایطی که منجر به تولید اکسپشن می‌شود را حذف کنید. نسخه نهایی کد که در یک محیط عملی اجرا می‌شود باید اکسپشن‌های مدیریت نشده اندکی را ببیند و بسیار باید قدرتمند باشد.

نکته در بعضی موارد، یک متد که نمی‌تواند وظیفه‌اش را کامل کند کشف می‌کند که وضعیت بعضی اشیاء خراب شده است و قابل برگشت نیست. اجازه دادن به ادامه‌ی اجرای برنامه می‌تواند به رفتار غیر قابل پیش‌بینی یا آسیب پذیری امنیتی منجر شود. وقتی این وضعیت یافت می‌شود، آن متد نباید یک اکسپشن تولید کند و به جای آن باید پرده را مجبور کند بلافاصله با فراخوانی **FailFast** از **System.Environment** بسته شود.

در ضمن مشکلی نیست اگر **System.Exception** را گرفته و کدی درون آکولت‌های بلوک **catch** اجرا کنید البته تا زمانی که اکسپشن را در انتهای کد مجدداً تولید کنید. گرفتن **System.Exception** و بلعیدن اکسپشن (نه بازتولید آن) هرگز نباید انجام شود چون شکست‌هایی را مخفی می‌کند که به برنامه اجازه می‌دهد با نتایج غیرقابل پیش‌بینی و آسیب پذیری احتمالی امنیتی اجرا شود. ابزار آنالیز کد ویژوال استودیو (**FxCopCmd.exe**) هر کدی که حاوی یک بلوک **catch (Exception)** باشد را علامت می‌گذارد مگر آنکه یک عبارت **throw** در کد بلوک موجود باشد. بخش "برگشت از یک عمل نیمه کامل وقتی یک اکسپشن غیر قابل احیا رخ می‌دهد - نگهداری وضعیت" به زودی در این فصل، این الگو را توضیح می‌دهد. سرانجام، این درست است که یک اکسپشن در یک ترد رخ دهد و در ترد دیگری باز تولید شود. مدل برنامه‌نویسی غیرهمزمان **Asynchronous Programming Model** (که در فصل ۲۷ "عملیات‌های غیرهمزمان وابسته به ورودی خروجی" بحث می‌شود) این را پشتیبانی می‌کند. برای نمونه، اگر یک ترد از استخر ترد کدی را اجرا کند که یک اکسپشن تولید نماید، **CLR** اکسپشن را گرفته و می‌بلعد و به ترد اجازه می‌دهد به استخر ترد برگردد. بعداً، تردی باید متد **EndXxx** را فراخوانی کند تا نتیجه عملیات غیرهمزمان را تعیین کند. متد **EndXxx** همان شی اکسپشنی را تولید می‌کند که توسط ترد استخر ترد که کار واقعی را انجام داده، تولید شده است. در این سناریو، اکسپشن توسط اولین ترد بلعیده می‌شود هر چند، اکسپشن توسط تردی که متد **EndXxx** را فراخوانی می‌کند بازتولید می‌شود، بنابراین از دید برنامه مخفی نمی‌شود.

احیای آرام از یک اکسپشن

گاهی شما با دانستن برخی از اکسپشن‌هایی که یک متد ممکن است تولید کند، آن را فراخوانی می‌کنید. چون شما انتظار این اکسپشن‌ها را دارید، شاید شما بخواهید کدی داشته باشید که به برنامه شما اجازه دهد از این وضعیت به آرامی احیا شده و به اجرا ادامه دهد.

مثالی با شبه کد ببینید:

```
public string CalculateSpreadsheetCell(Int32 row, Int32 column) {
    String result;
    try {
        result = /* Code to calculate value of a spreadsheet's cell */
    }
    catch (DivideByZeroException) {
        result = "Can't show value: Divide by zero";
    }
    catch (OverflowException) {
        result = "Can't show value: Too big";
    }
    return result;
}
```

این شبه کد محتویات یک خانه در یک صفحه گسترده (صفحات اکسل) را محاسبه کرده و یک رشته که نمایانگر مقدار است را به فراخوانی کننده برمی‌گرداند تا فراخوانی کننده بتواند رشته را در پنجره‌ی برنامه نمایش دهد. هر چند، محتویات یک خانه ممکن است نتیجه تقسیم یک خانه بر خانه‌ی دیگر باشد. اگر خانه حاوی مقسوم علیه، مقدار 0 داشته باشد، **CLR** یک شی **DividedByZeroException** تولید می‌کند. در این مورد، متد، این اکسپشن خاص را می‌گیرد و یک رشته خاص که به کاربر نمایش داده می‌شود را برمی‌گرداند. به طریق مشابه، محتویات یک خانه ممکن است نتیجه ضرب یک خانه در دیگری باشد. اگر مقدار ضرب شده در تعداد بیت مجاز جا نشود **CLR** یک شی **OverflowException** تولید می‌کند و مجدداً، یک رشته خاص به کاربر نمایش داده می‌شود.

وقتی شما اکسپشن‌های خاص را می‌گیرید، شرایطی که باعث تولید اکسپشن می‌شود را کاملاً درک کنید و بدانید چه نوع‌های اکسپشن از نوع اکسپشنی که شما می‌گیرید مشتق شده اند. **System.Exception** را (بدون بازتولید) نگیرید و مدیریت نکنید چون برای شما امکان ندارد که تمام اکسپشن‌های ممکن که درون بلوک **try** شما تولید می‌شوند را بدانید (به خصوص اگر شما به **OutOfMemoryException** و **StackOverflowException** توجه کنید).

برگشت از یک عملیات نیمه کامل وقتی یک اکسپشن غیرقابل احیا رخ می‌دهد – نگهداری وضعیت

معمولاً، برای انجام یک تک عمل انتزاعی، متدها چندین متد دیگر را فراخوانی می‌کنند. برخی از تک تک متدها ممکن است با موفقیت کامل شوند و برخی شاید نشوند. برای نمونه، بگوییم که شما یک مجموعه از اشیاء را به یک فایل روی دیسک سریالی می‌کنید. پس از سریالی کردن ۱۰ شی، یک اکسپشن تولید می‌شود (شاید دیسک پر باشد یا شی بعدی که باید سریالی شود با صفت سفارشی **Serializable** علامت زده نشده باشد). در این لحظه، اکسپشن باید تا فراخوانی کننده فیلتر شود اما در مورد وضعیت فایل روی دیسک چطور؟ فایل اکنون خراب شده است چون آن شامل یک شی نیمه سریالی شده است. عالی است اگر برنامه بتواند از عمل نیمه کامل به عقب برگردد تا فایل در وضعیتی قرار گیرد که قبل از اینکه هر گونه شی‌ای سریالی شود، بود. کد زیر روش صحیح پیاده‌سازی این را نشان می‌دهد:

```
public void SerializeObjectGraph(FileStream fs, IFormatter formatter, Object rootObj) {
    // Save the current position of the file.
    Int64 beforeSerialization = fs.Position;
    try {
        // Attempt to serialize the object graph to the file.
        formatter.Serialize(fs, rootObj);
    }
    catch { // Catch any and all exceptions.
        // If ANYTHING goes wrong, reset the file back to a good state.
        fs.Position = beforeSerialization;
        // Truncate the file.
        fs.SetLength(fs.Position);
        // NOTE: The preceding code isn't in a finally block because
        // the stream should be reset only when serialization fails.
        // Let the caller(s) know what happened by re-throwing the SAME exception.
        throw;
    }
}
```

برای آنکه به درستی عمل نیمه کامل را برگردانید، کدی بنویسید که همه‌ی اکسپشن‌ها را بگیرد. بله، تمام اکسپشن‌ها را بگیرد چون شما به اینکه چه گونه خطایی رخ داده اهمیت نمی‌دهید، شما نیاز دارید که ساختمان داده‌های خود را به وضعیت مناسبی برگردانید، بعد از آنکه اکسپشن را گرفته و مدیریت کردید، آن را نبلید – بگذارید فراخوانی کننده بداند که اکسپشن رخ داده است. شما این کار را با باز تولید همان اکسپشن انجام می‌دهید. در واقع، سی‌شارپ و بسیاری زبان‌های برنامه‌نویسی دیگر این کار را آسان می‌کنند. تنها از کلمه کلیدی **throw** سی‌شارپ بدون تعیین هیچ چیزی پس از **throw** استفاده کنید، همانند کد قبلی.

توجه کنید که بلوک **catch** در مثال قبلی یک نوع اکسپشن را تعیین نمی‌کند چون من می‌خواهم هر و همه اکسپشن‌ها را بگیرم. به علاوه، کد درون بلوک **catch** نیاز ندارد بداند دقیقاً چه اکسپشنی تولید شده، فقط اینکه بداند چیزی اشتباه شده، کافیت. خوشبختانه، سی‌شارپ به من اجازه می‌دهد این کار را با تعیین نکردن هیچ نوع اکسپشن و اینکه عبارت **throw** همان اکسپشنی که گرفته شده را بازتولید کند، به راحتی انجام دهم.

مخفی سازی جزئیات پیاده سازی برای حفظ یک "قرارداد"

در برخی موقعیت‌ها، شاید شما این را مفید یابید که یک اکسپشن را بگیرید و اکسپشن متفاوتی را بازتولید کنید. تنها دلیل این کار حفظ معنی یک قرارداد متد است. همچنین، نوع اکسپشن جدید که شما تولید می‌کنید باید یک اکسپشن خاص باشد (یک اکسپشن که به عنوان نوع پایه برای دیگر اکسپشن‌ها استفاده

نشود). یک نوع **PhoneBook** را تصور کنید که متدی را تعریف می‌کند که از روی یک نام یک شماره تلفن را پیدا می‌کند، همانگونه که در شبه کد زیر نشان داده شده است:

```
internal sealed class PhoneBook {
    private String m_pathname; // path name of file containing the address book

    // Other methods go here.

    public String GetPhoneNumber(String name) {
        String phone;
        FileStream fs = null;
        try {
            fs = new FileStream(m_pathname, FileMode.Open);
            // Code to read from fs until name is found goes here
            phone = /* the phone # found */
        }
        catch (FileNotFoundException e) {
            // Throw a different exception containing the name, and
            // set the originating exception as the inner exception.
            throw new NameNotFoundException(name, e);
        }
        catch (IOException e) {
            // Throw a different exception containing the name, and
            // set the originating exception as the inner exception.
            throw new NameNotFoundException(name, e);
        }
        finally {
            if (fs != null) fs.Close();
        }
        return phone;
    }
}
```

اطلاعات شماره تلفن‌ها از روی یک فایل (در مقابل یک ارتباط شبکه یا پایگاه داده) بدست می‌آید. هر چند، کاربر نوع **PhoneBook** این را نمی‌داند چون این یک جزئیات پیاده‌سازی است که ممکن است در آینده تغییر کند. پس اگر فایل یافت نشد یا به هر دلیل قابل خواندن نبود، فراخوانی کننده یک **FileNotFoundException** یا **IOException** را خواهد دید که قابل پیش‌بینی نیست. به بیان دیگر، وجود فایل و قابلیت خواندن شدن، بخشی از قرارداد عملی متد نیست: راهی برای فراخوانی کننده که این را حدس بزند وجود ندارد. پس متد **GetPhoneNumber** دو نوع اکسپشن را می‌گیرد و یک اکسپشن جدید **NameNotFoundException** تولید می‌کند.

هنگام استفاده از این تکنیک، شما باید اکسپشن‌های خاصی را بگیرید که کاملاً شما شرایطی که باعث تولید اکسپشن می‌شوند را درک می‌کنید. و شما باید نوع‌های اکسپشن که از نوع اکسپشنی که شما می‌گیرید، مشتق می‌شوند را بدانید. تولید یک اکسپشن به فراخوانی کننده اجازه می‌دهد بدانند متد کارش را کامل انجام نداده است و نوع **NameNotFoundException** به فراخوانی کننده دیدی انتزاعی از علت آن می‌دهد. تنظیم اکسپشن درونی به **FileNotFoundException** یا **IOException** مهم است تا دلیل واقعی اکسپشن گم نشود. بعلاوه، دانستن اینکه چه چیزی باعث اکسپشن شده می‌تواند برای برنامه‌نویسان نوع **PhoneBook** و احتمالاً برنامه‌نویسی که از نوع **PhoneBook** استفاده می‌کند، مفید باشد.

مهم وقتی شما از این تکنیک استفاده می‌کنید، شما به فراخوانی کننده درباره‌ی دو چیز دروغ می‌گویید. اول، شما درباره‌ی آنچه واقعا اشتباه شده دروغ می‌گویید. در مثال من، یک فایل یافت نشده بود اما من گزارش کردم یک نام یافت نشده بود. دوم، شما درباره‌ی اینکه شکست کجا رخ داده است دروغ می‌گویید. اگر **FileNotFoundException** مجاز باشد در پشته فراخوانی بالا رود، ویژگی **StackTrace** آن خطایی که درون سازنده‌ی **FileStream** رخ داده را منعکس خواهد کرد. اما وقتی من این اکسپشن را بلعم و یک اکسپشن **NameNotFoundException** جدید تولید کنم، ردیابی پشته نشان خواهد داد که خطا درون بلوک **catch** چندین خط دورتر از جاییکه اکسپشن واقعا رخ داده بود، رخ داده است. این می‌تواند خطایابی را مشکل کند، پس این تکنیک باید با مراقبت فراوانی استفاده شود.

حال بگذارید بگوییم که نوع **PhoneBook** کمی متفاوت پیاده‌سازی شده است. فرض کنید که نوع، یک ویژگی عمومی **PhoneBookPathname** ارائه می‌کند که به کاربر اجازه می‌دهد مسیر فایلی که شماره تلفن‌ها باید در آن جستجو شود را در آن بنویسد یا از آن بخواند. چون کاربر از این حقیقت که اطلاعات شماره تلفن‌ها از روی یک فایل خوانده می‌شود، آگاه است، من متد **GetPhoneNumber** را تغییر می‌دهم تا هیچ اکسپشنی نگیرد، به جای آن من اجازه می‌دهم هر اکسپشنی که تولید شد به بیرون متد انتشار یابد. توجه کنید که من هیچ یک از پارامترهای متد **GetPhoneNumber** را تغییر نمی‌دهم اما من انتزاعی که به کاربران نوع **PhoneBook** ارائه می‌کنم را تغییر می‌دهم. کاربران اکنون انتظار دارند یک مسیر، بخشی از قرارداد **PhoneBook** باشد.

گاهی برنامه‌نویسان یک اکسپشن را می‌گیرند و یک اکسپشن جدید را جهت افزودن اطلاعات اضافی یا زمینه (context) به یک اکسپشن، تولید می‌کنند. هر چند، اگر این همه‌ی چیز است که می‌خواهید انجام دهید، شما باید فقط نوع اکسپشنی که می‌خواهید، را بگیرید، با ویژگی مجموعه‌ای **Data** از شی اکسپشن، داده‌هایی بیافزایید و سپس همان شی اکسپشن را بازتولید کنید:

```
private static void SomeMethod(String filename) {
    try {
        // Do whatever here...
    }
    catch (IOException e) {
        // Add the filename to the IOException object
        e.Data.Add("Filename", filename);

        throw; // re-throw the same exception object that now has additional data in it
    }
}
```

یک استفاده خوب از این تکنیک: وقتی یک سازنده نوع یک اکسپشن تولید می‌کند که درون متد سازنده نوع گرفته نمی‌شود، CLR در درون، آن اکسپشن را گرفته و به جای آن یک **TypeInitializationException** جدید تولید می‌کند. این مفید است چون CLR کدی را درون متدهای شما تولید می‌کند که به صورت ضمنی سازنده‌ی نوع را فراخوانی می‌کند.^{۶۶} اگر سازنده‌ی نوع، یک **DividedByZeroException** تولید کند، کد شما شاید سعی کند آن را گرفته و از آن احیا شود اما شما حتی نمی‌دانید که در حال فراخوانی سازنده‌ی نوع هستید. پس CLR، **DividedByZeroException** را به یک **TypeInitializationException** تبدیل می‌کند تا شما واضحا بدانید که اکسپشن به خاطر یک شکست سازنده نوع، رخ داده است. مشکل با کد شما نیست.

در سوی دیگر، یک استفاده بد از این تکنیک: وقتی شما متدی را از طریق رفلکشن فراخوانی می‌کنید، CLR در درون، هر اکسپشن تولیدی توسط متد را گرفته و آن را به یک **TargetInvocationException** تبدیل می‌کند. این به شدت ناراحت کننده است که شما باید اکنون شی **TargetInvocationException** را گرفته و به ویژگی **InnerException** آن نگاه کنید تا دلیل واقعی شکست را بفهمید. در واقع، هنگام استفاده از رفلکشن، رایج است که کدی شبیه به این ببینید:

```
private static void Reflection(Object o) {
    try {
        // Invoke a DoSomething method on this object
```

^{۶۶} برای اطلاعات بیشتر در این باره، بخش "سازنده‌های نوع" در فصل ۸ "متدها" را ببینید.

```

var mi = o.GetType().GetMethod("DoSomething");
mi.Invoke(o, null); // The DoSomething method might throw an exception
}
catch (System.Reflection.TargetInvocationException e) {
    // The CLR converts reflection-produced exceptions to TargetInvocationException
    throw e.InnerException; // Re-throw what was originally thrown
}
}

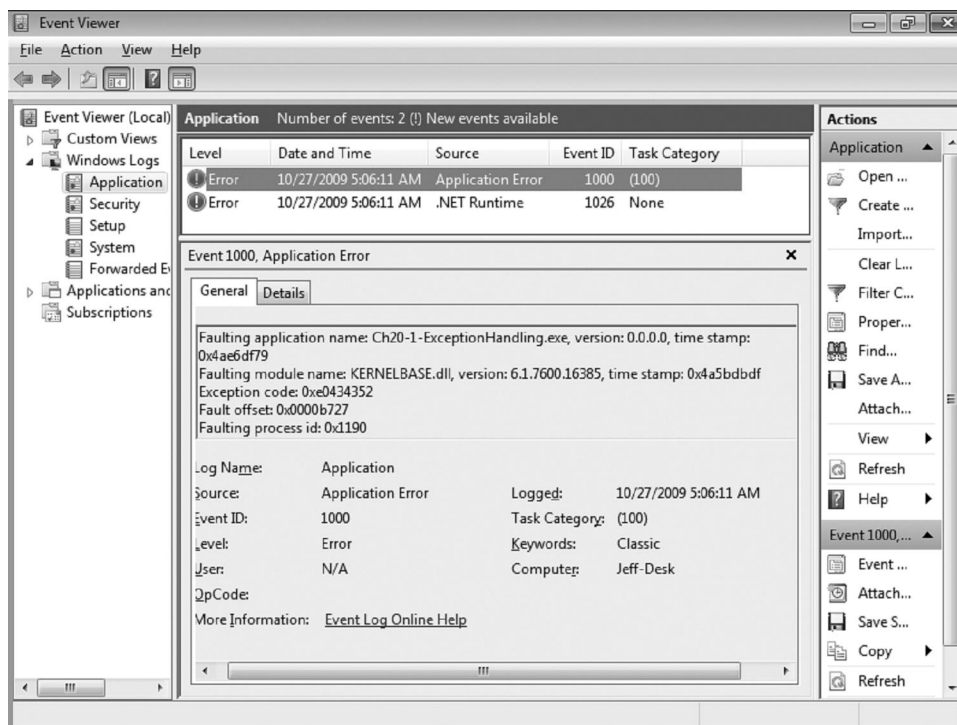
```

هر چند من خبرهای خوبی دارم: اگر شما از نوع اصلی **dynamic** سی شارپ (که در فصل ۵ "نوع‌های اصلی، ارجاعی و مقداری" بحث شد) برای درخواست (فراخوانی) یک عضو استفاده می‌کنید، کد تولیدی توسط کامپایلر، هر و همه‌ی اکسپشن‌ها را نمی‌گیرد و یک شی **TargetInvocationException** تولید نمی‌کند؛ اکسپشن تولید شده‌ی اصلی به سادگی در پشته بالا می‌رود. برای بسیاری برنامه‌نویسان این دلیل خوبی برای ترجیح استفاده از نوع اصلی **dynamic** سی شارپ بر رفلکشن است.

اکسپشن‌های مدیریت نشده

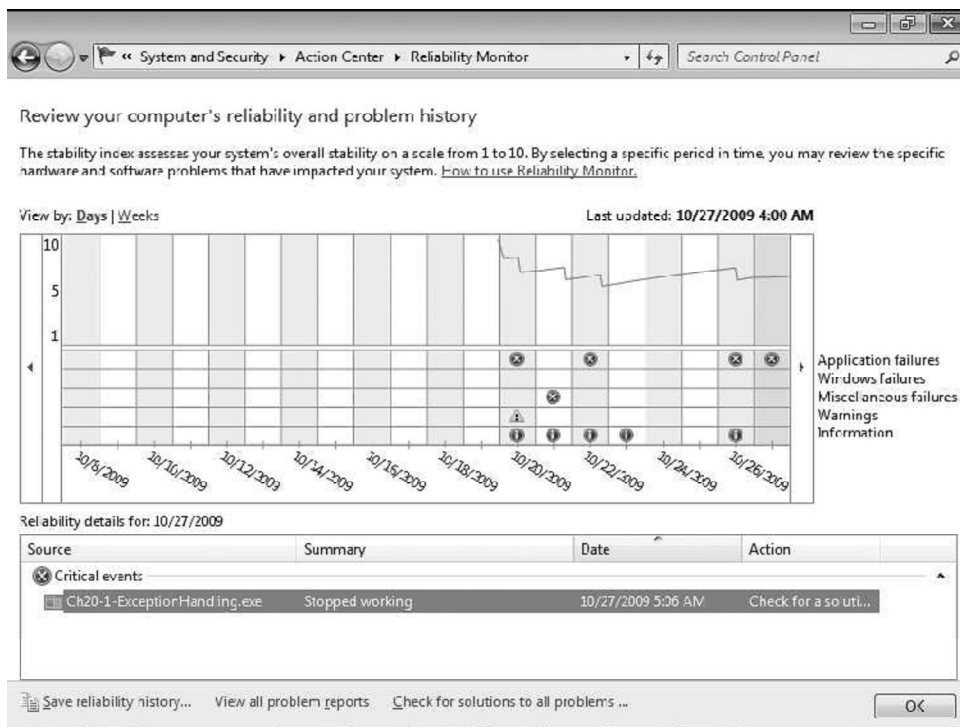
وقتی یک اکسپشن تولید می‌شود، CLR در پشته فراخوانی بالا می‌رود و به دنبال بلوک‌های **catch** که با نوع اکسپشن تولید شده، مطابقت می‌گردد. اگر هیچ بلوک **catch** مطابق با نوع اکسپشن تولیدی یافت نشود، یک اکسپشن مدیریت نشده **unhandled exception** رخ می‌دهد. وقتی CLR کشف می‌کند یک ترد در یک پردازش، یک اکسپشن مدیریت نشده دارد، CLR پردازش را از کار می‌اندازد (می‌بندد). یک اکسپشن مدیریت نشده یک وضعیتی را نشان می‌دهد که برنامه آن را پیش‌بینی نکرده و یک اشکال واقعی در برنامه محسوب می‌شود. در این لحظه، اشکال باید به شرکت سازنده‌ی برنامه گزارش شود. امیدوارانه، سازنده اشکال را برطرف کرده و یک نسخه جدید از برنامه توزیع می‌کند.

برنامه‌نویسان کتابخانه کلاس درباره‌ی اکسپشن‌های مدیریت نشده حتی نباید فکر کنند. تنها برنامه‌نویسان برنامه نیاز دارند درباره‌ی اکسپشن‌های مدیریت نشده نگران باشند و برنامه باید برای مواجهه با اکسپشن‌های مدیریت نشده سیاستی داشته باشد. مایکروسافت در واقع به برنامه‌نویسان برنامه توصیه می‌کند سیاست پیش فرض CLR را بپذیرند. آن اینست که وقتی یک برنامه یک اکسپشن مدیریت نشده را می‌گیرد، ویندوز یک گزارش در **event log** سیستم می‌نویسد. شما می‌توانید این ورودی را با بازکردن برنامه **Event Viewer** و سپس نگاه به **Windows Logs -> Application**، همانند شکل ۱-۲۰، ببینید.



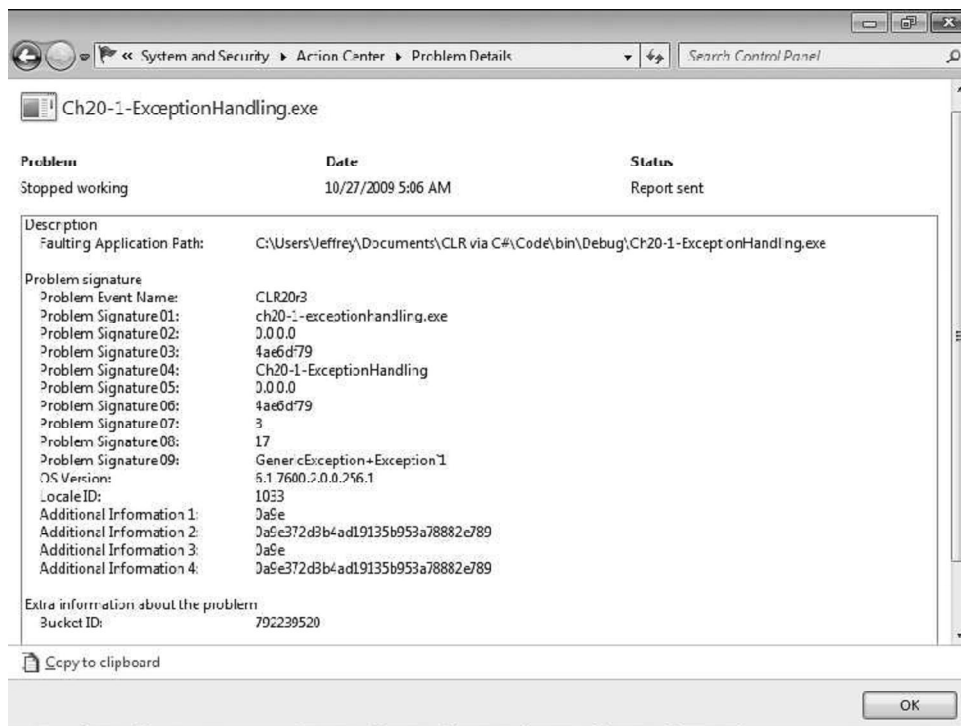
شکل ۱-۲۰ Windows Event log در حال نمایش یک برنامه که به خاطر یک اکسپشن مدیریت نشده بسته شده است

هر چند، شما می‌توانید جزئیات جالبتری درباره‌ی مشکل با استفاده از اپلت Windows Action Center بدست آورید. برای باز کردن Action Center، روی آیکن پرچم در کنار ساعت سیستم کلیک کنید، Open Action Center را انتخاب کنید، جعبه‌ی Maintenance را باز کنید، و سپس لینک "View reliability history" را انتخاب کنید. از اینجا، شما می‌توانید برنامه‌هایی که به خاطر یک اکسپشن مدیریت نشده از کار افتاده‌اند را در بخش پایانی ببینید، که در شکل ۲-۲۰ نشان داده شده است.



شکل ۲-۲۰ Reliability Monitor در حال نمایش یک برنامه که به خاطر یک اکسپشن مدیریت نشده از کار افتاده است.

برای دیدن جزئیات بیشتر درباره‌ی برنامه از کار افتاده، روی یک برنامه از کار افتاده در Reliability Monitor دابل کلیک کنید. جزئیات شبیه شکل ۳-۲۰ خواهد بود و معانی امضاها، مشکل‌ها، در جدول ۳-۲۰ توضیح داده شده است. تمام اکسپشن‌های مدیریت نشده که توسط برنامه‌های مدیریت شده تولید شده‌اند در قسمت CLR20r3 قرار داده می‌شود.



شکل ۳-۲۰ Reliability Monitor در حال نمایش جزئیات بیشتر درباره‌ی برنامه از کار افتاده به خاطر اکسپشن مدیریت نشده

جدول ۲-۲۰ امضاهای مشکل‌ها

امضای مشکل	توضیح *
01	نام فایل EXE (محدودیت ۳۲ کاراکتری)
02	شماره نسخه اسمبلی از فایل EXE
03	برچسب زمانی فایل EXE
04	نام کامل اسمبلی از فایل EXE (محدودیت ۶۴ کاراکتری)
05	نسخه اسمبلی شکست خورده
06	برچسب زمانی اسمبلی شکست خورده
07	متد و نوع اسمبلی شکست خورده. این مقدار یک علامت متادیتای MethodDef است (پس از حذف بایت بالای 0x06) که متدی که اکسپشن را تولید کرده شناسایی می‌کند. از روی این مقدار، شما می‌توانید با استفاده از ILDasm.exe برای تعیین نوع و متد متخلف استفاده کنید.
08	دستور IL متد شکست خورده. این مقدار یک آفست درون متد شکست خورده از دستور IL ای است که شما می‌توانید با استفاده از ILDasm.exe دستور متخلف را تعیین کنید.
09	نوع اکسپشن تولید شده (محدودیت ۳۲ کاراکتری)

* اگر رشته‌ای فرای محدودیت مجاز باشد، آنگاه کوتاه سازی‌های هوشمندانه‌ای مثل حذف "Exception" از نوع اکسپشن یا ".dll" از یک نام فایل صورت می‌گیرد. اگر رشته حاصل هنوز هم طولانی بود، آنگاه CLR یک مقدار با هش کردن یا اینکدینگ پایه ۶۴ رشته، تولید می‌کند.

پس از ضبط اطلاعات درباره‌ی برنامه‌ای که دچار مشکل شده، ویندوز پنجره‌ی پیامی را نشان می‌دهد که به کاربر نهایی اجازه می‌دهد اطلاعات پیرامون برنامه شکست خورده را به سرورهای مایکروسافت ارسال کند.^{۶۷} این، گزارش خطاهای ویندوز Windows Error Reporting نامیده می‌شود و اطلاعات بیشتر درباره‌ی آن در وب سایت (http://WinQual.Microsoft.com) Windows Quality قرار دارد.

^{۶۷} شما می‌توانید این پنجره پیام را با استفاده از P/Invoke و فراخوانی تابع SetErrorMode Win32 و ارسال SEM_NOGPFAULTERRORBOX به آن، غیرفعال کنید.

شرکت‌ها می‌توانند به انتخاب خود با مایکروسافت ثبت نام کنند تا بتوانند این اطلاعات را درباره‌ی برنامه‌ها و کامپوننت‌هایشان ببینند. ثبت نام رایگان است اما آن نیاز دارد که اسمبلی‌های شما با یک VeriSign ID (که Software Publisher's Digital ID for Authenticode نیز نامیده می‌شود) امضاء شده باشند.

طبیعتاً، شما می‌توانید سیستم خودتان را برای بدست آوردن اطلاعات اکسپشن‌های مدیریت نشده بسازید تا بتوانید مشکلات را در کدتان برطرف کنید. وقتی برنامه شما مقداردهی اولیه می‌شود، شما می‌توانید CLR را آگاه کنید که متدی دارید که می‌خواهید هرگاه هر تری در برنامه شما با یک اکسپشن مدیریت نشده مواجه می‌شود، این متد فراخوانی گردد.

متأسفانه، هر مدل برنامه‌ای که مایکروسافت تولید می‌کند روش خودش برای برخورد با اکسپشن‌های مدیریت نشده را دارد. اعضای که شما می‌خواهید در مستندات FCL نگاه کنید عبارتند از:

- برای هر برنامه‌ای، به رویداد **UnhandledException** از **System.AppDomain** نگاه کنید. برنامه‌های Silverlight با امنیتی که برای ثبت نام با این رویداد کافی باشد، اجرا نمی‌شوند.
- برای یک برنامه Windows Forms، به متد مجازی **OnThreadException** از **System.Windows.Forms.NativeWindow** و رویداد **ThreadException** از **System.Windows.Forms.Application** نگاه کنید.
- برای یک برنامه Windows Presentation Foundation (WPF) به رویداد **DispatcherUnhanledException** از **System.Windows.Application** و رویدادهای **UnhandledException** و **UnhandledExceptionFilter** از **System.Windows.Threading.Dispatcher** نگاه کنید.
- برای Silverlight، به رویداد **UnhandledException** از **System.Windows.Application** نگاه کنید.
- برای یک برنامه ASP.NET Web Form به رویداد **Error** از **System.Web.UI.TemplateControl** و **System.Web.UI.Page** و **System.Web.UI.UserControl** است. علاوه بر این، شما باید به رویداد **Error** از **System.Web.HttpApplication** هم نگاه کنید.
- برای یک برنامه Windows Communication Foundation (WCF) به ویژگی **ErrorHandlers** از **System.ServiceModel.Dispatcher.ChannleDispatcher** نگاه کنید.

قبل از آنکه من این بخش را ترک کنم، دوست دارم چند کلمه درباره‌ی اکسپشن‌های مدیریت نشده که می‌توانند در یک برنامه‌ی توزیع شده (distributed) مثل یک وب سایت یا یک وب سرویس رخ دهد، حرف بزنم. در یک دنیای ایده‌آل، یک برنامه سروری که با یک اکسپشن مدیریت نشده برخورد می‌کند، باید آن را گزارش کرده، به گونه‌ای به کلاینت خبر دهد که عمل درخواستی نتوانست کامل شود، سپس سرور باید از کار بیافتد. متأسفانه، ما در یک دنیای ایده‌آل زندگی نمی‌کنیم و بنابراین، شاید ممکن نباشد که خبر شکست را به کلاینت ارسال کنیم. برای برخی سرورهای عظیم (مثل Microsoft SQL Server) شاید عملی نباشد که سرور را از کار انداخته و یک نمونه جدید از آن شروع کنیم. برای یک برنامه سروری، اطلاعات درباره‌ی اکسپشن مدیریت نشده نباید به کلاینت برگردد چون یک کلاینت کار زیادی نمی‌تواند انجام دهد مخصوصاً اگر کلاینت توسط یک شرکت متفاوت پیاده‌سازی شده باشد. گذشته از این، سرور باید تا حد ممکن اطلاعات اندکی درباره‌ی خود به کلاینت‌هایش منتشر کند تا احتمال هک شدن سرور کاهش یابد.

نکته CLR برخی اکسپشن‌های تولیدی توسط کد اصلی (native code) را به عنوان اکسپشن‌های وضعیت خراب `corrupted state` (CSE) در نظر می‌گیرد چون آن‌ها معمولاً نتیجه‌ی وجود یک اشکال در خود CLR یا کد اصلی‌ای می‌باشند که برنامه‌نویس مدیریت‌شده کنترلی روی آن‌ها ندارد. به صورت پیش فرض، CLR اجازه نمی‌دهد کد مدیریت‌شده این اکسپشن‌ها را بگیرد و بلوک‌های **finally** اجرا نخواهند شد. لیست اکسپشن‌های اصلی Win32 که به عنوان CSE در نظر گرفته می‌شوند:

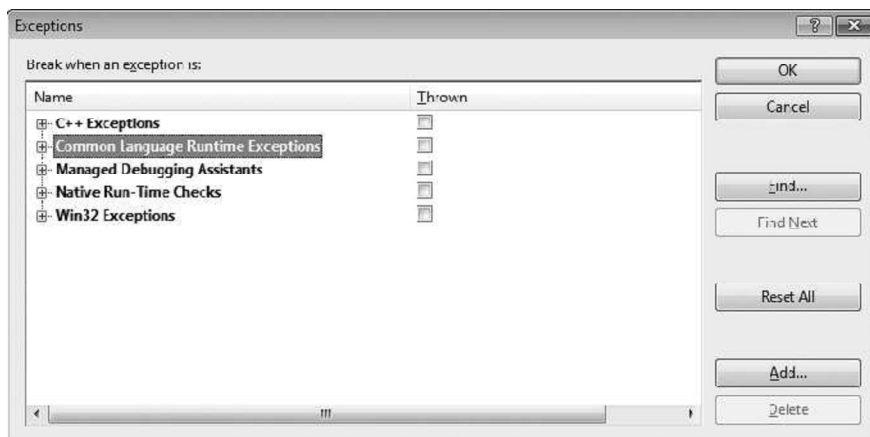
EXCEPTION_ACCESS_VIOLATION	EXCEPTION_STACK_OVERFLOW
EXCEPTION_ILLEGAL_INSTRUCTION	EXCEPTION_IN_PAGE_ERROR
EXCEPTION_INVALID_DISPOSITION	EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_PRIV_INSTRUCTION	STATUS_UNWIND_CONSOLIDATE

متدهای مدیریت‌شده تکی می‌توانند پیش فرض را بازنویسی کرده و این اکسپشن‌ها را با اعمال **System.Runtime.ExceptionServices.HandlerProcessCorruptedStateExceptionsAttribute** بر متد، بگیرند. به علاوه، **System.Security.SecurityCriticalAttribute** بر متد باید اعمال شده باشد. شما همچنین می‌توانید با تنظیم عنصر **legacyCorruptedStateExceptionPolicy** به `true` در فایل تنظیمات XML برنامه، پیش فرض را برای کل پردازش بازنویسی کنید. CLR اغلب این‌ها را به یک شی **System.Runtime.InteropServices.SEHException** تبدیل می‌کند به جز **EXCEPTION_ACCESS_VIOLATION** که به یک شی **System.AccessViolationException** و **EXCEPTION_STACK_OVERFLOW** که به یک شی **System.StackOverflowException** تبدیل می‌شود.

نکته درست قبل از فراخوانی یک متد، شما باید برای فضای پشته کافی با فراخوانی متد **EnsureSufficientExecutionStack** از کلاس **RuntimeHelper** مطمئن شوید. این متد بررسی می‌کند آیا ترد فراخوانی کننده فضای پشته کافی برای اجرای متد توسط (که خیلی خوب تعریف نشده است) را دارد. اگر فضای پشته کافی موجود نباشد، متد یک **InsufficientExecutionStackException** که شما می‌توانید آن را بگیرید، تولید می‌کند. متد **EnsureSufficientExecutionStack** هیچ آرگومانی نمی‌گیرد و `void` برمی‌گرداند. این متد نوعاً توسط متدهای بازگشتی استفاده می‌شود.

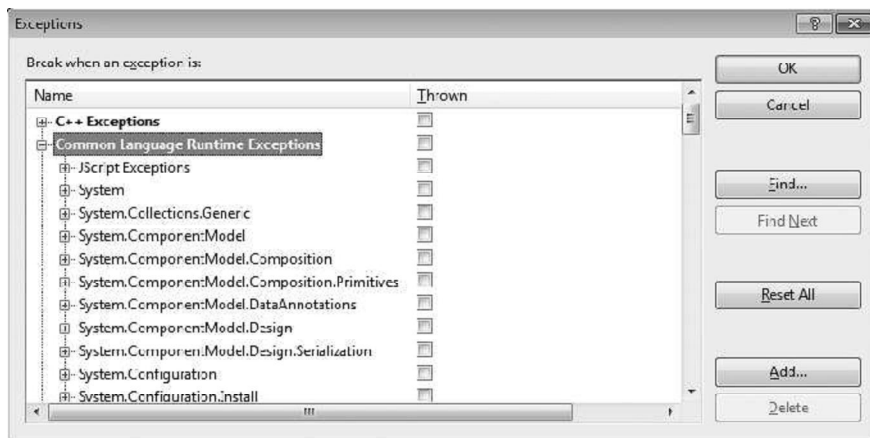
خطایابی اکسپشن‌ها

دیباگر ویژوال استودیو پشتیبانی ویژه‌ای برای اکسپشن‌ها دارد. وقتی یک **Solution** باز است، از منوی **Debug**، **Exceptions** را انتخاب کنید و شما پنجره محاوره‌ای نمایش داده شده در شکل ۴-۲۰ را خواهید دید.



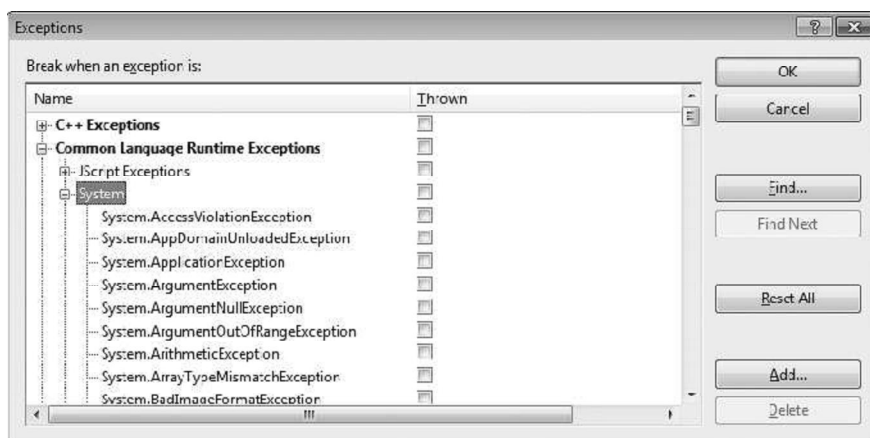
شکل ۴-۲۰ جعبه محاوره‌ای Exceptions در حال نمایش گونه‌های مختلف اکسپشن‌ها

این جعبه محاوره گونه‌های مختلف اکسپشن‌ها که ویژوال استودیو از آن آگاه است را نمایش می‌دهد. برای Common Language Runtime Exceptions باز کردن این شاخه که در شکل ۵-۲۰ نمایش داده شده است، مجموعه از فضاهای نام که دیباگر ویژوال استودیو از آن آگاه است را نشان می‌دهد.



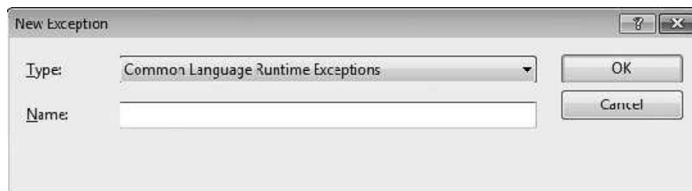
شکل ۵-۲۰ جعبه محاوره Exceptions در حال نمایش اکسپشن‌های CLR همراه با فضای نام

اگر شما یک فضای نام را گسترش دهید، شما تمام نوع‌های مشتق شده از **System.Exception** که درون آن فضای نام تعریف شده‌اند را خواهید دید. برای مثال، شکل ۶-۲۰ آنچه شما خواهید دید اگر فضای نام **System** را باز کنید، نشان می‌دهد.



شکل ۶-۲۰ جعبه محاوره Exceptions در حال نمایش اکسپشن‌های CLR که در فضای نام System تعریف شده‌اند

برای هر نوع اکسپشن اگر تیک **Thrown** آن خورده باشد، دیباگر به محضی که اکسپشن تولید شود، متوقف خواهد شد. در این لحظه، CLR برای یافتن بلوک‌های **catch** مطابق، تلاش نکرده است. این مفید است اگر شما می‌خواهید کدتان را که یک اکسپشن گرفته و مدیریت می‌کند، خطایابی کنید. همچنین مفید است اگر شک دارید که یک کامپوننت یا کتابخانه ممکن است اکسپشن‌ها را بلعیده یا بازتولید کند و شما مطمئن نیستید که دقیقاً کجا یک نقطه توقف (breakpoint) قرار دهید تا آن را در حین عمل بگیرید. اگر تیک **Thrown** برای یک نوع اکسپشن انتخاب نشده باشد، دیباگر وقتی اکسپشن تولید می‌شود هم متوقف می‌گردد اما تنها اگر نوع اکسپشن، مدیریت شده نباشد. برنامه‌نویسان اغلب تیک **Thrown** را خالی می‌کنند چون یک اکسپشن مدیریت شده بیان می‌کند که برنامه انتظار این وضعیت را داشته و با آن مواجه می‌شود؛ برنامه به صورت عادی به اجرا ادامه می‌دهد. اگر شما نوع‌های اکسپشن خودتان را تعریف کنید، شما می‌توانید آن را به این جعبه محاوره با کلیک بر **Add** اضافه کنید. این باعث می‌شود جعبه محاوره در شکل ۷-۲۰ ظاهر شود.



شکل ۷-۲ آگاه کردن ویژوال استودیو از نوع اکسپشن خودتان: پنجره محاوره New Exception

در این جعبه محاوره، شما ابتدا نوع اکسپشن را Common Language Runtime Exceptions انتخاب می‌کنید و سپس، شما می‌توانید نام کامل نوع اکسپشن خود را وارد کنید. توجه کنید نوعی که شما وارد می‌کنید مجبور نیست یک نوع مشتق شده از **System.Exception** باشد. نوع‌های غیر از سازگار با CLS کاملاً پشتیبانی می‌شوند. اگر شما دو یا بیشتر نوع با نام یکسان اما در اسمبلی‌های مختلف داشتید، راهی برای تمایز بین نوع‌های یکی از دیگری وجود ندارد. خوشبختانه این وضعیت به ندرت رخ می‌دهد.

اگر اسمبلی شما چندین نوع اکسپشن تعریف می‌کند، شما باید آن‌ها را در یک زمان وارد کنید. در آینده، من دوست دارم جعبه پیام را به گونه‌ای ببینم که به من اجازه دهد یک اسمبلی را جستجو کنم و تمام نوع‌های مشتق شده از **Exception** را به صورت خودکار به دیباگر ویژوال استودیو وارد کند. آنگاه هر نوع می‌تواند با اسمبلی نیز شناسایی شود که مشکل داشتن دو نوع با نام یکسان در اسمبلی‌های متفاوت را حل می‌کند.

ملاحظات عملکردی مدیریت اکسپشن

جامعه برنامه‌نویسان فعالانه عملکرد اکسپشن‌ها را به بحث می‌گذارند. برخی افراد مدعی‌اند که عملکرد مدیریت اکسپشن آنقدر بد است که آن‌ها حتی استفاده از مدیریت اکسپشن را رد می‌کنند. هر چند، من ادعا می‌کنم در پلتفرم شی‌گرا، مدیریت اکسپشن یک انتخاب نیست، آن اجبار است. و گذشته از آن، اگر شما از آن استفاده نکنید، به جای آن از چه چیزی استفاده خواهید کرد؟ آیا می‌خواهید متدهای **false/true** برگردانند تا موفقیت/شکست را نشان دهند یا شاید یک کد خطای نوع **enum**؟ خوب، اگر شما این کار را کردید، شما بدترین هر دو را دارید: **CLR** و کد کتابخانه کلاس اکسپشن تولید خواهند کرد و کد شما کدهای خطا برمی‌گرداند. شما مجبورید اکنون با هر دوی این‌ها در کدتان مواجه شوید.

سخت است که بین عملکرد مدیریت اکسپشن و روش‌های معمولی تر گزارش اکسپشن (مثل **HRESULT**، کدهای برگشتی خاص و...) مقایسه کرد. اگر شما کدی بنویسید که مقدار برگشتی هر فراخوانی متد را بررسی کرده و مقدار برگشتی را تا فراخوانی کننده‌های خودتان فیلتر کنید، عملکرد برنامه شما به شدت تحت تاثیر قرار می‌گیرد. اما با در نظر نگرفتن عملکرد، حجم کدنویسی اضافی که باید انجام دهید و احتمال خطاها، به شدت بالاست وقتی کدی می‌نویسید که مقدار برگشتی هر متد را بررسی می‌کند. مدیریت اکسپشن جایگزین بسیار بهتری است.

هر چند، مدیریت اکسپشن یک هزینه دارد: کامپایلرهای **C++** مدیریت نشده باید کدی تولید کنند که ردیابی کند چه اشیایی با موفقیت ساخته شده‌اند. کامپایلر همچنین باید کدی تولید کند که وقتی اکسپشن گرفته شد، مخرب هر شی که با موفقیت ساخته شده، را نیز فراخوانی کند. عالی است که کامپایلر این بار را به دوش می‌کشد اما مقدار زیادی کد ساماندهی در برنامه شما تولید می‌کند که بر اندازه کد و زمان اجرا اثر نامطلوب می‌گذارد.

در سوی دیگر، کامپایلرهای مدیریت شده کار بسیار ساده تری دارند چون اشیاء مدیریت شده در هیپ مدیریت شده تخصیص می‌یابند که توسط جمع‌آوری کننده زباله در حال مانیتورینگ است. اگر یک شی با موفقیت ساخته شود و یک اکسپشن تولید گردد، جمع‌آوری کننده زباله سرانجام حافظه شی را آزاد می‌کند. کامپایلرها نیاز ندارند هیچ گونه کد ساماندهی برای ردیابی اشیایی که با موفقیت ساخته می‌شوند، تولید کنند و نیاز ندارند مطمئن شوند که یک مخرب فراخوانی شده است. در مقایسه با **C++** مدیریت نشده، این یعنی کد کمتری توسط کامپایلر تولید می‌شود و کد کمتر که در زمان اجرا، اجرا شود منجر به عملکرد بهتر برنامه شما می‌شود.

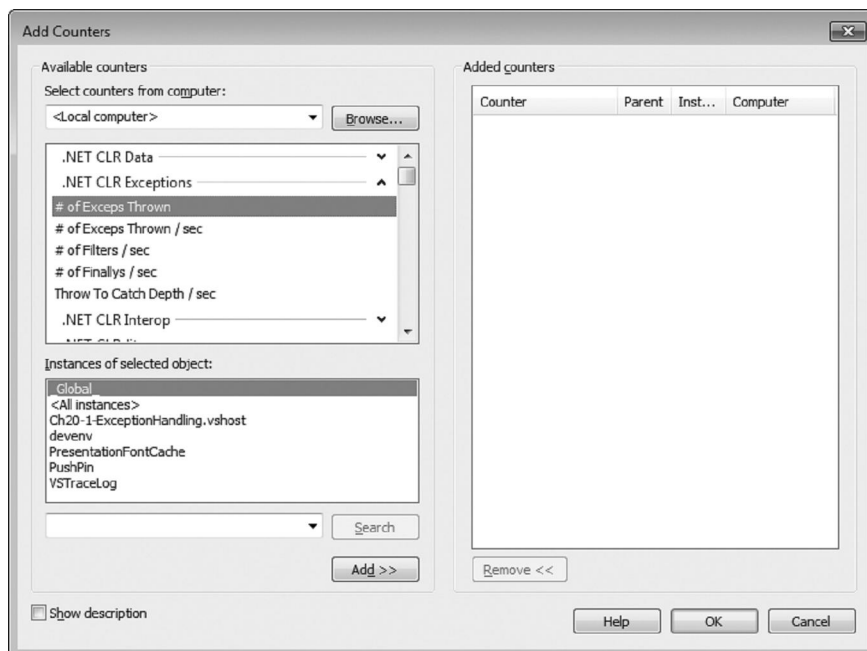
در طول سال‌ها، من از مدیریت اکسپشن در زبان‌های برنامه‌نویسی مختلف، سیستم عامل‌های مختلف و معماری پردازنده‌های مختلف استفاده کرده‌ام. در هر مورد، مدیریت اکسپشن متفاوت پیاده‌سازی شده است که هر پیاده‌سازی در رابطه با عملکرد، نقاط ضعف و قوت خودش را دارد. برخی پیاده‌سازی‌ها، ساخت-های مدیریت اکسپشن را مستقیماً در یک متد کامپایل می‌کند در حالیکه دیگر پیاده‌سازی‌ها اطلاعات مربوط به مدیریت اکسپشن را در یک داده‌ای همراه با متد ذخیره می‌کند - این جدول تنها اگر یک اکسپشن تولید شود، مورد دسترسی قرار می‌گیرد. برخی کامپایلرها نمی‌توانند متدهایی که حاوی مدیریت کننده اکسپشن هستند را خطی کنند و برخی کامپایلرها اگر متد، حاوی مدیریت کننده‌های اکسپشن باشد، متغیرها را ذخیره نمی‌کنند.

نکته اینست که شما نمی‌توانید تعیین کنید چه مقدار کد اضافی به برنامه شما هنگام استفاده از مدیریت اکسپشن اضافه می‌شود. در دنیای مدیریت شده، این حتی مشکل تر است چون کد اسمبلی شما می‌تواند در هر پلتفرمی که دات‌نت فریمورک را پشتیبانی کند اجرا شود پس کد تولیدی توسط کامپایلر JIT برای اداره کردن مدیریت اکسپشن وقتی اسمبلی شما روی یک ماشین **x86** اجرا می‌شود، بسیار متفاوت از کد تولیدی توسط کامپایلر JIT خواهد بود وقتی کد

شما روی یک پردازنده x64 یا IA64 اجرا می‌شود. همچنین، کامپایلرهای JIT که با دیگر پیاده‌سازی‌های CLR همراهند (مثل .NET Compact Framework میکروسافت یا پروژه متن باز Mono) نیز احتمالاً کد متفاوتی تولید می‌کنند.

در حقیقت، من توانسته‌ام بخشی از کدهای خودم را با تعدادی از کامپایلرهای JIT که میکروسافت (به صورت داخلی) دارد، آزمایش کنم و تفاوت در عملکرد که من مشاهده کردم بسیار شگفت‌انگیز بود. نکته اینست که شما باید کدتان را در پلتفرم‌های مختلف که انتظار دارید کاربران کد را در آن‌ها اجرا کنند، آزمایش کنید و تغییرات را بر اساس آن‌ها اعمال کنید. مجدداً، من درباره‌ی استفاده از مدیریت اکسپشن نگران نخواهم بود چون فواید آن بسی بیشتر از اثرات منفی عملکردی آن است.

اگر شما علاقه مند بودید ببینید چگونه مدیریت اکسپشن بر عملکرد کدتان اثرگذار است شما می‌توانید از ابزار Performance Monitor که با ویندوز عرضه می‌شود استفاده کنید. شکل ۸-۲۰ شمارنده‌های مرتبط با اکسپشن که همراه با دات‌نت فریمورک نصب شده‌اند را نشان می‌دهد.



شکل ۸-۲۰ Performance Monitor در حال نمایش شمارنده‌های .NET CLR Exceptions.

به ندرت، شما به متدی می‌رسید که مکرراً آن را فراخوانی می‌کنید که نرخ شکست بالایی نیز دارد. در این وضعیت، ضربه عملکردی ناشی از اکسپشن‌های تولیدی می‌تواند غیرقابل تحمل باشد. برای مثال، میکروسافت از چندین تن از مشتریان شنید که آن‌ها متد **Parse** از **Int32** را فراخوانی می‌کنند و مکرراً داده‌های ورودی از کاربر نهایی را ارسال می‌کنند که نمی‌تواند آن‌ها را تجزیه کند. چون **Parse** مکرراً فراخوانی می‌شود، ضربه عملکردی ناشی از تولید و گرفتن اکسپشن‌ها بر عملکرد کلی برنامه عوارض سنگینی داشت.

برای آنکه مشتریان را پاسخ داده و تمام راهنمایی‌های توضیح داده شده در این فصل را رعایت کند، میکروسافت یک متد جدید به کلاس **Int32** افزود. این متد جدید **TryParse** نامیده می‌شود و دو سربارگذاری شبیه به این دارد:

```
public static Boolean TryParse(String s, out Int32 result);
public static Boolean TryParse(String s, NumberStyles styles,
    IFormatProvider, provider, out Int32 result);
```

شما متوجه خواهید شد که این متدها یک **Boolean** برمی‌گردانند که نشان می‌دهد آیا **String** ارسال شده حاوی کاراکترهایی است که بتواند به یک **Int32** تبدیل شود. این متدها همچنین یک پارامتر خروجی به نام **result** برمی‌گردانند. اگر متد **true** برگرداند، **result** حاوی نتیجه‌ی تجزیه رشته به یک عدد صحیح ۳۲ بیتی است. اگر متدها **false** برگردانند، **result** حاوی **0** خواهد بود، اما شما نباید هر کدی که به آن نگاه می‌کند را اجرا کنید.

یک چیز که می‌خواهم کاملاً روشن کنم: مقدار برگشتی **Boolean** از یک متد **TryXxx**، **false** برمی‌گرداند تا یک و فقط یک نوع از شکست را نشان دهد. متد هنوز برای هر نوع دیگر شکست باید اکسپشن تولید کند. برای نمونه، **TryParse** از **Int32** یک **ArgumentException** تولید می‌کند اگر فرم آرگومان معتبر نباشد و مطمئناً هنوز امکان دارد هنگام فراخوانی **TryParse**، یک **OutOfMemoryException** تولید شود.

من همچنین می‌خواهم این را روشن کنم که برنامه‌نویسی شی‌گرا به برنامه‌نویس اجازه می‌دهد بهره‌ور باشد. یک روش که این را انجام می‌دهد، در معرض قرار ندادن کدهای خطا برای اعضای یک نوع است. به بیان دیگر، سازنده‌ها، متدها، ویژگی و غیره همه با این نظر تعریف شده‌اند که شکست نخواهند خورد و اگر به درستی تعریف شوند، برای اغلب استفاده‌های یک عضو، آن شکست نخواهد خورد و ضربه عملکردی در کار نخواهد بود چرا که اکسپشنی تولید نخواهد شد.

هنگام تعریف نوع‌ها و اعضایشان، شما باید اعضا را به گونه‌ای تعریف کنید که این بعید باشد برای سناریوهای رایج که انتظار دارید نوع‌هایتان در آن‌ها استفاده شوند، آن‌ها شکست بخورند. اگر بعدها از کاربران بشنوید که از عملکرد نوع به خاطر اکسپشن‌های تولیدی ناراضی هستند، آنگاه و فقط آنگاه شما باید به فکر افزودن متدهای **TryXxx** بیافتید. به بیان دیگر، شما باید بهترین مدل شی را ابتدا تولید کنید و سپس اگر کاربران آن را پس دادند، تعدادی متدهای **TryXxx** به نوعتان بیافزایید تا کاربرانی که با مشکل عملکردی مواجه هستند، از آن استفاده کنند. کاربرانی که با مشکل عملکردی مواجه نیستند به استفاده از نسخه‌های غیر **TryXxx** از متدها ادامه دهند، چون این مدل شی، بهتر است.

نواحی اجرایی محدود شده (CERs) Constrained Execution Regions

بسیاری از برنامه‌های نیاز ندارند قدرتمند بوده و از هر و همه‌ی شکست‌ها احیا شوند. این برای بسیاری برنامه‌های کلاینت مثل **Notepad.exe** و **Calc.exe** صحیح است. و البته، بسیاری از ما دیده ایم برنامه‌های **Microsoft Office** مثل **WinWord.exe**، **Excel.exe**، **Outlook.exe** به خاطر اکسپشن‌های مدیریت نشده از کار می‌افتند. همچنین، بسیاری از برنامه‌های سمت سروری مثل وب سرورها بی‌وقارند و اگر به خاطر یک اکسپشن مدیریت نشده شکست بخورند به صورت خودکار مجدداً اجرا می‌شوند. البته برخی سرورها مثل **SQL Server** مدیریت وضعیت دارند و از دست رفتن داده‌ها به خاطر یک اکسپشن مدیریت نشده احتمالاً بسیار فاجعه آمیز خواهد بود.

در **CLR**، ما **AppDomain** ها (بحث شده در فصل ۲۲) را داریم که حاوی وضعیت هستند. وقتی یک **AppDomain** تخلیه (**unload**) می‌شود، تمام وضعیت آن تخلیه می‌شود. و بنابراین اگر یک ترد در یک **AppDomain** با یک اکسپشن مدیریت نشده برخورد کند، مشکلی نیست اگر **AppDomain** تخلیه شود (که تمام وضعیت آن را از بین خواهد برد) بدون آنکه کل پردازش از کار بیاندازد.^{۶۸}

طبق تعریف، یک ناحیه اجرایی محدود شده **CER** یک بلوک از کد است که باید نسبت به شکست انعطاف پذیر باشد. چون **AppDomain** ها قابل تخلیه شدن هستند، وضعیت آن‌ها از بین می‌رود، **CER** ها معمولاً برای نگهداری هر وضعیتی که میان چندین **AppDomain** یا پردازش مشترک است، استفاده می‌شوند. **CER** ها هنگام سعی در حفظ وضعیت وقتی به صورت غیرمنتظره اکسپشن تولید می‌شود، مفیدند. گاهی ما به این گونه از اکسپشن‌ها، اکسپشن‌های غیرهمزمان **asynchronous exceptions** می‌گوییم. برای مثال، هنگام فراخوانی یک متد، **CLR** مجبور است یک اسمبلی را بارگذاری کرده، یک شی نوع در هیپ بارگذاری کننده‌ی **AppDomain** بسازد، سازنده‌ی استاتیک نوع را فراخوانی کند، **IL** را به کد اصلی **JIT** کند و غیره. هر یک از این عملیات‌ها می‌توانند شکست بخورند و **CLR** شکست را با تولید یک اکسپشن گزارش می‌کند.

اگر هر یک از این عملیات‌ها درون یک بلوک **catch** یا **finally** شکست بخورند، آنگاه کد احیا یا پاکسازی شما به صورت کامل اجرا نخواهد شد. در اینجا مثالی است که مشکل احتمالی را به نمایش می‌گذارد.

```
private static void Demo1() {
    try {
        Console.WriteLine("In try");
    }
    finally {
        // Type1's static constructor is implicitly called in here
        Type1.M();
    }
}

private sealed class Type1 {
    static Type1() {
```

^{۶۸} این کاملاً درست است اگر ترد تمام زندگی‌اش را در یک تک **AppDomain** (مثل سناریوهای **ASP.NET** و رویه‌های ذخیره شده مدیریت شده **SQL Server**) به سر کند. اما شما ممکن است مجبور باشید کل پردازش را از بین ببرید اگر یک ترد در طول عمرش از مرزهای **AppDomain** عبور کند.

```

        // if this throws an exception, M won't get called
        Console.WriteLine("Type1's static ctor called");
    }

    public static void M() { }
}

```

وقتی من کد فوق را اجرا می‌کنم، من خروجی زیر را می‌گیرم:

```

In try
Type1's static ctor called

```

آنچه می‌خواهیم اینست که حتی شروع به اجرای کد درون بلوک **try** نکنیم مگر آنکه بدانیم که کد درون بلوک **catch** و **finally** همراه آن، تضمینی (یا تا حد ممکن تضمینی) اجرا می‌شود. ما می‌توانیم این کار را با تغییر کد فوق بدین صورت انجام دهیم:

```

private static void Demo2() {
    // Force the code in the finally to be eagerly prepared
    RuntimeHelpers.PrepareConstrainedRegions();
    // System.Runtime.CompilerServices namespace
    try {
        Console.WriteLine("In try");
    }
    finally {
        // Type2's static constructor is implicitly called in here
        Type2.M();
    }
}

```

```

public class Type2 {
    static Type2() {
        Console.WriteLine("Type2's static ctor called");
    }

    // Use this attribute defined in the System.Runtime.ConstrainedExecution namespace
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    public static void M() { }
}

```

حال وقتی من این نسخه از کد را اجرا می‌کنم، من خروجی زیر را می‌گیرم:

```

Type2's static ctor called
In try

```

متد **PrepareConstrainedRegions** متد بسیار ویژه‌ای است. وقتی کامپایلر JIT می‌بیند این متد بلافاصله قبل از یک بلوک **try** فراخوانی شده است، مشتاقانه کد درون بلوک‌های **catch** و **finally** که همراه **try** هستند را کامپایل می‌کند. کامپایلر JIT هر اسمبلی را بارگذاری کرده، هر شی نوع را ساخته، هر سازنده استاتیک را فراخوانی کرده و هر متد را JIT می‌کند. اگر هر یک از این عملیات‌ها منجر به یک اکسپشن شود، آنگاه اکسپشن قبل از آنکه ترد وارد بلوک **try** شود، رخ داده است. وقتی کامپایلر JIT مشتاقانه متدها را آماده می‌کند، آن همچنین در تمام گراف فراخوانی حرکت کرده و مشتاقانه متدهای فراخوانی شده را آماده می‌کند. هرچند، کامپایلر JIT تنها متدهایی که **ReliabilityContractAttribute** با **Consistency.WillNotCorruptState** یا **Consistency.MayCorruptInstance** بر آن اعمال شده است را آماده می‌کند چون CLR درباره‌ی متدهایی که ممکن است **AppDomain** یا وضعیت پردازش را خراب کنند نمی‌تواند تضمینی بکند. درون یک بلوک **catch** یا **finally** که شما با فراخوانی **PrepareConstrainedRegions** از آن محافظت می‌کنید، شما می‌خواهید مطمئن شوید همانطور که توضیح دادم تنها متدهایی که **ReliabilityContractAttribute** بر آن‌ها اعمال شده است را فراخوانی کنید.

ReliabilityContractAttribute شبیه به این است:

```
public sealed class ReliabilityContractAttribute : Attribute {
    public ReliabilityContractAttribute(Consistency consistencyGuarantee, Cer cer);
    public Cer Cer { get; }
    public Consistency ConsistencyGuarantee { get; }
}
```

این صفت به یک برنامه‌نویس اجازه می‌دهد قرارداد قابلیت اطمینان از یک متد خاص^{۶۹} را برای فراخوانی کننده‌های احتمالی متد، مستندسازی کند. هر دو نوع **Cer** و **Consistency** نوع‌های شمارشی هستند که اینگونه تعریف شده اند:

```
enum Consistency {
    MayCorruptProcess, MayCorruptAppDomain, MayCorruptInstance, WillNotCorruptState
}
```

```
enum Cer { None, MayFail, Success }
```

اگر متدی که شما می‌نویسید قول دهد که هیچ گونه وضعیتی را خراب نکند، از **Consistency.WillNotCorruptState** استفاده کنید. در غیر این صورت مدتان را با استفاده از یکی از سه حالت ممکن که با وضعیتی که متد شما ممکن است آن را تخریب کند مطابقت دارد، مستندسازی کنید. اگر متدی که شما می‌نویسید قول دهد شکست نخورد، از **Cer.Success** استفاده کنید. در غیر این صورت از **Cer.MayFail** استفاده کنید. هر متدی که **ReliabilityContractAttribute** بر آن اعمال نشده باشد معادل این است که اینگونه علامت زده شده باشد:

```
[ReliabilityContract(Consistency.MayCorruptProcess, Cer.None)]
```

مقدار **Cer.None** بیان می‌کند که متد هیچ تضمین CER ای نمی‌کند. به بیان دیگر، با در نظر گرفتن CER، نوشته نشده است. بنابراین، ممکن است شکست بخورد و ممکن است شکست خود را گزارش نکند یا نکند. به خاطر داشته باشید اغلب این تنظیمات به یک متد راهی می‌دهند که آنچه به فراخوانی کننده‌های احتمالی ارائه می‌کند را مستندسازی کند تا آن‌ها بدانند انتظار چه چیزی را داشته باشند. CLR و کامپایلر JIT از این اطلاعات استفاده نمی‌کنند.

وقتی شما می‌خواهید یک متد قابل اعتماد بنویسید، آن را کوچک ساخته و آنچه انجام می‌دهد را محدود کنید. مطمئن شوید که هیچ شی‌ای را تخصیص نمی‌دهد (برای مثال، هیچ بسته‌بندی وجود نداشته باشد)، هیچ متد مجازی یا متد رابط را فراخوانی نمی‌کند یا از هر نماینده‌ای یا رفلکشن استفاده نمی‌کند چون کامپایلر JIT نمی‌تواند بگوید چه متدی فراخوانی می‌شود. هر چند، شما می‌توانید با فراخوانی یکی از سه متد تعریف شده توسط کلاس **RuntimeHelpers** این متدها را آماده کنید:

```
public static void PrepareMethod(RuntimeMethodHandle method)
public static void PrepareMethod(RuntimeMethodHandle method,
    RuntimeTypeHandle[] instantiation)
public static void PrepareDelegate(Delegate d);
public static void PrepareContractedDelegate(Delegate d);
```

توجه کنید که کامپایلر و CLR هیچ گونه کاری انجام نمی‌دهند که بازبینی کنند متدهایی که شما نوشته‌اید در حقیقت به اندازه‌ای که از طریق **ReliabilityContractAttribute** تعیین نموده‌اید، تضمینی کار کنند. اگر شما کار اشتباهی انجام دهید، خراب شدن وضعیت ممکن است.

نکته حتی اگر تمام متدها مشتاقانه آماده شوند، یک فراخوانی متد هنوز هم می‌تواند منجر به یک **StackOverflowException** شود. وقتی CLR میزبانی نمی‌شود، یک **StackOverflowException** باعث می‌شود پردازش به وسیله فراخوانی **Environment.FailFast** توسط CLR بلافاصله بسته شود. وقتی میزبانی می‌شود، متد **PreparedConstrainedRegions** پشته را بررسی می‌کند تا ببیند آیا تقریباً ۴۸ کیلوبایت فضای پشته باقی مانده است. اگر فضای پشته محدود باشد، **StackOverflowException** قبل از ورود به بلوک **try** رخ می‌دهد.

شما همچنین باید به متد **ExecuteCodeWithGuaranteedCleanup** از **RuntimeHelpers** که راه دیگری برای اجرای کد با پاکسازی تضمین شده، است، نگاه کنید:

```
public static void ExecuteCodeWithGuaranteedCleanup(TryCode code, CleanupCode backoutCode,
```

^{۶۹} شما می‌توانید این صفت را به یک رابط، یک سازنده، یک ساختار، یک کلاس یا یک اسمبلی اعمال کنید تا بر اعضای درون آن‌ها اثر بگذارد.

```
Object userData);
```

هنگام فراخوانی این متد، شما بدنه‌ی بلوک **try** و **finally** را به عنوان متدهای کالبی که فرم کلی آن‌ها به ترتیب با دو نماینده زیر مطابق است، ارسال می‌کنید:

```
public delegate void TryCode(Object userData);
public delegate void CleanupCode(Object userData, Boolean exceptionThrown);
```

و سرانجام، راه دیگری برای اجرای تضمین شده‌ی کد، استفاده از کلاس **CriticalFinalizerObject** است که من در فصل ۲۱ آن را با جزئیات توضیح می‌دهم.

قراردادهای کد

قراردادهای کد **Code Contracts** روشی به شما ارائه می‌کنند که به صورت اعلانی تصمیمات طراحی که درباره‌ی کد، درون خود کد گرفته اید را مستندسازی کنید. قراردادهای کد به فرم زیر هستند:

- **پیش شرطها Preconditions** معمولاً برای اعتبارسنجی آرگومان‌ها استفاده می‌شوند.
- **پس شرطها Postconditions** برای اعتبارسنجی وضعیت وقتی یک متد به خاطر یک برگشت معمولی یا به خاطر تولید یک اکسپشن، اتمام می‌یابد، استفاده می‌شوند.
- **ثابت‌های شی Object Invariants** برای حصول اطمینان از اینکه فیلدهای یک شی در کل دوره حیات یک شی در وضعیت خوبی باقی بمانند، استفاده می‌شوند.

قراردادهای کد استفاده، درک، تکامل، آزمایش^{۷۰}، مستندسازی و خطایابی اولیه را آسان می‌کنند. شما می‌توانید به پیش شرطها، پس شرطها و ثابت‌های شی به عنوان بخش‌هایی از امضای یک متد نگاه کنید. همین طور، شما می‌توانید یک قرارداد را با یک نسخه جدید از کدتان، آسانتر کنید اما شما نمی‌توانید بدون از بین بردن سازگاری با نسخه‌های پیشین یک قرارداد را در نسخه جدید سختگیرانه تر کنید.

در قلب قراردادهای کد، کلاس **System.Diagnostics.Contracts.Contract** قرار دارد:

```
public static class Contract {
    // Precondition methods: [Conditional("CONTRACTS_FULL")]
    public static void Requires(Boolean condition);
    public static void EndContractBlock();

    // Preconditions: Always
    public static void Requires<TException>(Boolean condition)
        where TException : Exception;

    // Postcondition methods: [Conditional("CONTRACTS_FULL")]
    public static void Ensures(Boolean condition);
    public static void EnsuresOnThrow<TException>(Boolean condition)
        where TException : Exception;

    // Special Postcondition methods: Always
    public static T Result<T>();
    public static T OldValue<T>(T value);
    public static T ValueAtReturn<T>(out T value);

    // Object Invariant methods: [Conditional("CONTRACTS_FULL")]
    public static void Invariant(Boolean condition);
```

^{۷۰} برای کمک به آزمایش‌های خودکار، ابزار **Pex** که توسط **Microsoft Research** ساخته شده است را ببینید.

```

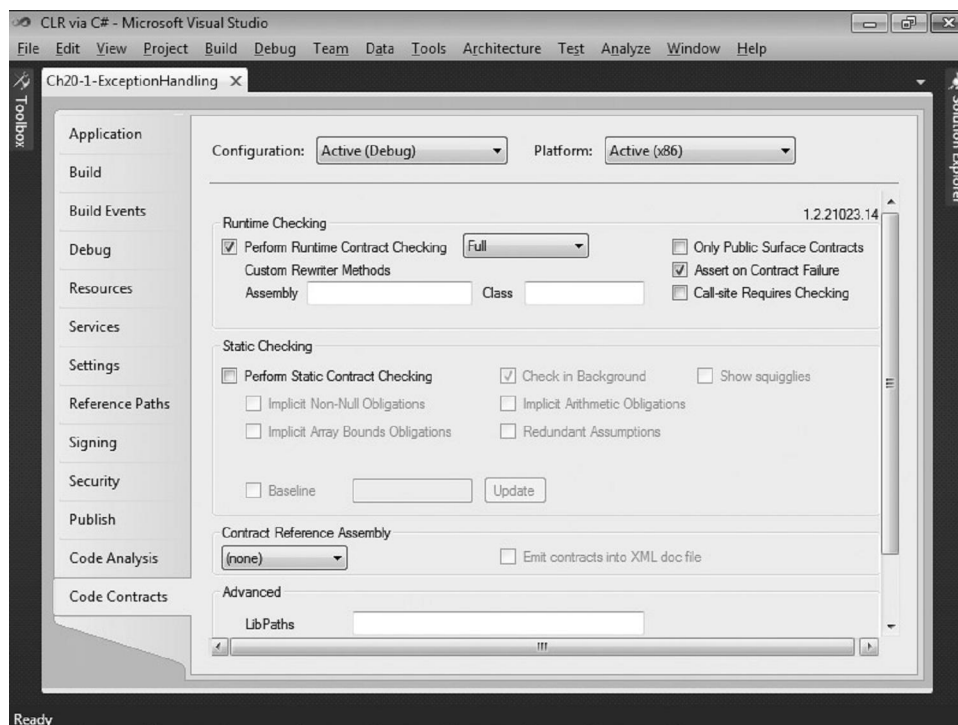
// Quantifier methods: Always
public static Boolean Exists<T>(IEnumerable<T> collection, Predicate<T> predicate);
public static Boolean Exists(Int32 fromInclusive, Int32 toExclusive,
    Predicate<Int32> predicate);
public static Boolean ForAll<T>(IEnumerable<T> collection, Predicate<T> predicate);
public static Boolean ForAll(Int32 fromInclusive, Int32 toExclusive,
    Predicate<Int32> predicate);
// Helper methods: [Conditional("CONTRACTS_FULL")] or [Conditional("DEBUG")]
public static void Assert(Boolean condition);
public static void Assume(Boolean condition);

// Infrastructure event: usually your code will not use this event
public static event EventHandler<ContractFailedEventArgs> ContractFailed;
}

```

همانگونه که در بالا بیان شد، بسیاری از این متدهای استاتیک، صفت **[Conditional("CONTRACTS_FULL")]** بر آن‌ها اعمال شده است. برخی از متدهای کمکی همچنین صفت **[Conditional("DEBUG")]** بر آن‌ها اعمال شده است. این یعنی کامپایلر هر کدی که این متدها را فراخوانی کند، نادیده می‌گیرد مگر آنکه هنگام کامپایلر کدتان، نماد مناسب تعریف شده باشد. هر متدی که با **"Always"** علامت زده شده باشد یعنی کامپایلر همیشه کد برای فراخوانی متد را تولید می‌کند. همچنین، متدهای **Requires**، **Requires<TException>**، **Ensures**، **EnsuresOnThrow**، **Assert**، **Invariant** و **Assume** یک سربارگذاری‌های اضافی نیز دارند (نشان داده نشده) که یک آرگومان پیام **String** می‌گیرند تا شما بتوانید صریحاً یک پیام را تعیین کنید که وقتی قرارداد نقض می‌شود باید آن پیام ظاهر گردد.

طبق پیش فرض، قراردادها تنها برای مستندات سازی بکار گرفته می‌شوند همانطور که شما هنگام ساخت پروژه‌ی خود نماد **CONTRACTS_FULL** را تعریف نخواهید کرد. به این منظور که ارزش بیشتری برای استفاده از قراردادها بدست آورید، شما باید ابزارهای اضافی و یک صفحه ویژگی برای ویژوال استودیو را از <http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx> دانلود کنید. علت اینکه چرا تمام این ابزارهای قراردادهای کد در ویژوال استودیو قرار داده نشده‌اند این است که این تکنولوژی نسبتاً جدید است و به سرعت در حال بهبود است. وب سایت **DevLabs** از مایکروسافت، می‌تواند نسخه‌های جدید و بهبودها را سریعتر از خود ویژوال استودیو ارائه کند. پس از دانلود و نصب ابزارهای اضافی، شما خواهید دید پروژه-هایتان یک بخش مشخصات جدید برایشان اضافه شده که در شکل ۹-۲۰ نشان داده شده است.



شکل ۹-۲۰ بخش Code Contracts برای یک پروژه ویژوال استودیو

برای آنکه ویژگی‌های قراردادهای کد را فعال کنید، گزینه **Perform Runtime Contract Checking** را انتخاب کرده و **Full** را از جلوی آن انتخاب کنید. این کار یک نماد **CONTRACTS_FULL** وقتی پروژه‌ی خود را بسازید تعریف می‌کند و ابزارهای مناسب (که به مختصر توضیح داده می‌شوند) را پس از ساخت پروژه‌ی شما اجرا می‌کند. اکنون در زمان اجرا، وقتی یک قرارداد نقض می‌شود، رویداد **ContractFailed** از **Contract** فراخوانی می‌شود. معمولاً، برنامه‌نویسان هیچ متدی را با این رویداد ثبت نمی‌کنند اما اگر شما بکنید، آنگاه هر متدی که شما ثبت کنید، یک شی **ContractFailedEventArgs** که شبیه به این است، دریافت می‌کند:

```
public sealed class ContractFailedEventArgs : EventArgs {
    public ContractFailedEventArgs(ContractFailureKind failureKind,
        string message, string condition, Exception originalException);

    public ContractFailureKind    FailureKind        { get; }
    public string                 Message            { get; }
    public string                 Condition          { get; }
    public Exception              OriginalException  { get; }

    public Boolean Handled { get; } // true if any handler called SetHandled
    public void SetHandled();      // Call to ignore the violation; sets Handled to true

    public Boolean Unwind { get; } // true if any handler called SetUnwind or threw
    public void SetUnwind();      // Call to force ContractException; set Unwind to true
}
```

چندین مدیریت‌کننده‌ی رویداد می‌توانند با این رویداد ثبت شوند. هر متد، نقض قرارداد را به نحوی که بخواهد پردازش می‌کند. برای مثال، یک مدیریت‌کننده می‌تواند نقض را گزارش کند، نقض را (با فراخوانی **SetHandled**) نادیده بگیرد، یا پردازش را متوقف و از کار بیاندازد. اگر هر متدی **SetHandled** را فراخوانی کند، آنگاه نقض، مدیریت‌شده محسوب می‌شود و پس از آنکه تمام متدهای مدیریت‌کننده برگشتند، کد برنامه اجازه دارد به اجرا ادامه دهد مگر آنکه مدیریت‌کننده‌ی **SetUnwind** را فراخوانی کند. اگر مدیریت‌کننده‌ی **SetUnwind** را فراخوانی کند، آنگاه پس از آنکه

تمام متدهای مدیریت کننده اجرایشان کامل شد، یک **System.Diagnostics.Contracts.ContractException** تولید می‌شود. توجه کنید که این نوع برای **mscorlib.dll** درونی است و بنابراین شما نمی‌توانید یک بلوک **catch** بنویسید که آن را صریحاً بگیرد. همچنین توجه کنید اگر یک متد مدیریت کننده، یک اکسپشن مدیریت نشده تولید کند، آنگاه متدهای مدیریت کننده‌ی باقی مانده فراخوانی می‌شوند و سپس یک **ContractException** تولید می‌شود.

اگر هیچ مدیریت کننده رویداد موجود نباشد و یا اگر هیچ یک از آن‌ها **SetHandled** یا **SetUnwind** را فراخوانی نکرده یا یک اکسپشن مدیریت نشده تولید نکنند، آنگاه پردازش پیش فرض برای نقض قرارداد در ادامه اتفاق می‌افتد. اگر CLR میزبانی شده باشد، میزبان با خبر می‌شود که یک قرارداد نقض شده است. اگر CLR یک برنامه را روی یک پنجره غیرمحاویره‌ای اجرا می‌کند (که برای یک برنامه سرویس ویندوز اینگونه است) آنگاه **Environment.FailFast** فراخوانی می‌شود تا فوراً پردازش را ببندد. اگر شما در حالیکه گزینه **Assert On Contract Failure** انتخاب شده است کامپایل کنید، آنگاه یک جعبه دیالوگ ادعا (**assert**) ظاهر خواهد شد که به شما اجازه می‌دهد برنامه‌ی خود را به یک دیباگر متصل کنید. اگر این گزینه انتخاب نشده باشد، آنگاه یک **ContractException** تولید می‌شود.

بگذارید به یک کلاس نمونه که از قراردادهای کد استفاده می‌کند نگاه کنیم:

```
public sealed class Item { /* ... */ }

public sealed class ShoppingCart {
    private List<Item> m_cart = new List<Item>();
    private Decimal m_totalCost = 0;

    public ShoppingCart() {
    }

    public void AddItem(Item item) {
        AddItemHelper(m_cart, item, ref m_totalCost);
    }

    private static void AddItemHelper(List<Item> m_cart, Item newItem,
        ref Decimal totalCost) {

        // Preconditions:
        Contract.Requires(newItem != null);
        Contract.Requires(Contract.ForAll(m_cart, s => s != newItem));

        // Postconditions:
        Contract.Ensures(Contract.Exists(m_cart, s => s == newItem));
        Contract.Ensures(totalCost >= Contract.OldValue(totalCost));
        Contract.EnsuresOnThrow<IOException>(totalCost == Contract.OldValue(totalCost));

        // Do some stuff (which could throw an IOException)...
        m_cart.Add(newItem);
        totalCost += 1.00M;
    }

    // Object invariant
    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(m_totalCost >= 0);
    }
}
```

}

متد **AddItemHelper** تعدادی قرارداد کد تعریف می‌کند. پیش شرطها تعیین می‌کنند که **newItem** نباید **null** باشد و آیتمی که به کارت اضافه می‌شود نباید قبلا در کارت باشد. پس شرطها تعیین می‌کنند که آیتم جدید باید در کارت نباشد و کل هزینه باید حداقل به اندازه‌ی باشد که قبل از آنکه آیتم به کارت افزوده شده است، بود. پس شرطها همچنین تعیین می‌کنند که اگر **AddItemHelper** خواست به هر دلیلی یک **IOException** تولید کند، آنگاه **totalCost** نسبت به آنچه قبل از آنکه متد شروع به اجرا کند، تغییر نیافته باشد. متد **ObjectInvariant** تنها یک متد خصوصی است که وقتی فراخوانی می‌شود، اطمینان حاصل می‌کند فیلد **m_totalCost** از شی هرگز حاوی یک مقدار منفی نیست.

مهم تمام اعضای که در یک تست پیش شرط، پس شرط، یا ثابت ارجاع می‌شوند باید خالی از اثرات جانبی باشند. این لازم است چون بررسی شرطها نباید وضعیت خود شی را تغییر دهد. به علاوه، تمام اعضای ارجاعی در یک تست پیش شرط باید حداقل به اندازه‌ی متدی که آن‌ها را تعریف می‌کند دسترس پذیر باشند. این لازم است چون فراخوانی کننده‌های متد باید قادر باشند بازبینی کنند که آن‌ها تمام پیش شرطها را از قبل فراخوانی متد دارا هستند. در سوی دیگر، اعضای ارجاعی در یک تست پس شرط یا ثابت می‌توانند هر دسترس پذیری‌ای داشته باشند تا زمانی که کد بتواند کامپایل شود. علت اینکه که دسترس پذیری در اینجا مهم نیست بدین خاطر است که تست‌های پس شرط و ثابت بر توانایی فراخوانی کننده در فراخوانی صحیح متد اثرگذار نیستند.

مهم با توجه به وراثت، یک نوع مشتق شده نمی‌تواند پیش شرطهای یک عضو مجازی تعریف شده در یک نوع پایه را بازنویسی کرده و تغییر دهد. به طریق مشابه، یک نوع که یک عضو رابط را پیاده سازی می‌کند، نمی‌تواند پیش شرطهای تعریف شده توسط عضو رابط را تغییر دهد. اگر یک عضو، قرارداد صریحی برایش تعریف نشده باشد، آنگاه عضو یک قرارداد ضمنی که منطقی شبیه به این است دارد:

`Contract.Requires(true);`

و چون یک قرارداد نمی‌تواند با نسخه‌های جدید سختگیرانه تر شود (بدون از بین بردن سازگاری با نسخه‌های قبلی) شما باید هنگام معرفی یک عضو مجازی، خلاصه، یا رابط جدید با دقت به پیش شرطها توجه کنید. برای پس شرطها و ثابت‌های شی، قراردادهای می‌توانند افزوده یا حذف شوند وقتی که شرطهای بیان شده در عضو مجازی/خلاصه/رابط و شرطهای بیان شده در عضو بازنویسی شده فقط **AND** منطقی هم شده باشند.

پس اکنون، شما می‌دانید چگونه قراردادهای را تعریف کنید. حال بگذارید در این باره صحبت کنیم که آن‌ها در زمان اجرا چگونه عمل می‌کنند. شما تمام قراردادهای پیش شرط و پس شرط را در بالای متدهایتان جاییکه براحتی قابل یافتن باشند، تعریف می‌کنید. البته، قراردادهای پیش شرط، وقتی متد فراخوانی می‌شود، تست‌هایشان را ارزیابی می‌کنند. هر چند، ما نمی‌خواهیم قراردادهای پس شرط تست‌هایشان را تا زمانی که متد برمی‌گردد ارزیابی کنند. به منظور این که رفتار مورد نظر برسیم، اسمبلی تولیدی توسط کامپایلر سی‌شارپ توسط ابزار **Code Contract Rewriter** باید پردازش شود (**CCRewrite.exe**)، که در مسیر `C:\Program Files (x86)\Microsoft\Contracts\Bin` قرار دارد، که یک نسخه تغییر یافته از اسمبلی تولید می‌کند. پس از آنکه شما گزینه **Perform Runtime Contract Checking** را برای پروژه خود انتخاب کنید، ویژوال استودیو این ابزار را برای شما به صورت خودکار هرگاه شما پروژه را بسازید، فراخوانی (اجرا) می‌کند. این ابزار، **IL** را در تمام متدهای شما آنالیز می‌کند و **IL** را بازنویسی می‌کند تا هر قرارداد پس شرطی در پایان هر متد اجرا شود. اگر متد شما چندین نقطه بازگشت درونش دارد، آنگاه ابزار **CCRewrite.exe** کد **IL** متد را تغییر می‌دهد تا تمام نقاط برگشتی قبل از برگشت متد، کد پس شرط را اجرا کنند.

ابزار **CCRewrite.exe** در یک نوع به دنبال هر متدی که با یک صفت **[ContractInvariantMethod]** علامت زده شده باشد، می‌گردد. متد می‌تواند هر نامی داشته باشد اما طبق قرارداد، مردم معمولاً متد را **ObjectInvariant** نامگذاری کرده و متد را **private** علامت می‌زنند (همانگونه که در بالا انجام دادم). متد باید هیچ آرگومانی نپذیرد و یک نوع برگشتی **void** داشته باشد. وقتی ابزار **CCRewrite.exe** متدی را می‌بیند که با این صفت علامت زده شده است، کد **IL** ای را در انتهای هر متد نمونه **public** درج می‌کند تا متد **ObjectInvariant** را فراخوانی کند. به این روش، وضعیت شی وقتی هر متدی برمی‌گردد بررسی می‌شود تا اطمینان حاصل شود هیچ متدی قرارداد را نقض نکرده باشد. توجه کنید ابزار **CCRewrite.exe** یک متد **Finalize** یا **Dispose** از **IDisposable** را تغییر نمی‌دهد تا متد **ObjectInvariant** را فراخوانی کند چون این برای وضعیت یک شی درست است که هنگام از بین رفتن و نابود شدن، تغییر کند. همچنین توجه کنید یک تک نوع می‌تواند چندین متد با صفت

[ContractInvariantMethod] داشته باشد، این مفید است وقتی می‌خواهید با نوع‌های جزئی کار کنید. ابزار **CCRewrite.exe** کد **IL** را تغییر می‌دهد تا در پایان هر متد عمومی، تمام این متدها را (به ترتیب تعریف نشده) فراخوانی کند.

متدهای **Assert** و **Assume** برخلاف دیگر متدها هستند. اول اینکه شما نباید آن‌ها را بخشی از امضای متد در نظر بگیرید و شما مجبور نیستید آن‌ها را در آغاز یک متد قرار دهید. در زمان اجرا، این دو متد معادل عمل می‌کنند: آن‌ها فقط بررسی می‌کنند شرط ارسالی به آن‌ها درست باشد و اگر نباشد یک اکسپشن تولید می‌کنند. هر چند، ابزار دیگری وجود دارد، **Code Contract Checker (CCCheck.exe)** که کد **IL** تولیدی توسط کامپایلر را بررسی می‌کند تا از لحاظ آماری بررسی کند هر شرط ارسالی به **Assert**، **true** است و ابزار، عبارت را به بدنه‌ی حقایقی که به عنوان صحیح شناخته می‌شوند می‌افزاید. معمولاً، شما از **Assert** استفاده می‌کنید و سپس یک **Assert** را به یک **Assume** تغییر می‌دهید اگر ابزار **CCCheck.exe** نتواند از لحاظ آماری اثبات کند که عبارت درست است. بگذارید یک مثال را بررسی کنیم. فرض کنید من تعریف نوع زیر را دارم:

```
internal sealed class SomeType {
    private static String s_name = "Jeffrey";

    public static void ShowFirstLetter() {
        Console.WriteLine(s_name[0]); // warning: requires unproven: index < this.Length
    }
}
```

وقتی من این کد را با فعال بودن **Perform Static Contract Checking** می‌سازم، ابزار **CCCheck.exe** هشدار می‌دهد که در کامنت فوق آمده است را نشان می‌دهد. این هشدار به من خبر می‌دهد که خواندن اولین حرف از **s_name** ممکن است شکست خورده و یک اکسپشن تولید کند چون اثبات نشده است که **s_name** همیشه به یک رشته حاوی حداقل یک کاراکتر اشاره می‌کند.

بنابراین، آنچه ما دوست داریم انجام دهیم افزودن یک ادعا به متد **ShowFirstLetter** است:

```
public static void ShowFirstLetter() {
    Contract.Assert(s_name.Length >= 1); // warning: assert unproven
    Console.WriteLine(s_name[0]);
}
```

متأسفانه، وقتی **CCCheck.exe** این کد را آنالیز می‌کند، هنوز هم ناتوان است که ارزیابی کند آیا **s_name** همیشه به یک رشته حاوی حداقل یک کاراکتر اشاره دارد، بنابراین ابزار هشدار مشابهی تولید می‌کند. گاهی ابزار ناتوان است که ادعاها را ارزیابی کند به خاطر محدودیت‌های موجود در ابزار؛ نسخه‌های آتی ابزار قادر خواهند بود آنالیز کاملتری انجام دهند.

برای غلبه بر کمبودهای موجود در ابزار یا برای آنکه مدعی شوید چیزی درست است که ابزار هرگز قادر به اثبات آن نیست، ما می‌توانیم **Assert** را به **Assume** تغییر دهیم. اگر به عنوان یک قاعده بدانیم هیچ کد دیگری **s_name** را تغییر نمی‌دهد، آنگاه ما **ShowFirstLetter** را به این تغییر می‌دهیم:

```
public static void ShowFirstLetter() {
    Contract.Assume(s_name.Length >= 1); // No warning at all now!
    Console.WriteLine(s_name[0]);
}
```

با این نسخه از کد، ابزار **CCCheck.exe** سخن ما را گرفته و نتیجه می‌گیرد **s_name** همیشه به یک رشته حاوی حداقل یک حرف اشاره می‌کند. این نسخه از متد **ShowFirstLetter** از بررسی کننده‌ی قرارداد کد بدون هیچ گونه هشدار عبور می‌کند. حال، بگذارید درباره‌ی ابزار **Code Contract Reference Assembly Generator (CCRefGen.exe)** صحبت کنیم. اجرای **CCRefGen.exe** برای فعال سازی بررسی قراردادها به شما کمک می‌کند خطاها را سریعتر بیابید، اما تمام کدی که هنگام بررسی قرارداد تولید می‌شود اسمبلی شما را بزرگتر کرده و بر عملکرد زمان اجرای آن اثرگذار است. برای بهبود این وضعیت شما از ابزار **CCRefGen.exe** برای ساخت یک اسمبلی ارجاعی قرارداد **contract reference assembly** مجزا استفاده می‌کنید. ویزوال استودیو اگر شما **Contract Reference Assembly** را به **Build** تنظیم کنید، این ابزار را به صورت خودکار برای شما اجرا می‌کند. اسمبلی‌های قرارداد معمولاً **AssemName.Contracts.dll** نامیده می‌شوند (برای مثال، **MSCorLib.Contracts.dll**) و این اسمبلی‌ها تنها حاوی متادیتا و **IL** هستند که قراردادها را توصیف می‌کند و دیگر هیچ. شما می‌توانید یک اسمبلی ارجاعی قرارداد را شناسایی کنید چون

شده است. ابزار `CCRewrite.exe` و ابزار `CCCheck.exe` می‌توانند از اسمبلی‌های ارجاعی قرارداد به عنوان ورودی وقتی این ابزارها در حال تنظیم و آنالیز هستند استفاده کنند.

آخرین ابزار (`CCDocGen.exe`) `Code Contract Document Generator` اطلاعات قرارداد را به فایل‌های مستندات `XML` که قبلاً توسط کامپایلر سی‌شارپ وقتی از سویچ کامپایلر `/doc:file` استفاده می‌کنید، تولید شده است، می‌افزاید. این فایل `XML`، که توسط ابزار `CCDocGen.exe` بدان افزوده می‌شود می‌تواند توسط ابزار `Sandcastle` مایکروسافت پردازش شود تا مستنداتی به سبک `MSDN` که اکنون حاوی اطلاعات قرارداد هست را تولید کند.