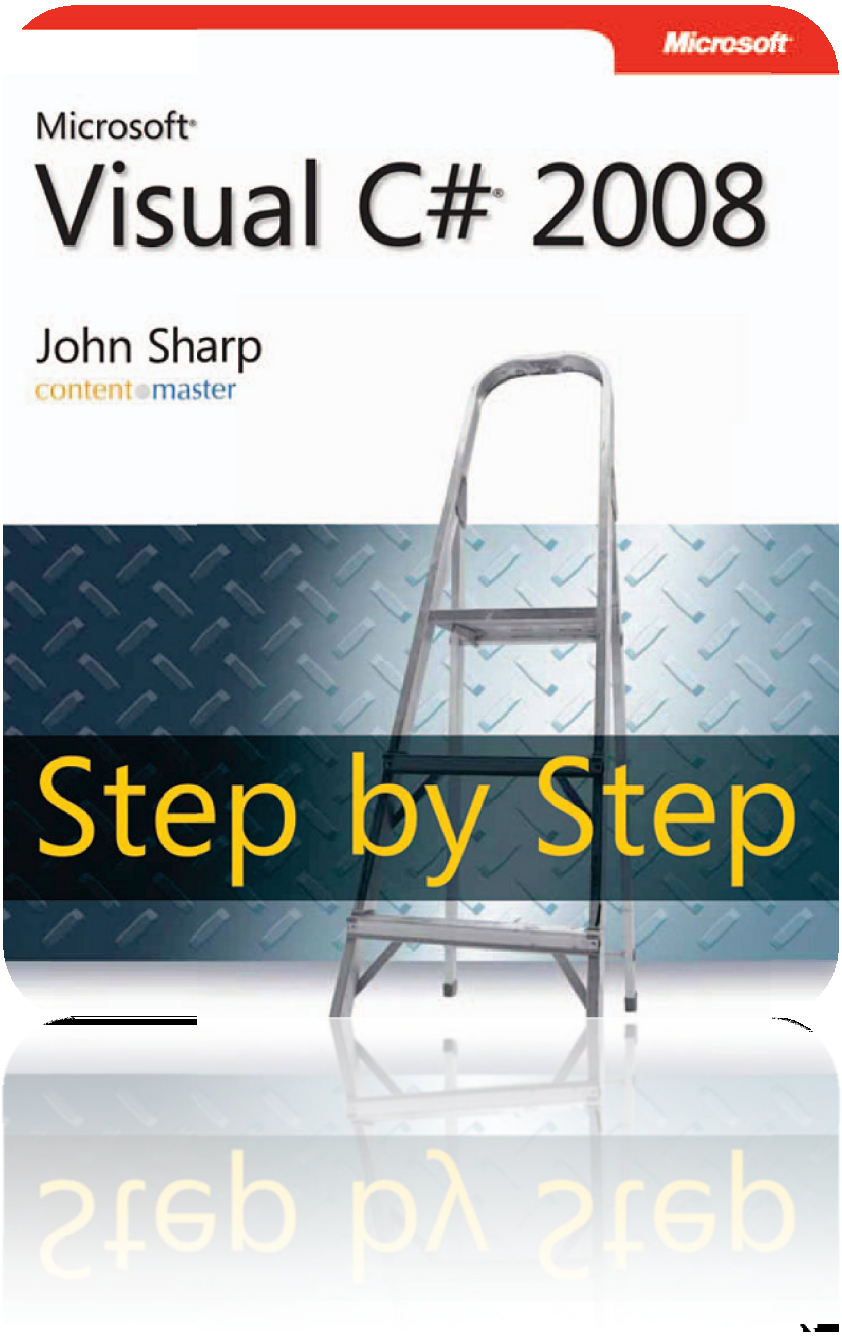
به نام خداوند مهرگستر مهربان



گزیده کتاب ویژوال #C

گزیده کتاب ویژوال C# جان شارپ

ترجمه: مهدی محبیان

تقدیم به عاشقان برنامه نویسی شیء گرا

استفاده از این کتاب با ذکر یک صلوات رایگان است.

فهرست مطالب

فصل ۱

آغاز برنامه نویسی با محیط ویژوال استودیو ۲۰۰۸

بعد از اتمام این فصل، قادر خواهید بود که:

- ☐ از محیط برنامه نویسی مایکروسافت ویژوال استودیو ۲۰۰۸ استفاده کنید.
- ☐ یک برنامه کنسول C# ایجاد کنید.
- ☐ منظور از فضاهای اسمی (namespace) را توضیح دهید.
- ☐ یک برنامه ساده گرافیکی C# ایجاد کنید.

C# زبان جزء گرای قدرتمند مایکروسافت می‌باشد. C# نقش مهمی در معماری چارچوب Microsoft .NET بازی می‌کند و برخی از افراد آن را با نقش C در توسعه یونیکس (UNIX) مقاسه می‌کنند. اگر شما از قبل یک زبان مانند C، C++ یا Java را می‌شناسید، خواهید دید که ترکیب نوشتاری C# به نحو اطمینان بخشی آشناست. اگر شما برای برنامه نویسی در زبان‌های دیگر به کار گرفته شده‌اید، بایست به زودی قادر باشید که ترکیب نوشتاری C# را فرابگیرید و آن را احساس کنید؛ شما تنها نیازمند این هستید تا یاد بگیرید که آکولادها و نقطه ویرگول‌ها را در جای درست بگذارید. امیدوارانه، این کتاب صرفاً برای راهنمایی شماست!

در بخش I، شما پایه‌های C# را فراخواهید گرفت. شما درخواهید یافت که چگونه متغیرها را اعلان کنید و چگونه از عملگرهای محاسباتی مانند علامت جمع (+) و علامت (-) برای دست کاری مقادیر در متغیرها، استفاده کنید. شما خواهید دید که چگونه متدها را بنویسید و آرگومان‌ها را به متدها ارسال کنید. در ضمن شما یاد خواهید گرفت که چگونه از دستورات انتخابی مانند *if* و دستورات ازسرگیری و تکرر مانند *while* استفاده کنید. بالاخره، شما خواهید دانست که C# چگونه از از اخطارها برای اداره خطاها در یک روش مطبوع و با کاربرد آسان، استفاده می‌کند. این عناوین هسته C# را تشکیل داده و این پایه یکپارچه را شکل می‌دهند.

ویژوال استودیو ۲۰۰۸ یک محیط برنامه نویسی ابزار-پیشرفته است که حاوی تمامی عاملیت‌هایی است که شما برای ایجاد پروژه‌های کوچک یا بزرگ C# نیاز دارید. شما حتی می‌توانید پروژه‌هایی را ایجاد کنید که به طور یکپاچه‌ای واحدهای کامپایل شده توسط زبان‌های برنامه نویسی دیگر را ترکیب می‌کنند. برای اولین تمرین، شما محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ را شروع کرده و یاد می‌گیرید که چگونه یک برنامه کنسول ایجاد کنید.

توجه یک برنامه کنسول، برنامه‌ای است که به جای فراهم کردن یک واسط کاربر گرافیکی، در یک پنجره خط فرمان اجرا می‌شود.

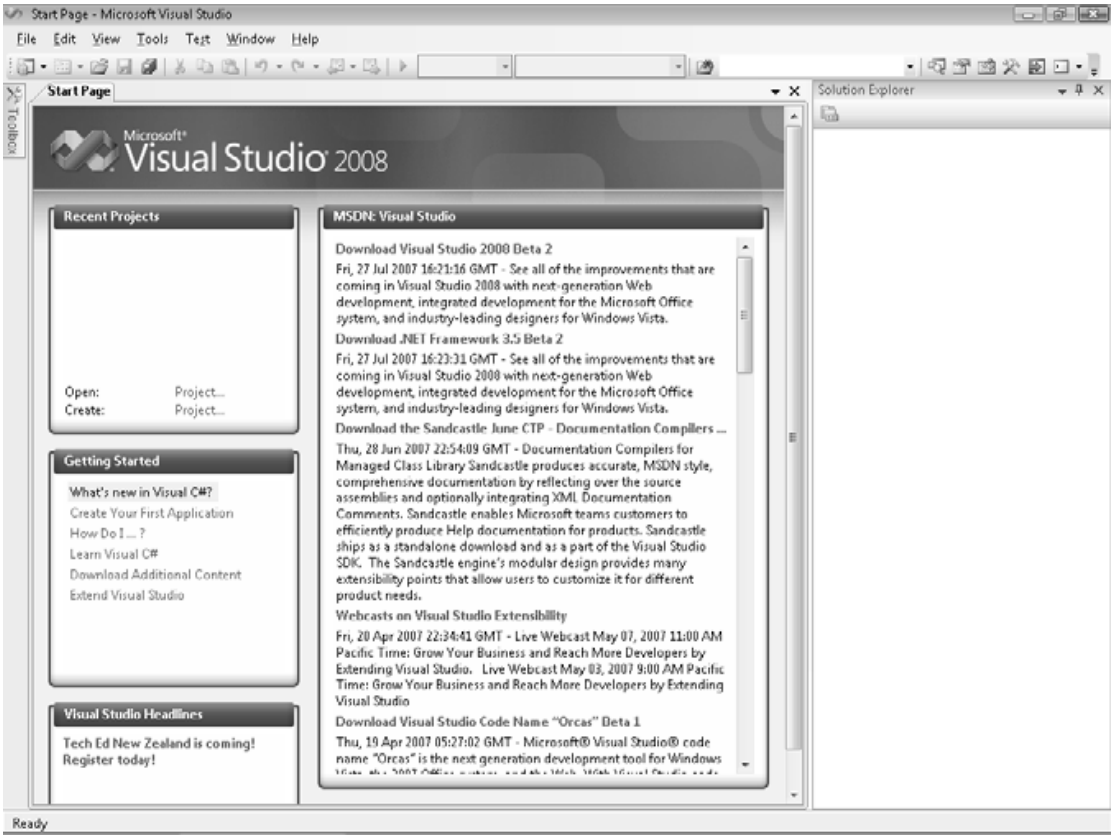
ایجاد یک برنامه کنسول در ویژوال استودیو ۲۰۰۸

- اگر در حال استفاده از ویرایش استاندارد ویژوال استودیو ۲۰۰۸ یا ویرایش حرفه‌ای ویژوال استودیو ۲۰۰۸ هستید، اعمال زیر را برای شروع ویژوال استودیو ۲۰۰۸ انجام دهید:

۱ روی میله وظایف ویندوز، روی دکمه *Start* کلیک کنید، به *All Programs* و سپس گروه برنامه *Microsoft Visual Studio 2008* بروید.

۲ در گروه برنامه Microsoft Visual Studio 2008، *Microsoft Visual Studio 2008* را کلیک کنید.

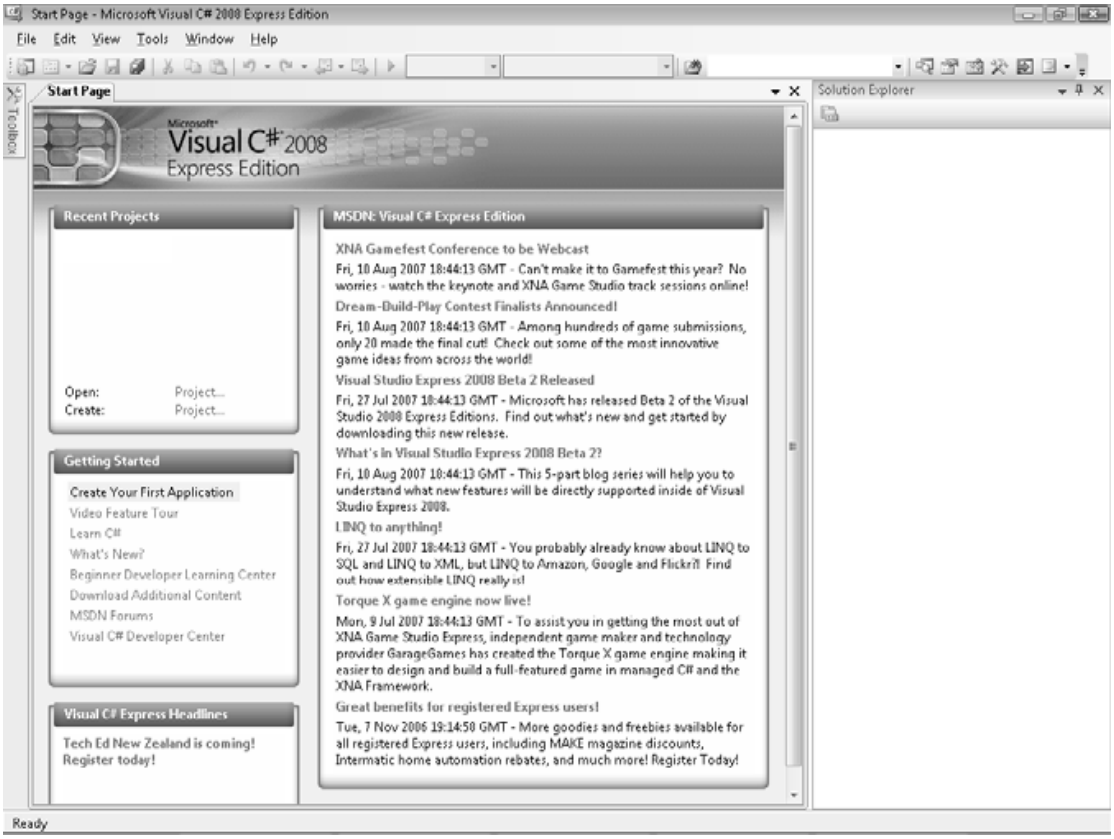
ویژوال استودیو ۲۰۰۸ شروع می‌شود، مانند این:



توجه اگر اولین بار است که از ویژوال استودیو ۲۰۰۸ استفاده می‌کنید، شما ممکن است یک جعبه گفتگو ببینید که سعی می‌کند که شما تنظیمات محیط توسعه پیش فرض خود را انتخاب کنید. ویژوال استودیو ۲۰۰۸ می‌تواند خودش را مطابق زبان طراحی ترجیحی شما سامان دهد. جعبه‌های گفتگوی متعدد و ابزارها در محیط توسعه جامع (IDE) انتخاب‌های پیش فرض خودشان را برای زبان انتخابی شما تنظیم خواهند کرد. *Visual C# Development Settings* را از فهرست انتخاب کنید و سپس دکمه *Start Visual Studio* را کلیک کنید. بعد از یک تأ خیر کوتاه، IDE ویژوال استودیو ۲۰۰۸ ظاهر می‌شود.

- اگر شما در حال استفاده از Visual C# 2008 Express Edition هستید، در میله وظایف ویندوز دکمه *Start* را کلیک کنید، *All Programs* و سپس *Microsoft Visual C# 2008 Express Edition* را انتخاب کنید.

Visual C# 2008 Express Edition شروع می‌شود، مانند این:



توجه برای پرهیز از تکرار، در سرتاسر کتاب، هنگامی که شما نیازمند باز کردن ویرایش استاندارد ویژوال استودیو ۲۰۰۸، ویرایش حرفه‌ای ویژوال استودیو ۲۰۰۸ یا ویرایش اکسپرس ویژوال استودیو ۲۰۰۸ هستید، از ساده سازی " شروع ویژوال استودیو ۲۰۰۸ " استفاده شده است. اضافه بر این، مگر اینکه به طور صریح بیان شود، همه ارجاع‌ها به ویژوال استودیو ۲۰۰۸ به ویرایش استاندارد ویژوال استودیو ۲۰۰۸، ویرایش حرفه‌ای ویژوال استودیو ۲۰۰۸ یا ویرایش اکسپرس ویژوال C# ۲۰۰۸ اعمال می‌شوند.

☐ اگر در حال استفاده از ویرایش استاندارد ویژوال استودیو ۲۰۰۸ یا ویرایش حرفه‌ای ویژوال استودیو ۲۰۰۸ هستید، اعمال زیر برای ایجاد یک برنامه کنسول جدید انجام دهید.

۱ روی منوی *File*، *New* را انتخاب و سپس *Project* را کلیک کنید. جعبه گفتگوی *New Project* باز می‌شود . این جعبه گفتگو الگوهایی را که شما می‌توانید به عنوان یک نقطه شروع برای ایجاد یک برنامه به کار ببرید، لیست می‌کند. جعبه گفتگو، الگوها را مطابق زبان برنامه نویسی که شما در حال استفاده از آن هستید و نوع برنامه، دسته بندی می‌کند.

۲ در قاب *Project types*، روی *Visual C#* کلیک کنید. در قاب *Templates*، آیکون *Console Application* را کلیک کنید.

۳ در فیلد *Location*، اگر در حال استفاده از سیستم عامل ویندوز ویستا هستید، **C:\Users*YourName*\Documents\Microsoft Press\Visual CSharp Step By Step\Chapter 1** را تایپ کنید. اگر در حال استفاده از ویندوز XP یا ویندوز سرور ۲۰۰۳

هستید، **C:\Documents and Settings*YourName*\My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1** را تایپ کنید.

توجه برای صرفه جویی در جا در سرتاسر این کتاب، من به سادگی به مسیر "C:\Users*YourName*\Documents" یا "C:\Documents and Settings*YourName*\My Documents" به عنوان پوشه مستندات شما مراجعه خواهم کرد.

توضیح اگر پوشه‌ای که تعیین می‌کنید وجود نداشته باشد، ویژوال استودیو ۲۰۰۸ آن را برای شما ایجاد می‌کند.

۴ در فیلد *Name*، نام **TextHello** را تایپ کنید.

۵ اطمینان حاصل کنید که جعبه چک *Create directory for solution* انتخاب شده باشد و پس از آن *OK* را کلیک کنید.

☐ اگر شما در حال استفاده از Visual C# 2008 Express Edition هستید، جعبه گفتگوی *New Project* به شما اجازه نخواهد داد تا مکان فایل‌های پروژه را تعیین کنید؛ این مکان به طور پیش فرض پوشه C:\Users*YourName*\ AppData\Local\Temporary Projects است. آن را با رویه زیر تغییر دهید:

۱ در منوی *Tools*، گزینه *Options* را کلیک کنید.

۲ در جعبه گفتگوی *Options*، جعبه چک *Show All Settings* را چک دار کنید و سپس *Projects and Solutions* را در نمای درختی واقع در قاب چپ، کلیک کنید.

۳ در قاب راست، در جعبه متن *Visual Studio projects location*، پوشه *Microsoft Press|Visual CSharp Step By Step|Chapter 1* را تحت پوشه Documents خود تعیین کنید.

☐ اگر در حال استفاده از Visual C# 2008 Express Edition هستید، اعمال زیر را برای ایجاد یک برنامه کنسول جدید انجام دهید.

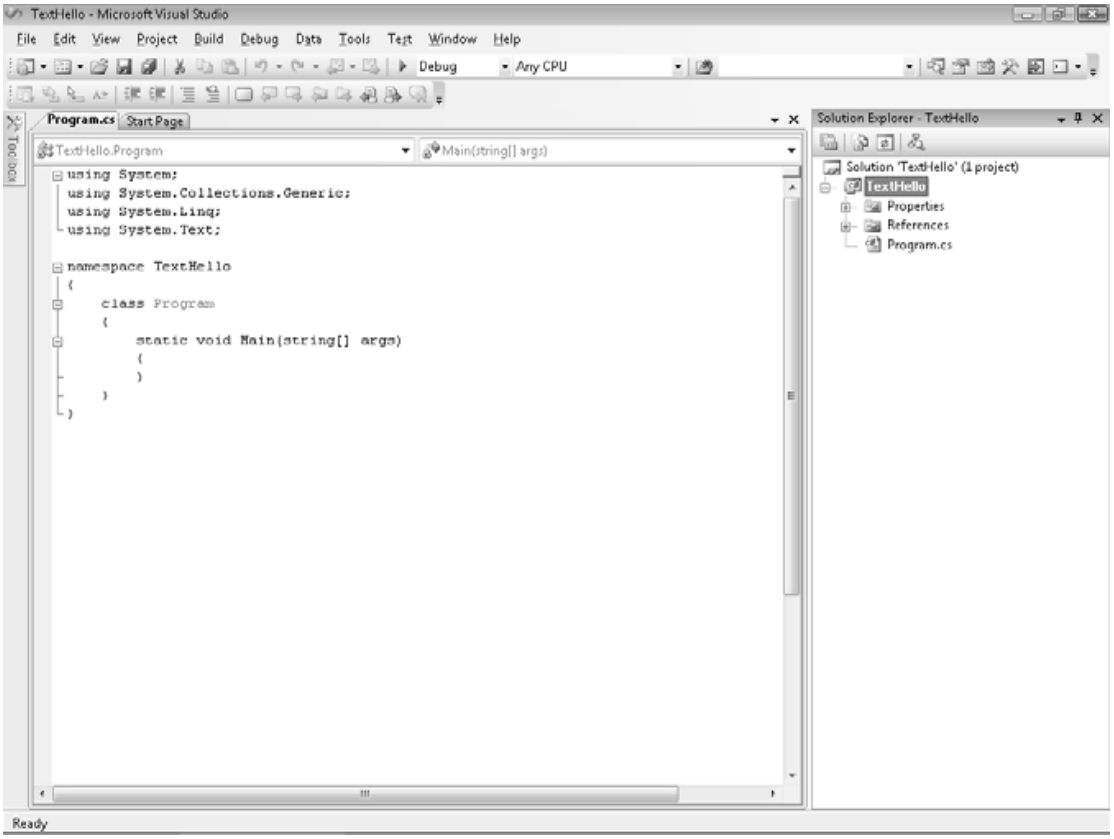
۱ روی منوی *File*، گزینه *New Project* را کلیک کنید.

۲ در جعبه گفتگوی *New Project*، آیکون *Console Application* را کلیک کنید.

۳ در فیلد *Name*، اسم **TextHello** را تایپ کنید.

۴ *OK* را کلیک کنید.

ویژوال استودیو ۲۰۰۸ پروژه را با استفاده از الگوی برنامه کنسول ایجاد می‌کند و کد شروع کننده را برای پروژه نمایش می‌دهد، مانند این:



میله منو در بالای صفحه دسترسی به ویژگی‌هایی که شما در محیط برنامه نویسی به کار خواهید برد را در اختیار قرار می‌دهد. شما می‌توانید از ماوس یا صفحه کلید برای دسترسی به منوها و فرمانها استفاده کنید، دقیقاً همان طور که در برنامه‌های برپایه پنجره‌ها می‌توانید. *میله ابزار* در زیر میله منو مستقر شده است و دکمه‌های میان بری را برای اجرای فرمان‌هایی که بارها به کاربرده می‌شوند، در دسترس قرار می‌دهد.

پنجره ویرایشگر کد و متن که بخش اصلی IDE را اشغال کرده است، محتویات فایل‌های منبع را نمایش می‌دهد. در یک پروژه چند فایله، هرگاه بیشتر از یک فایل را ویرایش کنید، هر فایل منبع صفحه مخصوص خود را دارد که با اسم فایل منبع برچسب دار شده است. شما می‌توانید صفحه را کلیک کنید تا فایل منبع برده را در پنجره ویرایشگر کد ومتن به پیش زمینه بیاورد. *Solution Explorer* اسامی فایل‌های مرتبط با پروژه را، همراه با آیتم‌های دیگر نمایش می‌دهد. در ضمن شما می‌توانید یک اسم فایل را در *Solution Explorer* دابل کلیک کنید تا آن فایل منبع را در پنجره ویرایشگر کد و متن به پیش زمینه برسانید.

قبل از نوشتن کد، فایل‌های فهرست شده در *Solution Explorer* را که ویژوال استودیو ۲۰۰۸ به عنوان بخشی از پروژه شما ایجاد کرده است، بازدید کنید:

- **Solution 'TextHello'** این فایل، فایل solution تراز بالا است، که از آن به ازای هر برنامه یکی موجود است. اگر از مرورگر ویندوز برای نگاه کردن به پوشه Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello استفاده می‌کنید، شما خواهید دید

که نام واقعی این فایل TextHello.sln است. هر فایل solution حاوی ارجاع‌هایی به یک یا چند پروژه است.

- **TextHello** این فایل پروژه C# است. هر فایل پروژه به یک یا چند فایل حاوی کد منبع و آیتم‌های دیگر برای پروژه اشاره می‌کند. تمامی کد منبع واقع در یک پروژه واحد باید در زبان برنامه نویسی یکسان نوشته شود. در مرورگر ویندوز، این فایل در واقع TextHello.csproj نامیده شده است و در پوشه \My Documents\Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello\TextHello شما ذخیره شده است.

- **Properties** این یک پوشه در پروژه TextHello است. اگر آن را بسط دهید، شما خواهید دید که‌این پوشه حاوی یک فایل با نام AssemblyInfo.cs است. AssemblyInfo.cs یک فایل ویژه است که شما می‌توانید برای اضافه کردن مشخصه‌ها به یک برنامه استفاده کنید، مانند اسم یک مؤلف، تاریخی که برنامه نوشته شده است و ... . شما می‌توانید مشخصه‌های اضافی را برای اصلاح روشی که برنامه به آن روش اجرا می‌شود، تعیین کنید. فراگیری چگونگی استفاده از این مشخصه‌ها خارج از سطح این کتاب است.

- **References** این یک پوشه است که حاوی ارجاع‌هایی به کد کامپایل شده است که برنامه شما استفاده می‌کند. هرگاه کد کامپایل می‌شود، به یک اسمبلی تبدیل شده و یک نام منحصر به فرد مفروض می‌گیرد. طراحان از اسمبلی‌ها برای بسته بندی بیت‌های سودمند کد که آنها نوشته‌اند استفاده می‌کنند بنابراین طراحان می‌توانند آن را به طراحان دیگر، کسانی که ممکن است بخواهند از کد در برنامه‌هایشان استفاده کنند، توزیع کنند. خیلی از ویژگی‌هایی که هنگام نوشتن برنامه‌ها با استفاده از این کتاب استفاده خواهید کرد، استفاده از اسمبلی‌های تدارک دیده شده توسط مایکروسافت با ویژوال استودیو ۲۰۰۸ را امکان پذیر می‌کند.

- **Program.cs** این یک فایل منبع C# است و همانی است که هنگامی که پروژه اولین بار ایجاد شد، به طور جاری در پنجره ویرایشگر کد و متن نمایش داده می‌شود. شما کد خود را برای برنامه کنسول در این فایل خواهید نوشت. در ضمن این فایل حاوی برخی کدهایی که ویژوال استودیو ۲۰۰۸ به طور خودکار عرضه می‌کند، است که شما می‌توانید به زودی آنها را آزمایش و بررسی کنید.

نوشتن اولین برنامه شما

فایل Program.cs یک کلاس با نام *Program* تعریف می‌کند که حاوی یک متد با نام *Main* است. همه متدها باید درون یک کلاس تعریف شوند. شما در فصل ۷، در بخش " ایجاد و مدیریت کلاس‌ها و اشیاء " درباره کلاس‌ها بیشتر یاد خواهید گرفت. متد *Main* استثنایی است--- این متد نقطه ورودی برنامه را برمی‌گزیند. *Main* باید یک متد استاتیک باشد. (شما به صورت جزئیاتی در فصل ۳، بخش " نوشتن متدها و اعمال سطح دسترسی " متدها را خواهید دید و درباره متدهای استاتیک در فصل ۷ بحث خواهد شد.

مهم C# یک زبان حساس به حالت حروف است. شما باید *Main* را با یک حرف درشت *M* بنویسید.

در فعالیت‌های زیر، شما کدی را برای نمایش پیغام Hello World در کنسول خواهید نوشت؛ برنامه کنسول Hello World را بنا و اجرا خواهید کرد؛ و یاد خواهید گرفت که چگونه فضاهای اسمی برای تفکیک اجزای کد به کار برده می‌شوند.

کد را با استفاده از IntelliSense بنویسید

- در پنجره ویرایشگر کد و متن در حال نمایش فایل Program.cs، مکان نما را در متد *Main* بلافاصله بعد از آکولاد باز---{--- قرار دهید و سپس Enter را برای ایجاد یک سطر جدید فشار دهید. روی سطر جدید، واژه **Console**، که اسم یک کلاس توکار است را تایپ کنید. به محض اینکه شما حرف C را در آغاز کلمه **Console** تایپ کردید، یک لیست IntelliSense ظاهر می‌شود. این لیست حاوی همه کلید واژه‌های C# و انواع داده که در این متن معتبر هستند، می‌باشد. شما یا می‌توانید تایپ را ادامه دهید یا در میان لیست مرور کرده و گزینه Console را با ماوس دابل کلیک کنید. متناوباً، بعد از اینکه شما *Con* تایپ کرده‌اید، لیست IntelliSense به طور خودکار در روی گزینه Console خواهد نشست و شما می‌توانید با فشار دادن کلید Tab یا Enter آن را انتخاب کنید.

Main بایست مانند این به نظر برسد:

```
static void Main(string[ ]  args)

{

    Console

}
```

توجه Console یک کلاس توکار است که حاوی متدهایی برای نمایش دادن پیغامها روی صفحه و گرفتن ورودی از صفحه کلید است.

- یک نقطه بلافاصله بعد از *Console* تایپ کنید. لیست دیگر IntelliSense ظاهر می‌شود، که متدها، خاصیت‌ها و فیلدهای کلاس *Console* را نمایش می‌دهد.

- به پایین لیست رفته، *WriteLine* را انتخاب کنید و سپس Enter را فشار دهید. به طور متناوب، می‌توانید به تایپ کاراکترهای *W*, *r*, *i*, *t*, *e*, *L* ادامه دهید تا اینکه *WriteLine* انتخاب شود و پس از ان Enter را فشار دهید. لیست IntelliSense بسته می‌شود و کلمه *WriteLine* به فایل منبع اضافه می‌شود. اکنون *Main* بایست مانند این به نظر رسد:

```
static void Main(string[] args)

{

        Console.WriteLine

}
```

- یک پارانتز باز ---)--- تایپ کنید. توضیح IntelliSense دیگری ظاهر می‌شود. این توضیح پارامترهایی را که متد *WriteLine* می‌تواند بگیرد نمایش می‌دهد. در واقع، *WriteLine* یک متد *overloaded* است، به‌این معنی که کلاس *Console* حاوی بیشتر از یک متد با نام *WriteLine* است--- در واقع این کلاس ۱۹ نسخه مختلف از این متد را عرضه می‌کند. هر نسخه از متد *WriteLine* می‌تواند برای بیرون دادن انواع متفاوتی از داده به کار برده شود. (فصل ۳ متدهای *overloaded* را با جزئیات بیشتری توضیح می‌دهد.) اکنون *Main* بایست مانند این به نظر رسد:

```
static void Main(string[] args)

{

        Console.WriteLine(

}
```

توضیح شما می‌توانید کلیدهای جهت بالا و پایین را کلیک کنید تا در توضیح IntelliSense در میان *overloaded*های متفاوت *WriteLine* مرور انجام دهید.

- یک پارانتز بستن---(--- را تایپ کنید که با یک نقطه-ویرگول (;) پی گرفته می‌شود.

Main بایست مانند این به نظر رسد:

```
static void Main(string[] args)

{

Console.WriteLine();

}
```

- مکان نما را حرکت داده و رشته **"Hello World"** را به انضمام علائم نقل قول، در میان پارانتزهای چپ و راست بعد از متد *WriteLine* تایپ کنید. *Main* بایست مانند این به نظر می‌رسد:

```
static void Main(string[] args)
```

```
{

Console.WriteLine(“Hello World”);

}
```

توضیح عادت کنید که جفت کاراکترهای همتا، مانند (and) و {and} را قبل از پر کردن آنها با محتویاتشان تایپ کنید. اگر شما منتظر باشید تا اینکه بعد از اینکه محتویات را وارد کردید، کاراکتر بستن را وارد کنید، خیلی راحت می‌توانید کاراکتر بستن را فراموش کنید.

آیکون‌های IntelliSense

هرگاه شما یک نقطه را بعد از اسم یک کلاس تایپ می‌کنید، IntelliSense اسم هر عضو آن کلاس را نمایش می‌دهد. در سمت چپ هر اسم عضو یک آیکون است که نوع آن عضو را شرح می‌دهد. آیکون‌های عمومی‌و انواع آنها در زیر ضمیمه شده‌اند:

آیکون	مفهوم
	متد
	خاصیت
	کلاس
	ساختار
	شمارشی
	واسط
	نماینده
	متد الحاقی

در ضمن شما آیکون‌های IntelliSense دیگر را ببینید که به محض اینکه شما کد را در زمینه‌های متفاوت تایپ می‌کنید، ظاهر می‌شوند.

توجه خیلی از اوقات شما سطرهایی از کد را حاوی دو فوروارد اسلش (//) که با متن معمولی پی گرفته می‌شود، خواهید دید. اینها مستندات (کامنت‌ها) هستند. آنها توسط کامپایلر نادیده گرفته می‌شوند اما برای طراحان خیلی سودمند هستند زیرا آنها به سند کمک می کنند که یک برنامه در واقع مشغول انجام چه کاری است.برای مثال:

```
Console.ReadLine();     // Wait for the user to press the Enter key
```

کامپایلر تمام متن را از دو اسلش تا انتهای سطر دور خواهد انداخت. در ضمن شما می‌توانید کامنت‌های چند سطری را با یک فوروارد اسلش که با یک ستاره پی گرفته می‌شود (*/) اضافه کنید. کامپایلر تا زمانی که یک ستاره با اسلش فوروارد (/*) پیدا کند از متن صرف نظر می‌کند که‌این متن می‌تواند چندین سطر پایین رفته باشد. شما تشویق می‌شوید که با جدیت کد خود را با بسیاری کامنت‌های معنادار هر قدر هم که لازم شد، مستند کنید.

بنا کردن و اجرای برنامه کنسول

در منوی *Build*، آیتم *Build Solution* را کلیک کنید.

این عملیات کد C# را کامپایل می‌کند، با نتیجه دادن در یک برنامه که شما می‌توانید آن را اجرا کنید. پنجره خروجی در پایین پنجره ویرایشگرکد و متن ظاهر می‌شود.

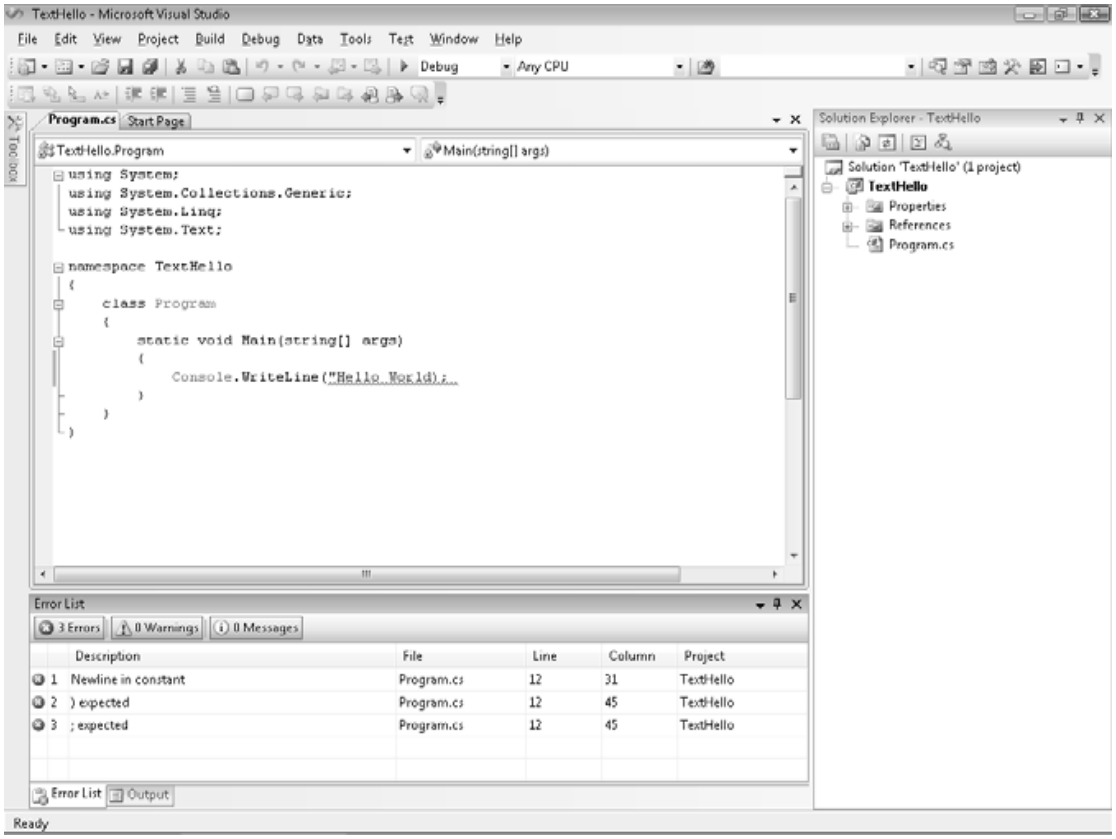
توضیح اگر پنجره خروجی ظاهر نشود، در منوی *View*، گزینه *Output* را برای نمایش پنجره خروجی کلیک کنید.

در پنجره خروجی (*Output*)، شما باید بایست پیغامهایی شبیه زیر ببینید که نشان می‌دهند که چگونه برنامه کامپایل شده است.

```
------ Build started: Project: TextHello, Configuration: Debug Any CPU ----

C:\Windows\Microsoft.NET\Framework\v3.5\Csc.exe /config /nowarn:1701;1702 …

Compile complete -- 0 errors, 0 warnings

TextHello -> C:\Documents and Settings\John\My Documents\Microsoft Press\…
```

```
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ========
```

اگر شما برخی اشتباهات را داشته باشید، آنها در پنجره فهرست خطا (*Error List*) ظاهر خواهند شد. تصویر زیر نشان می‌دهد که اگر شما علامت نقل قول بستن را بعد از متن Hello World در دستور *WriteLine* فراموش کنید چه اتفاقی خواهد افتاد. توجه کنید که یک اشتباه واحد، برخی اوقات سبب بروز چندین خطای کامپایل می‌شود.



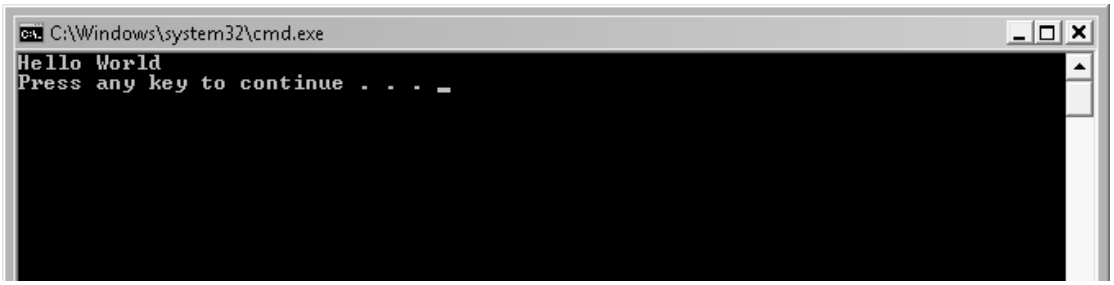
توضیح شما می‌توانید یک آیتم را در پنجره *Error List* دابل کلیک کنید و مکان نما در روی سطری جای خواهد گرفت که باعث بروز خطا شده است. در ضمن شما بایست توجه کنید که ویژوال استودیو ۲۰۰۸ در زیر سطرهایی از کد که هرگاه شما آنها را وارد کنید، کامپایل نخواهند شد، یک خط قرمز مواج نمایش می‌دهد.

اگر شما دستورالعمل‌های قبلی را با دقت پی گرفته‌اید، نبایست هیچ خطا یا هشداری وجود داشته باشد و برنامه بایست با موفقیت بنا شود.

توضیح در اینجا قبل از بنا کردن پروژه هیچ نیازی به ذخیره فایل به طور صریح نیست زیرا دستور *Build Solution* به طور خودکار فایل را ذخیره می‌کند. اگر شما در حال استفاده از Visual Studio 2008 Standard Edition یا Visual Studio 2008 Professional Edition هستید، پروژه در موقعیتی ذخیره می‌شود که شما هنگام ایجاد کردن آن تعیین کرده‌اید. اگر شما در حال استفاده از Visual C# 2008 Express Edition هستید، پروژه در موقعیت موقت ذخیره می‌شود و تنها زمانی که به طور صریح پروژه را با استفاده از فرمان *Save All* در منوی *File* ذخیره کنید یا هنگام بستن Visual C# 2008 Express Edition به پوشه‌ای که شما در جعبه گفتگوی *Options* تعیین کرده‌اید، کپی می‌شود.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

یک پنجره فرمان باز می‌شود. پیغام Hello World ظاهر می‌شود و سپس برنامه منتظر می‌ماند تا شما یک کلید را فشار دهید، همان طور که در شکل زیر نشان داده شده است:



توجه عکس العمل "Press any key to continue . . ." توسط ویژوال استودیو ۲۰۰۸ تولید می‌شود؛ شما هیچ کدی برای انجام این کار نمی‌نویسید. اگر شما برنامه را توسط فرمان *Start Debugging* در منوی *Debug* اجرا کنید، برنامه اجرا می‌شود، اما پنجره فرمان بدون این که منتظر بماند تا شما کلیدی را فشار دهید، بلافاصله بسته می‌شود.

مطمئن شوید که پنجره فرمان در حال نمایش دادن خروجی برنامه، فوکوس داشته باشد و Enter را فشار دهید.

پنجره فرمان بسته می‌شود و شما به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ برمی‌گردید.

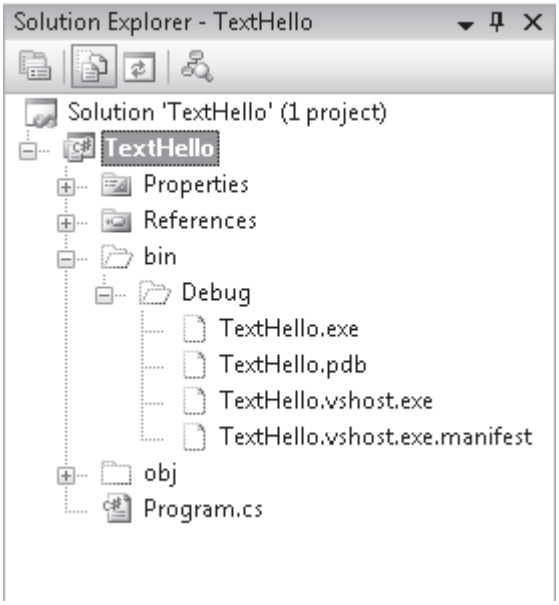
در *Solution Explorer*، پروژه TextHello (نه solution) را کلیک کنید و سپس دکمه میله ابزار *Show All Files* را روی میله ابزار *Solution Explorer* کلیک کنید--- این دکمه، دومین دکمه از سمت چپ روی میله ابزار در پنجره Solution Explorer است.

مدخل‌هایی با نام‌های *bin* و *obj* در بالای فایل Program.cs باز می‌شوند. این مدخل‌ها به طور مستقیم با پوشه‌هایی به نام *bin* و *obj* در پوشه پروژه (Microsoft Press\Visual CSharp Step by Step\Chapter 1\TextHello\TextHello) مربوط می‌شوند. ویژوال استودیو ۲۰۰۸ هنگامی که شما برنامه را بنا کنید، ایجاد می‌کند و آنها حاوی نسخه قابل اجرای برنامه به ضمیمه برخی فایل‌های به کار رفته برای ایجاد و دیباگ برنامه هستند.

در *Solution Explorer*، علامت به اضافه (+) را در سمت چپ مدخل *bin* کلیک کنید. پوشه دیگری با نام *Debug* ظاهر می‌شود.

تـوجه در ضمن شما ممکن است یک پوشه با نام *Release* را هم ببینید.

در *Solution Explorer*، علامت به اضافه (+) را در سمت چپ پوشه *Debug* کلیک کنید. چهار آیتم دیگر با نام‌هـای TextHello.exe، TextHello.pdb، TextHello.vshost.exe و TextHello.vshost.exe.manifest ظـاهر می‌شوند، مانند این:



تـوجه اگر شما در حال استفاده از Visual C# 2008 Express Edition هستید، ممکن است همه‌این فایل‌ها را نبینید.

فایل TextHello.exe برنامه کامپایل شده است و همان فایلی است که هنگامی که شما *Start Without Debugging* را در منوی *Debug* کلیک می‌کنید، اجرا می‌شود. فایل‌هـای دیگر حـاوی اطلاعـاتی هسـتند که اگر شـما برنامـه را بـه شـیوه *Debug* اجـرا کنیـد (هرگـاه *Start Debugging* را در منـوی *Debug* کلیـک کنید) توسط ویژوال استودیو ۲۰۰۸ به کار برده می‌شوند.

استفاده از فضاهای نام (Namespaceها)

مثالی که شما تاکنون دیده‌اید یک برنامه بسیار کوچکی است. گرچـه، برنامـه‌های کوچـک می‌تواننـد بـه طرف زودی به برنامـه‌های بزرگتـر رشـد کننـد. زمـانی کـه یـک برنامـه بـزرگ می‌شـود، دو مسـئله پـیش می‌آید. اول، درک کردن و نگه داری برنامه‌های بزرگ مشکل تر از فهم و نگـه داری از برنامـه‌های کوچـک تر است. دوم، کد بیشتر به معنای اسامی بیشتر، متدهای بیشتر و کلاس‌های بیشتر است. همین کـه

تعداد اسامی افزایش می‌یابد، احتمال اینکه بنا کردن پروژه عقیم بماند بیشتر می‌شود زیرا دو یا چند اسم با هم برخورد می‌کنند (به خصوص زمانی که برنامه هم از کتابخانه‌های شخص ثالث نوشته شده توسط طراحانی که آنها نیز از اسامی متنوعی استفاده کرده‌اند، استفاده می‌کند).
در گذشته، برنامه نویسان سعی می‌کردند مشکل برخورد اسم را توسط پیشوند دار کردن اسامی با برخی از انواع توصیف کننده‌ها (قیدها) حل کنند. این راه حل خوب نیست زیرا قابل مقیاس گذاری نیست؛ اسامی بلند تر می‌شوند و شما زمان کمتری برای نوشتن نرم افزار صرف می‌کنید و زمان بیشتری برای تایپ کردن (در اینجا یک تفاوتی وجود دارد) و خواندن و بازخوانی، به طور نامفهوم اسامی بلند صرف کرد.

فضاهای اسمی (Namespaceها) برای حل این مشکل توسط ایجاد کردن یک کانتینر مشخص برای شناسه‌های دیگر، مانند کلاس‌ها ، کمک می‌کند. اگر دو کلاس با نام یکسان در فضاهای اسمی متفاوتی زندگی کنند، با یکدیگر قاطی نخواهند شد. شما می‌توانید یک کلاس با نام *Greeting* را درون فضای اسمی با نام *TextHello* ایجاد کنید، مانند این:

```
namespace TextHello

{

    class Greeting

    {

        ...

    }

}
```

در این صورت شما می‌توانید در برنامه خود به کلاس *Greeting* به صورت *TextHello.Greeting* مراجعه کنید. اگر طراح دیگری نیز یک کلاس *Greeting* در یک فضای اسمی متفاوت، مانند *NewNamespace*، ایجاد کند و آن را روی کامپیوتر شما نصب کند، برنامه شما هنوز هم همان طور که انتظار می‌رفت کار می‌کند زیرا آنها در حال استفاده از کلاس *TextHello.Greeting* هستند. اگر شما می‌خواهید که به کلاس *Greeting* طراح دیگر ارجاع کنید، باید آن را به صورت *NewNamespace.Greeting* تعیین کنید.

این تمرین خوبی است که کلاس‌های خود را در فضاهای اسمی تعریف کنید و محیط ویژوال استودیو ۲۰۰۸ از این توصیه به واسطه استفاده از اسم پروژه شما به عنوان فضای اسمی تراز بالا، پیروی می‌کند. جعبه توسعه نرم افزاری (SDK) چارچوب .NET. نیز به‌این توصیه وفادار است؛ هر کلاس در چارچوب .NET. درون یک فضای اسمی زندگی می‌کند. برای مثال، کلاس *Console* درون فضای اسمی *System* زندگی می‌کند. این حرف به معنای این است که اسم کامل آن باید در واقع *System.Console* باشد.

البته ؛ در صورتی که شما بایستی نام کامل یک کلاس را هر زمان که از آن استفاده می‌کنید، بنویسید، وضعیت بهتر از پیشوندار کردن توصیف کننده‌ها یا حتی نامیدن کلاس با تعدادی اسم منحصر به فرد به صورت سرتاسری مانند *SystemConsole* و به دردسر نیفتادن با یک فضای اسمی ، نخواهد بود. خوشبختانه، شما می‌توانید این مشکل را با یک دستور *using* در برنامه تان حل کنید. اگر به برنامه

TextHello در ویژوال استودیو ۲۰۰۸ برگردید و به فایل Program.cs در پنجره *ویرایشگر کد ومتن* نگاه کنید، شما دستورات زیر را در بالای فایل ملاحظه خواهید کرد:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;
```

یک دستور *using* یک فضای اسمی را به سطح دسترسی (میدان دید) می‌رساند. در کد متعاقب در همان فایل، شما دیگر نبایست به طور ضمنی اشیاء را با فضای اسمی که آنها به آن تعلق دارند، قیددار کنید. چهار فضای اسمی نشان داده شده دربردارنده کلاس‌هایی هستند که خیلی اوقات به کار برده می‌شوند که ویژوال استودیو ۲۰۰۸ هر زمان که شما یک پروژه جدید ایجاد می‌کنید به طور خودکار این دستورات *using* را اضافه می‌کند. شما می‌توانید دستورهای *using* بیشتری را به بالای یک فایل منبع اضافه کنید.

فعالیت زیر مفهوم فضاهای اسمی را با عمق بیشتری شرح می‌دهد.

تلاش برای تمام نویسی اسامی

در پنجره *ویرایشگر کد ومتن* که در حال نمایش فایل Program.cs است، اولین دستور *using* در بالای فایل را توضیح دار کنید، مانند این:

```
//using System;
```

در منوی *Build*، گزینه *Build Solution* را کلیک کنید.

بنا کردن عقیم می‌ماند و پنجره فهرست خطا (*Error List*) پیغام خطای زیر را نمایش می‌دهد:

```
The name 'Console' does not exist in the current context.
```

در پنجره فهرست خطا (*Error List*)، پیغام خطا را دابل کلیک کنید.

شناسه‌ای که باعث خطا شده در فایل منبع Program.cs انتخاب می‌شود.

در پنجره *ویرایشگر کد ومتن*، متد *Main* را برای استفاده از اسم کاملاً قیددار شده *System.Console* ویرایش کنید. *Main* بایست مانند این به نظر برسد:

```
static void Main(string[] args)

{

System.Console.WriteLine("Hello World");
```

```
}
```

توجه هرگاه شما *System.* را تایپ می‌کنید، اسامی تمامی آیتم‌ها در فضای اسمی *System* توسط IntelliSense نمایش داده می‌شود.

در منوی *Build*، گزینه *Build Solution* را کلیک کنید.

بناکردن در این نوبت بایست موفقیت آمیز باشد. اگر این گونه نبود، مطمئن شوید که *Main* دقیقاً آن طوری است که در کد قبلی ظاهر شد و سپس سعی کنید که مجدداً آن را بنا کنید.

برنامه را اجرا کنید تا مطمئن شوید که برنامه هنوز هم توسط کلیک کردن *Start Without Debugging* در منوی *Debug* کار می‌کند.

فضاهای اسمی و اسمبلی‌ها

یک دستور *using* رساندن آیتم‌ها در یک فضای اسمی را به درون سطح دسترسی (میدان دید) ساده سازی می‌کند و شما را از مجبور شدن برای قیددار کردن اسامی کلاس‌ها در کدتان به طور کامل، خلاص می‌کند. کلاس‌ها به درون اسمبلی‌ها (*assemblies*) کامپایل می‌شوند. یک اسمبلی فایلی است که معمولاً پسوند اسم فایل dll. دارد، گرچه اگر بخواهیم دقیق تر بگوییم، برنامه‌های قابل اجرا با پسوند نام فایل exe. نیز جزو اسمبلی‌ها هستند.

یک اسمبلی می‌تواند حاوی کلاس‌های بسیاری باشد. کلاس‌هایی که کتابخانه کلاس چارچوب .NET دربردارد، مانند *System.Console*، در اسمبلی‌هایی که به ضمیمه ویژوال استودیو ۲۰۰۸ در روی کامپیوتر شما نصب می‌شوند، تدارک دیده شده‌اند. شما درخواهید یافت که کتابخانه کلاس چارچوب .NET دربردارنده چندین هزار کلاس است. اگر آنها در اسمبلی یکسانی نگه داشته شده بودند، اسمبلی بزرگ جثه و برای نگه داری و پشتیبانی مشکل می‌شد. (اگر مایکروسافت یک متد یگانه را در یک کلاس یگانه به روزرسانی می‌کرد، بایست سراسر کتابخانه کلاس را به تمامی طراحان توزیع می‌کرد!)

به همین دلیل، کتابخانه کلاس چارچوب .NET. به میان تعدادی اسمبلی منشعب می‌شود، جزءبندی شده توسط حوزه کارکردی به آنهایی که کلاس‌هایی که آنها در بردارند، با هم رابطه دارند. برای مثال، یک اسمبلی هسته وجود دارد که دربردارنده همه کلاس‌های عمومی‌است، مانند *System.Console* و اسمبلی‌های بیشتری وجود دارد که حاوی کلاس‌هایی برای دست کاری پایگاه‌های داده، دسترسی به سرویس‌های وب، ساخت واسطهای کاربر گرافیکی و ... هستند . اگر می‌خواهید از یک کلاس در یک اسمبلی استفاده کنید، شما باید یک ارجاع به آن اسمبلی به پروژه خودتان اضافه کنید. در این صورت شما می‌توانید دستورات *using* را به کد خود اضافه کنید که‌ایتم‌ها در فضای اسمی در آن اسمبلی را به سطح دسترسی (میدان دید) می‌رسانند.

بایست توجه کنید که لزوماً یک هم ارزی یک به یک (1:1) بین یک اسمبلی و یک فضای اسمی وجود ندارد؛ یک اسمبلی تک می‌تواند حاوی کلاس‌هایی برای چندین فضای اسمی باشد و یک فضای

اسمی تک می‌تواند چندین اسمبلی را پدید آورد. اینها همگی در ابتدا گیج کننده به نظر می‌رسند، اما شما به زودی آنها را به کار خواهید بست.

هر گاه شما از ویژوال استودیو ۲۰۰۸ برای ایجاد یک برنامه استفاده می‌کنید، الگویی که انتخاب می‌کنید به طور خودکار دربردارنده ارجاعاتی به اسمبلی‌های مقتضی است. برای مثال، در *Solution Explorer* برای پروژه TextHello، علامت به اضافه (+) را در سمت چپ پوشه *References* کلیک کنید. خواهید دید که یک برنامه کنسول به طور خودکار دربردارنده ارجاعاتی به اسمبلی‌هایی با نام *System*، *System.Core*، *System.Data* و *System.Xml* است. شما می‌توانید به یک پروژه ارجاعاتی برای اسمبلی‌های اضافی توسط راست کلیک کردن پوشه *References* و کلیک بر روی *Add Reference*، اضافه کنید--- شما انجام این مأموریت را در فعالیت‌های بعدی، تمرین خواهید کرد.

ایجاد یک برنامه گرافیکی

تاکنون، شما از ویژوال استودیو ۲۰۰۸ برای ایجاد و اجرای یک برنامه کنسول پاه استفاده کرده‌اید. محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ ضمناً هر چیزی را که شما برای ایجاد برنامه‌های گرافیکی برمبنای پنجره‌ها نیاز دارید را دربردارد. شما می‌توانید واسط کاربر برپایه فرم یک برنامه برمبنای پنجره‌ها را به طور تعاملی طراحی کنید. ویژوال استودیو ۲۰۰۸ سپس دستورات برنامه را برای پیاده سازی واسط کاربر که شما طراحی کرده‌اید، تولید می‌کند.

ویژوال استودیو ۲۰۰۸ شما را با دو نمای یک برنامه گرافیکی مجهز می‌کند: نمای طراحی و نمای کد. شما از پنجره ویرایشگر متن و کد برای اصلاح و نگه داری کد و منطق برای یک برنامه گرافیکی استفاده می‌کنید و از پنجره نمای طراحی (*Design View*) برای چیدن واسط کاربر خود استفاده می‌کنید. شما می‌توانید هرگاه که می‌خواهید بین این دو نما سوییچ کنید.

در مجموعه فعالیت‌های زیر، شما چگونگی ایجاد یک برنامه گرافیکی با استفاده از ویژوال استودیو ۲۰۰۸ را یاد خواهید گرفت. این برنامه یک فرم ساده حاوی این موارد است: یک جعبه متن جایی که شما می‌توانید نام خود را وارد کنید و یک دکمه که هرگاه شما آن را کلیک کنید یک خوشامدگویی شخصی سازی شده را در یک جعبه پیغام نمایش می‌دهد.

توجه ویژوال استودیو ۲۰۰۸ دو الگو برای ساخت برنامه‌های گرافیکی عرضه می‌کند--- الگوی Windows Forms Application و الگوی WPF Application . Windows Forms یک تکنولوژی است که در ابتدا با .NET Framework version 1.0 ظاهر شد. WPF یا Windows Presentation Foundation ، یک تکنولوژی تسهیل شده است که اولین بار با .NET Framework version 3.0 ظاهر شد. این تکنولوژی ویژگی‌ها و قابلیت‌های اضافی بسیاری علاوه بر Windows Forms عرضه می‌کند و شما بایست برای همه طراحی‌ها و پیشرفت‌های جدید، استفاده از آن را به Windows Forms در اولویت قرار دهید.

ایجاد یک برنامه گرافیکی در ویژوال استودیو ۲۰۰۸

□ اگر شما در حال استفاده از Visual Studio 2008 Standard Edition یا Visual Studio 2008 Professional Edition هستید، اعمال زیر را برای ایجاد یک برنامه گرافیکی جدید انجام دهید:

در منوی *File*، *New* و سپس *Project* را انتخاب کنید.

جعبه گفتگوی *New Project* باز می‌شود.

در قاب *Project Types*، روی *Visual C#* کلیک کنید.

در قاب *Templates*، روی آیکون *WPF Application* کلیک کنید.

مطمئن شوید که فیلد *Location*، به پوشه *Documents|Microsoft Press|Visual CSharp Step by Step|Chapter 1* شما اشاره می‌کند.

در فیلد *Name*، نام **WPFHello** را تایپ کنید.

در فیلد *Solution*، مطمئن شوید که *Create new solution* انتخاب شده است.

این عمل یک محلول جدید برای نگه داشتن پروژه‌ایجاد می‌کند. متقابلاً، *Add to Solution*، پروژه را به محلول TextHello اضافه می‌کند.

OK را کلیک کنید.

اگر شما در حال استفاده از Visual C# 2008 Express Edition هستید، کارهای زیر را برای ایجاد یک برنامه گرافیکی جدید انجام دهید.

در منوی *File*، گزینه *New Project* را کلیک کنید.

اگر پنجره پیغام *New Project* ظاهر شد، *Save* را برای ذخیره تغییرات خودتان به پروژه TextHello کلیک کنید. در جعبه گفتگوی *Save Project*، بررسی کنید که فیلد *Location* به *Microsoft Press|Visual CSharp Step By Step|Chapter 1* تحت پوشه Documents شما تنظیم شده باشد و پس از آن *Save* را کلیک کنید.

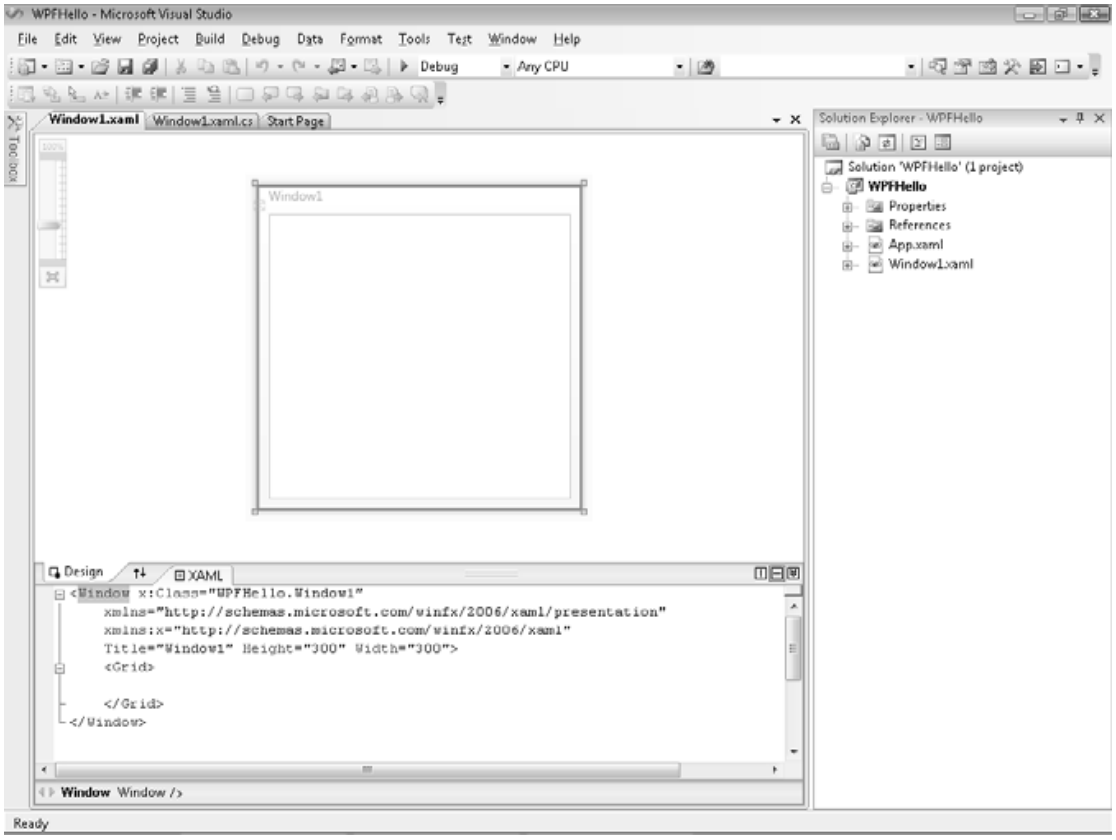
در جعبه گفتگوی *New Project*، آیکون *WPF Application* را کلیک کنید.

در فیلد *Name*، نام **WPFHello** را تایپ کنید.

OK را کلیک کنید.

ویژوال استودیو ۲۰۰۸ برنامه جاری شما را می‌بندد و برنامه WPF جدیدی ایجاد می‌کند. ویژوال استودیو ۲۰۰۸ یک فرم WPF خالی در پنجره نمای طراحی (*Design View*)، به انضمام پنجره دیگری حاوی یک توضیح XAML از فرم نمایش می‌دهد، همان طور که در تصویر زیر نشان داده شده است:

توضیح پنجره‌های خروجی (*Output*) و فهرست خطا (*Error List*) را ببندید تا فضای بیشتری برای نمایش پنجره نمای طراحی (*Design View*) فراهم شود.



XAML مخفف Extensible Application Markup Language است و یک زبان شبیه XML است که توسط برنامه‌های WPF برای تعریف آرایش یک فرم و محتویاتش به کار برده می‌شود. اگر شما شناختی از XML دارید، XAML بایست آشنا به نظر برسد. شما در واقع اگر دوست ندارید که از پنجره نمای طراحی ویژوال استودیو استفاده کنید یا اگر شما دسترسی به ویژوال استودیو ندارید، می‌توانید یک فرم WPF را به طور کامل توسط نوشتن یک تعریف XAML تعریف کنید؛ مایکروسافت یک ویرایشگر XAML با نام XMLPad تدارک دیده است که شما می‌توانید به صورت رایگان از وب سایت MSDN دانلود کنید.

در فعالیت زیر، شما از پنجره نمای طراحی برای اضافه کردن سه کنترل به فرم ویندوز استفاده خواهید کرد و اندکی کد C# را که به طور خودکار توسط ویژوال استودیو ۲۰۰۸ تولید شده است برای پیاده-سازی این کنترل‌ها ، امتحان خواهید کرد.

ایجاد یک واسط کاربر

برگه *Toolbox* را که در سمت چپ فرم در پنجره نمای طراحی ظاهر می‌شود، کلیک کنید.

جعبه ابزار ظاهر می‌شود، در حای که‌اندکی از فرم را پوشانده است و اجزا و کنترل‌های متعددی را نمایش می‌دهد که شما می‌توانید روی یک فرم ویندوز جای دهید. بخش عمومی‌لیستی از کنترل‌هایی که توسط

بیشتر برنامه‌های WPF به کار برده می‌شوند، نمایش می‌دهد. بخش کنترل‌ها یک لیست بسیار گسترده از کنترل‌ها را نمایش می‌دهد.

در بخش عمومی، برچسب را کلیک کنید و پس از آن بخش قابل مشاهده فرم را کلیک کنید.

یک کنترل برچسب به فرم اضافه می‌شود (شما تا یک لحظه دیگر آن را به موقعیت صحیحش جا به جا خواهید کرد)، و جعبه ابزار (*Toolbox*) از نظر ناپدید می‌شود.

توضیح اگر می‌خواهید که جعبه ابزار قابل مشاهده باقی بماند اما هیچ بخشی از فرم را پنهان نکند، دکمه *Auto Hide* را در سمت راست میله عنوان *Toolbox* (که مانند یک گیره به نظر می‌رسد) کلیک کنید. *Toolbox* به طور ثابت در سمت چپ پنجره ویژوال استودیو ۲۰۰۸ ظاهر می‌شود و پنجره نمای طراحی (*Design View*) برای تطبیق نمودن با آن منقبض می‌شود. (اگر شما یک رزولوشن صفحه پایین داشته باشید، ممکن است فضای زیادی را از دست بدهید.) کلیک کردن *Auto Hide* یک بار دیگر باعث می‌شود که *Toolbox* دوباره ناپدید شود.

کنترل برچسب روی فرم احتمالاً به طور دقیق در جایی نیست که شما می‌خواهید. شما می‌توانید کنترلهایی را که به یک فرم اضافه کرده‌اید برای تغییر موقعیت آنها، کلیک کرده و بکشید (drag کنید). با استفاده از این تکنیک، کنترل برچسب را حرکت دهید تا اینکه برچسب نسبت به گوشه چپ بالایی فرم مستقر شود. (جایابی دقیق برای این برنامه ضروری نیست.)

توجه تعریف XAML از یک فرم در قاب پایین اکنون شامل کنترل برچسب، به انضمام خاصیت‌هایی مانند موقعیتش در روی فرم، کنترل شده توسط خاصیت *Margin*، است. خاصیت *Margin* از چهار عدد تشکیل شده است که فاصله هر لبه برچسب از لبه‌های فرم را نشان می‌دهند. اگر کنترل را در هر جای فرم جابه جا کنید، خاصیت *Margin* تغییر می‌کند. اگر فرم تغییر اندازه داده شود، کنترل‌های مهار شده به لبه‌های فرمی‌که حرکت می‌کند، تغییر اندازه داده می‌شوند تا مقادیر حاشیه آنها حفظ شود. شما می‌توانید از این امر توسط تنظیم کردن مقادیر *Margin* به صفر، جلوگیری کنید. شما درباره *Margin* و نیز خاصیت‌های *Height* و *Width* کنترل‌های WPF در فصل ۲۲ " آشنایی با Windows Presentation Foundation" بیشتر یاد می‌گیرید.

در منوی *View*، گزینه *Properties Window* را کلیک کنید.

پنجره *Properties* در پایین ترین سمت راست صفحه، زیر *Solution Explorer* نمایش می‌یابد (اگر از قبل نمایش نیافته بود). پنجره *Properties* روش دیگری برای شما فراهم می‌کند تا خاصیت‌ها را برای آیتم‌های روی یک فرم، به اضافه‌ایتم‌های دیگر در یک پروژه اصلاح کنید. *Properties* نسبت به حالت متنی که در آن خاصیت‌ها را برای آیتم انتخاب شده جاری نمایش می‌دهد حساس است. اگر میله عنوان فرم نمایش یافته در پنجره نمای طراحی (*Design View*) را کلیک کنید، شما می‌توانید ببینید که پنجره خاصیت‌ها (*Properties*) خاصیت‌ها را برای فرم خودش نمایش می‌دهد. اگر کنترل برچسب را کلیک کنید، پنجره در عوض خاصیت‌ها را برای برچسب نمایش می‌دهد. اگر شما در هر جایی غیر از فرم کلیک کنید، پنجره

خاصیت‌ها (*Properties*)، خاصیت‌ها را برای یک آیتم مبهم به نام *grid* (شبکه) نمایش می‌دهد. یک شبکه همانند یک کانتینر برای آیتم‌های روی یک فرم WPF عمل می‌کند و شما می‌توانید از شبکه (grid) همراه با چیزهای دیگر، برای نشان دادن اینکه‌ایتم‌ها چگونه روی فرم بایست تراز شوند و با یکدیگر گروه بندی شوند، استفاده کنید.

کنترل برچسب را روی فرم کلیک کنید. در پنجره خاصیت‌ها (*Properties*)، بخش *Text* را پیدا کنید.

با استفاده از خاصیت‌های واقع در این بخش، شما می‌توانید فونت و اندازه فونت را برای برچسب تعیین کنید اما متن واقعی را که برچسب نمایش می‌دهد در این بخش نمی‌توانید تعیین کنید.

خاصیت *FontSize* را به **20** تغییر دهید و سپس میله عنوان فرم را کلیک کنید.

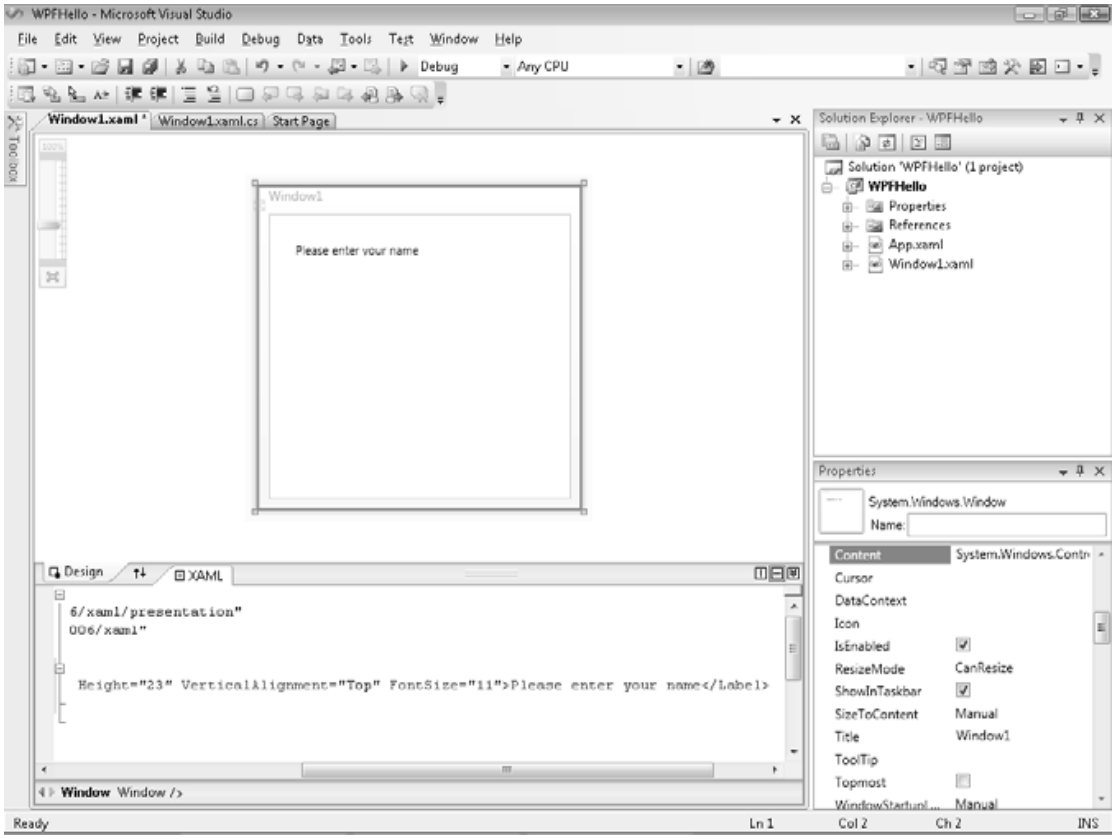
اندازه متن واقع در برچسب تغییر می‌کند، با اینکه متن برای نمایش متن به‌اندازه کافی بزرگ نیست. خاصیت *FontSize* را به **12** تغییر دهید.

توجه متن نمایش یافته در برچسب ممکن است بلافاصله خودش را در پنجره نمای طراحی *Design View* تغییر اندازه ندهد. هنگامی که شما برنامه را ساخت و اجرا کنید، یا اگر فرم را در پنجره *Design View* بسته و باز کنید، برچسب خودش را تصحیح خواهد کرد.

تعریف XAML فرم را در پایین ترین قاب از سمت راست مروز کنید و خاصیت‌های کنترل برچسب را امتحان کنید.

کنترل برچسب از یک تگ <Label> حاوی مقادیر خاصیت، پی گرفته شده با متنی برای خود برچسب ("Label") که پس از آن یک تگ بستن </Label> می‌آید، تشکیل می‌شود.

متن برچسب را (تنها قبل از تگ بستن) به **Please enter your name** تغییر دهید، همان طور که در تصویر زیر نشان داده شده است.



توجه کنید که متن نمایش یافته در برچسب روی فرم تغییر می‌کند، با اینکه برچسب هنوز هم برای نمایش آن به درستی خیلی کوچک است.

فرم را در پنجره *Design View* کلیک کنید و سپس جعبه ابزار (*Toolbox*) را دوباره نمایش دهید.

توجه اگر شما فرم را در پنجره *Design View* کلیک نکنید، جعبه ابزار (*Toolbox*) پیغام "There are no usable controls in this group." را نمایش می‌دهد.

در *Toolbox*، آیتم *TextBox* را کلیک کنید و پس از آن فرم را کلیک کنید. یک کنترل جعبه متن به فرم اضافه می‌شود. کنترل جعبه متن را جابه جا کنید تا اینکه یک راست در زیر کنترل برچسب باشد.

توضیح هرگاه شما یک کنترل را روی یک فرم بکشید (Drag کنید)، هنگامی که کنترل با کنترل‌های دیگر به طور عمودی یا افقی تراز شود، نشانگرهای تراز به طور خودکار ظاهر می‌شوند. این امر برای مطمئن شدن از اینکه کنترل‌ها به طور شست و رفته صف- آرایی شده‌اند، یک سرنخ ویژوال سریع به شما می‌دهد.

در حالی که کنترل جعبه متن انتخاب شده است، در پنجره خاصیت‌ها (*Properties*)، مقدار خاصیت *Name* نمایش یافته در بالای پنجره را به **userName** تغییر دهید.

توجه درباره قراردادهای نام گذاری برای کنترل‌ها و متغیرها در فصل ۲، بخش " کار با متغیرها، عملگرها و عبارت‌ها " بیشتر یاد خواهید گرفت.

جعبه ابزار (*Toolbox*) را دوباره نمایش دهید، *Button* را کلیک کنید، و سپس فرم را کلیک کنید. کنترل دکمه را به سمت راست کنترل جعبه متن روی فرم بکشید (Drag کنید) تا اینکه کف دکمه به طور افقی با کف جعبه متن تراز شود.

با استفاده از پنجره خاصیت‌ها ، خاصیت *Name* کنترل دکمه را به **ok** تغییر دهید.

در تعریف XAML فرم، متن را به سمت راست برای نمایش عنوان نمایش یافته توسط دکمه مرور کنید و آن را از Button به **OK** تغییر دهید. بررسی کنید که عنوان کنترل دکمه در روی فرم تغییر می‌کند.

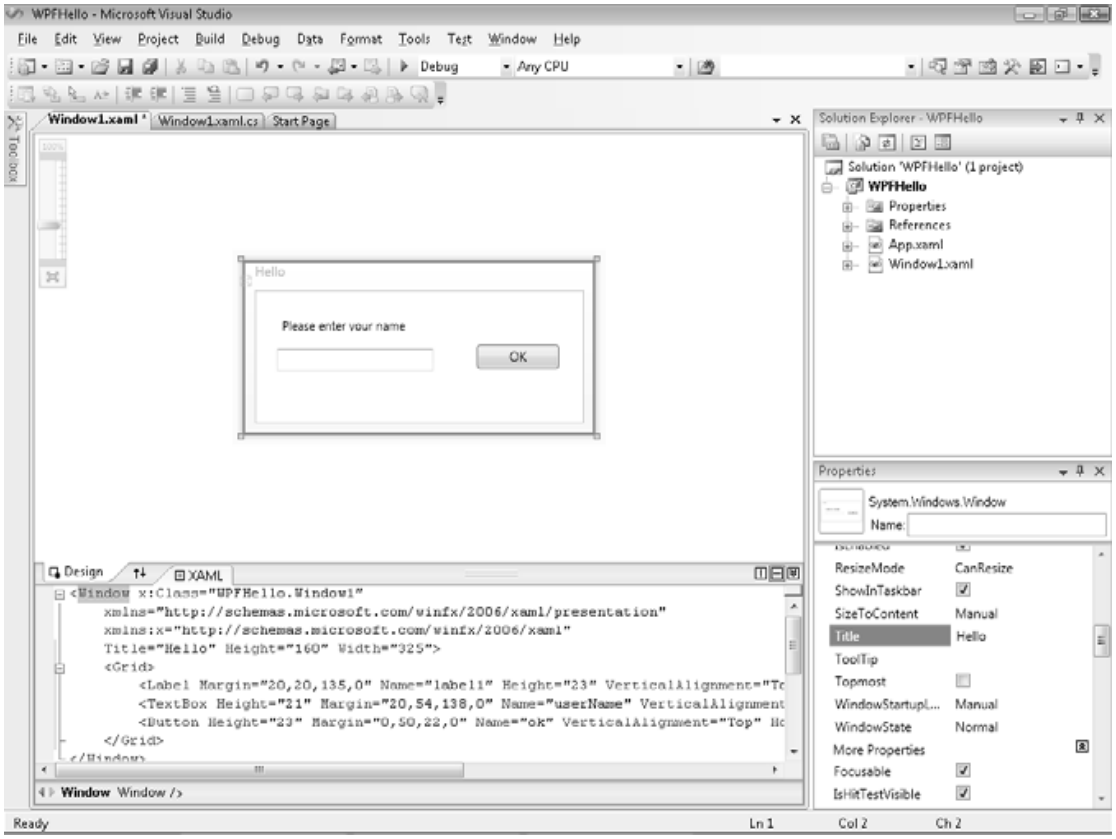
میله عنوان فرم Window1.xaml را در پنجره *Design View* کلیک کنید. در پنجره خاصیت‌ها، خاصیت *Title* را به **Hello** تغییر دهید.

در پنجره *Design View*، توجه کنید هنگامی که فرم انتخاب می‌شود، یک دستگیره تغییر اندازه (یک چهارگوش کوچک) روی پایین ترین گوشه دست راست فرم ظاهر می‌شود. اشاره گر ماوس را روی دستگیره تغییراندازه حرکت دهید. هرگاه اشاره گر به یک پیکان اریب دوسرنوک دار تبدیل شد، اشاره گر را کلیک کرده و بکشید تا فرم تغییر اندازه پیدا کند. هرگاه فاصله پیرامون کنترل‌ها تقریباً مساوی شد، کشیدن را متوقف کرده و دکمه ماوس را رها کنید.

مهم میله عنوان فرم را و نه آرایش شبکه درون فرم را قبل از تغییر اندازه آن، کلیک کنید. اگر شبکه (grid) را انتخاب کنید، شما آرایش کنترل‌ها را روی فرم تغییر می‌دهید نه‌اندازه خود فرم را.

توجه اگر فرم را باریک تر کنید، دکمه OK با یک فاصله ثابت، مشخص شده با خاصیت *Margin* دکمه، از لبه دست راستی فرم باقی می‌ماند. اگر شما فرم را خیلی باریک تر کنید، دکمه *OK* روی کنترل جعبه متن نوشته خواهد شد. حاشیه دست راستی برچسب نیز ثابت است و هنگامی که برچسب منقبض می‌شود به محض اینکه فرم باریک تر می‌شود، متن برای برچسب شروع به ناپدید شدن می‌کند.

اکنون فرم بایست شبیه‌این به نظر برسد:



در منوی *Build*، گزینه *Build Solution* را کلیک کنید و بررسی کنید که پروژه با موفقیت بنا شده باشد.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

برنامه بایست اجرا شود و فرم شما را نمایش دهد. شما می‌توانید اسم خود را در جعبه متن تایپ کنید و سپس *OK* را کلیک کنید، اما هنوز چیزی اتفاق نمی‌افتد. شما نیاز دارید که‌اندکی کد را برای پردازش رویداد *Click* برای دکمه *OK* اضافه کنید، که آن چیزی است که شما بعداً می‌خواهید انجام دهید.

دکمه *Close* (*X*) واقع در گوشه بالایی سمت راست فرم) را کلیک کنید تا فرم را ببندید و به ویژوال استودیو برگردید.

شما بدون نوشتن یک سطر واحد از کد C# مدیریت شده‌اید تا یک برنامه گرافیکی ایجاد کنید. با این حال این کار هنوز کار زیادی انجتم نداده است (شما بایست برخی کدها را به زودی بنویسید)، اما ویژوال استودیو در واقع کد بسیاری برای شما تولید می‌کند که وظایف روتینی را اداره می‌کنند که تمامی برنامه‌های گرافیکی باید انجام دهند، مانند راه‌اندازی و نمایش یک فرم. قبل از اضافه کردن کد شخصی خود به برنامه، کمک می‌کند که یک آشنایی از آن چه ویژوال استودیو برای شما تولید کرده است، داشته باشید.

در *Solution Explorer*، علامت به اضافه (+) را پهلوی فایل Window1.xaml کلیک کنید. فایل Window1.xaml.cs ظاهر می‌شود. فایل Window1.xaml.cs را دابل کلیک کنید. کد برای فرم در ویرایشگر کد و متن ظاهر می‌شود. این کد مانند زیر به نظر می‌رسد:

```
using System;
```

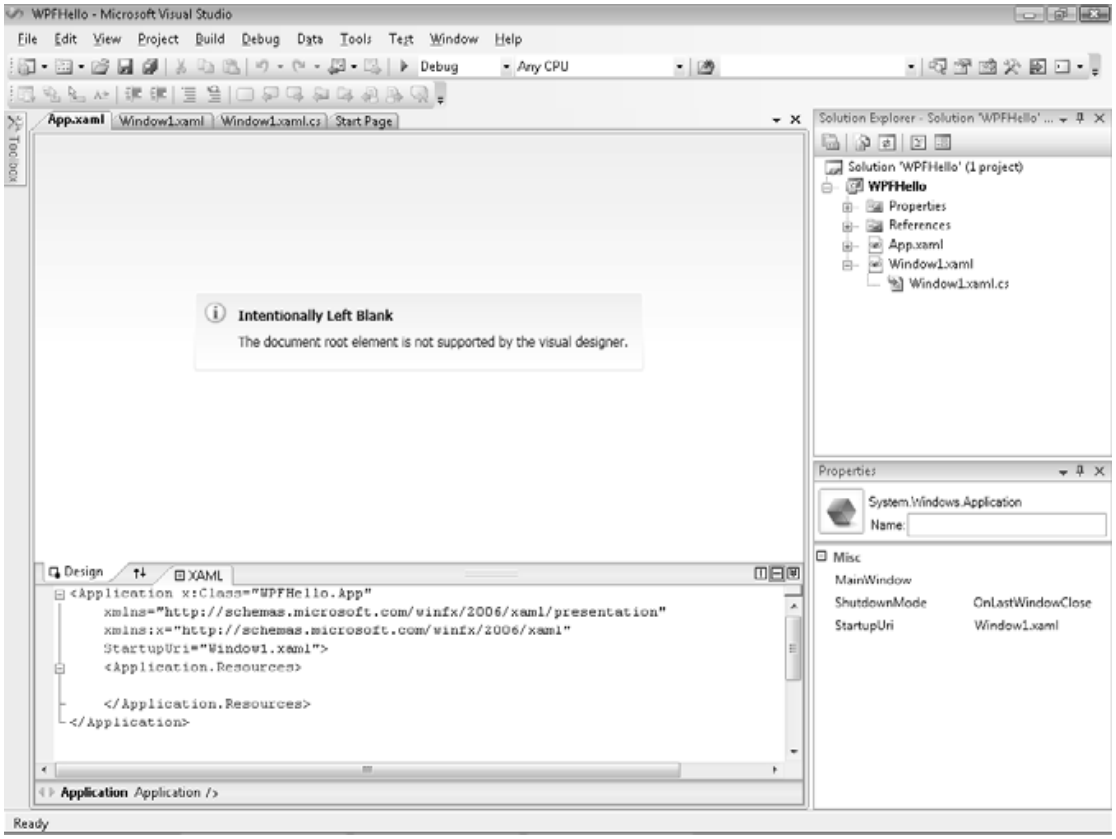
```
using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Navigation;

using System.Windows.Shapes;

namespace WPFHello

{

/// <summary>

/// Interaction logic for Window1.xaml

/// </summary>

public partial class Window1: Window

{

public Window1()

{

InitializeComponent();

}

}

}
```

صرف نظر از تعدادخوبی از دستورات *using* برای رسیدن به میان سطح دسترسی (میدان دید) فضای‌های اسمی که اغلب برنامه WPF استفاده می‌کنند، فایل دربردارنده تعریف یک کلاس با نام *Window1* است اما نه خیلی دیگر. در این اینجا یک خرده کوچکی ار کد لازم برای کلاس *Window1* است که به عنوان یک سازنده شناخته می‌شود که یک متد با نام *InitializeComponent* را فراخوانی می‌کند، ولیکن آن همه چیز است. (یک سازنده *constructor*/ یک متد مخصوصی است با همان نامی کلاس دارد. سازنده هنگامی که یک وهله از کلاس ایجاد می‌شود اجرا می‌شود و می‌تواند دربردارنده کدی برای مقداردهی اولیه وهله باشد. درباره سازنده‌ها در فصل ۷ بیشتر یاد خواهید گرفت.) در واقع، برنامه حاوی کد بیشتری است، اما این کدها به طور خودکار برمبنای تعریف XAML فرم، تولید می‌شود و از چشم شما مخفی است. این کد مخفی عملیات‌هایی نظیر ایجاد و تخریب فرم، و ایجاد و مستقر کردن کنترل‌های جورواجور روی فرم را انجام می‌دهد.

منظور کدی که شما می‌توانید در این کلاس ببینید این است که شما بتوانید متدهای مال خود را برای اداره کردن منطق لازم برای برنامه اضافه کنید، مانند آنچه که هنگام کلیک دکمه *OK* توسط کاربر، اتفاق می‌افتد.

توضیح شما می‌توانید توسط راست کلیک کردن در هر جایی در پنجره نمای طراحی (*Design View*) و سپس کلیک کردن *View Code* فایل کد C# را هم برای یک فرم WPF نمایش دهید.

بسیار خوب در این نقطه شما ممکن است نگران شده باشید که متد *Main* کجاست و هنگامی که برنامه اجرا می‌شود فرم چگونه نمایش داده خواهد شود؛ به یاد داشته باشید که متد *Main* نقطه را تعریف می‌کند که در برنامه شروع می‌شود. در *Solution Explorer*، شما بایست به فایل منبع دیگری با نام App.xaml توجه کنید. اگر این فایل را دابل کلیک کنید، پنجره نمای طراحی / *View Design* پیغام "Intentionally Left Blank," را نمایش می‌دهد، اما فایل یک تعریف XAML است. یک خاصیت در کد XAML با نام *StartupUri* خوانده می‌شود و به فایل Window1.xaml نشان داده شده در اینجا اشاره می‌کند:



اگر علامت به اضافه (+) را مجاور App.xaml در *Solution Explorer* کلیک کنید، شما خواهید دید که در آنجا یک فایل Application.xaml.cs نیز است. اگر شما این فایل را دابل کلیک کنید، درخواهید یافت که دربردارنده کد زیر است:

```
using System;

using System.Collections.Generic;

using System.Configuration;

using System.Data;

using System.Linq;

using System.Windows;

namespace WPFHello

{

/// <summary>

/// Interaction logic for App.xaml

/// </summary>
```

```
public partial class App: Application

{

}

}
```

یک مرتبه دیگر، در آنجا تعدادی از دستورات *using* وجود دارد، اما نه خیلی، نه حتی یک متد *Main* . در حقیقت، *Main* در آنجاست، اما هنوزهم مخفی است. کد برای *Main* برمبنای تنظیمات در فایل App.xaml تولید می‌شود؛ به ویژه، *Main* فرم تعیین شده توسط خاصیت *StartupUri* را ایجاد و نمایش خواهد داد. اگر شما بخواهید یک متفاوت را نمایش دهید، فایل App.xaml را ویرایش می‌کنید.

زمان آن رسیده است که خودتان هم قدری کدنویسی کنید!

نوشتن کد برای دکمه OK

برگه *Window1.xaml* را بالای پنجره ویرایشگر کد ومتن را کلیک کنید تا Window1 در پنجره نمای طراحی نمایش یابد.

دکمه *OK* را روی فرم دابل کلیک کنید.

فایل Window1.xaml.cs در پنجره ویرایشگر کد و متن ظاهر می‌شود، اما یک متد جدید با نام *ok_Click* به آن اضافه شده است. ویژوال استودیو ۲۰۰۸ به طور خودکار کدی برای فراخوانی این متد هرگاه که کاربر دکمه *OK* را کلیک می‌کند، تولید می‌کند. این یک مثال از یک رویداد است و شما درباره‌اینکه رویدادها چگونه کار می‌کنند به محض اینکه از طریق کتا پیشرفت کردید، بیشتر یاد خواهید گرفت.

کد نشان داده شده در حالت **پررنگ** (**BOLD**) را به متد *ok_Click* اضافه کنید:

```
void ok_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show(“Hello “ + userName.Text);

}
```

این کدی است که هروقت کاربر دکمه *OK* را کلیک کند، اجرا می‌شود. هنوز خیلی نگران ترکیب نوشتاری این کد نباشید (تنها مطمئن شوید که آن را دقیقاً همان طور که نشان داده شده است، کپی کنید) زیرا در باره متدها در فصل ۳ بیشتر یاد خواهید گرفت. بخش جذاب دستور *MessageBox.Show*

است . این دستور یک جعبه پیغام حاوی متن "Hello" با هر اسمی که کاربر در جعبه متن اسم کاربر (username) تایپ کرده است روی فرم الحاق شده، نمایش می‌دهد.

برگه *Window1.xaml* را دربالای پنجره ویرایشگر کد و متن کلیک کنید تا Window1 دوباره در پنجره نمای طراحی (*Design View*) ظاهر شود.

در پایین ترین قاب درحال نمایش تعریف XAML فرم، عنصر *Button* را بررسی کنید، اما مواظب باشید که چیزی را تغییر ندهید. توجه کنید که *Button* حاوی یک عنصر با نام *Click* است که به متد *ok_Click* اشاره می‌کند:

```
<Button Height=”23” … Click=”ok_Click”>OK</Button>
```

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

هرگاه فرم ظاهر می‌شود، اسم خود را در جعبه متن تایپ کنید، و سپس *OK* را کلیک کنید. یک جعبه پیغام ظاهر می‌شود، در حالی که با اسم به شما خوشامد می‌گوید.

OK را در جعبه پیغام کلیک کنید.

جعبه پیغام بسته می‌شود.

فرم را ببندید.

- اگر می‌خواهید که به فصل بعدی بروید

ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۲ مراجعه کنید.

اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید

در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۲

کارکردن با متغیرها، عملگرها و عبارات

بعد از تکمیل این فصل، شما قادر خواهید بود که:

- دستورات، شناسه‌ها و واژه‌های کلیدی را یاد بگیرید.
- از متغیرها برای ذخیره اطلاعات استفاده کنید.
- با انواع داده اصلی کار کنید.
- از عملگرهای ریاضیاتی مانند علامت به اضافه (+) و علامت منها (–) استفاده کنید.
- متغیرها را (به صورت پله‌ای) کاهش و افزایش دهید.

در فصل ۱، "به C# خوش آمدید"، شما یاد گرفتید که چگونه از محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ برای ایجاد و اجرای یک برنامه کنسول و یک برنامه Windows Presentation Foundation (WPF) استفاده کنید. در این فصل، شما با عناصری از ترکیب نوشتاری/نحوی و معناشناختی ویژوال C#، به انضمام دستورها، واژه‌های کلیدی و شناسه‌ها آشنا می‌شوید. شما انواع اصلی که برای زبان C# توکار هستند و خصوصیات مقادیری که هر نوع نگه می‌دارد را مطالعه خواهید کرد. در ضمن خواهید دید که چگونه متغیرهای محلی (متغیرهایی که تنها در یک متد یا بخش کوچک دیگری از کد وجود دارد) را اعلان کرده و استفاده کنید، درباره عملگرهای ریاضیاتی که C# عرضه می‌کند یاد خواهید گرفت، درخواهید یافت که چگونه از عملگرها برای دست کاری مقادیر استفاده کنید و یاد خواهید گرفت که چگونه عبارات حاوی دو یا چند عملگر را کنترل کنید.

درک دستورات

یک دستور (*statement*) فرمانی است که یک عمل را انجام می‌دهد. شما دستورات را ترکیب می‌کنید تا متدها را ایجاد کنید . در فصل ۳، بخش " نوشتن متدها و اعمال سطح دسترسی " درمورد متدها بیشتر یاد خواهید گرفت، اما فی الحال، یک متد را به عنوان یک دنباله مشخصی از دستورات تصور کنید. *Main*، که در فصل پیش معرفی شد، یک نمونه برای یک متد است. دستورات در C# از یک مجموعه خوش تعریف از قوانین که فرمت و ساختار دستورات را شرح می‌دهند، پیروی می‌کنند. این قوانین مجموعاً به عنوان ترکیب نوشتاری/نحوی (*syntax*) شناخته می‌شوند. (در مقابل، مشخصات آن چه که دستورات انجام می‌دهند، مجموعاً به عنوان معناشناختی (*semantics*) شناخته می‌شوند.) یکی از ساده ترین و مهم ترین قوانین ترکیب نوشتاری/نحوی C# می‌گوید که شما باید همه دستورات را با یک نقطه ویرگول (**;**) خاتمه دهید. برای مثال، نقطه ویرگول خاتمه دهنده اش، دستور زیر کامپایل نخواهد شد:

```
Console.WriteLine(“Hello World”);
```

توضیح C# یک زبان فرمت آزاد (“free format”) است، که یعنی فضای سفید، مانند یک کاراکتر جای خالی یا یک سطر جدید، با معنا نیستند به جز وقتی که به عنوان یک تفکیک کننده باشند.

فوت وفن برنامه نویسی خوب در هر زبانی یاد گرفتن ترکیب نوشتاری/ نحوی و معناشناسی زبان و سپس استفاده از زبان در یک شیوه اصطلاحی و طبیعی است. این رویکرد برنامه‌های شما را به سهولت بسیار قابل نگه داری می‌کند. در فصول سرتاسر این کتاب، شما مثال‌های بسیار مهمی‌از دستورات C# را خواهید دید.

استفاده از شناسه‌ها

شناسه‌ها اسامی هستند که شما برای مشخص کردن عناصر در برنامه خود، مانند فضاهای اسمی، کلاس‌ها، متدها و متغیرها استفاده می‌کنید (شما به زودی درباره متغیرها مطالبی را یاد خواهید گرفت). در C#، شما باید هنگام انتخاب شناسه‌ها قوانین ترکیب نوشتاری/نحوی زیر وفادار بمانید :

- شما تنها می‌توانید از کاراکترهای حروف (بزرگ و کوچک)، ارقام و زیر خط استفاده کنید.
- یک شناسه باید با یک حرف شروع شود (یک زیر خط به عنوان یک حرف لحاظ می‌شود).

برای مثال *result*، *score_*، *footballTeam* و *plan9* شناسه‌های معتبر هستند، در حالی که *result%*، *footballTeam$* و *9plan* شناسه‌های نامعتبر هستند.

مهم C# یک زبان حساس نسبت به حالت حروف است : *footballTeam* و *FootballTeam* شناسه‌های یکسانی نیستند.

شناسایی واژه‌های کلیدی

زبان C#، ۷۷ شناسه را برای استفاده خودش رزرو می‌کند و شما نمی‌توانید از این شناسه‌ها را برای مقاصد شخصی خود استفاده مجدد کنید. این شناسه‌ها واژه‌های کلیدی خوانده می‌شوند و هر یک از آنها یک معنی به خصوص دارند. نمونه‌هایی از این کلید واژه‌ها *class*، *namespace* و *using* هستند. شما معنای اغلب این کلید واژه‌ها را به محض اینکه در این کتاب پیش رفتید، یاد خواهید گرفت. کلید واژه‌ها در جدول متعاقب فهرست شده اند.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

توضیح در پنجره ویرایشگر کد و متن ویژوال استودیو ۲۰۰۸، واژه‌های کلیدی هنگامی که شما آنها را تایپ می‌کنید، آبی رنگ می‌شوند.

C# از شناسه‌های زیر نیز استفاده می‌کند. این شناسه‌ها توسط C# رزرو نشده اند، یعنی اینکه شما می‌توانید از این اسامی به عنوان شناسه‌ها برای متدها، متغیرها و کلاس‌های شخصی مال خودتان استفاده کنید، اما شما باید در واقع از انجام این کار بپرهیزید گویی که ابداً امکان پذیر نیست.

from	join	select	yield

get	let	set
group	orderby	value
into	partial	where

استفاده از متغیرها

یک متغیر یک موقعیت انبلره است که یک مقدار را نگه می‌دارد. شما می‌توانید یک متغیر را همانند یک جعبه در حافظه کامپیوتر برای نگه داشتن اطلاعات موقتی، تصور کنید. شما باید به هر متغیر در یک برنامه یک اسم نامبهم دهید که به طور منحصر به فردی آن را در متنی که در آن استفاده می‌شود، مشخص می‌کند. شما از اسم یک متغیر برای ارجاع به مقداری که متغیر نگه می‌دارد استفاده می‌کنید. برای مثال، اگر می‌خواهید که مقدار بهای یک آیتم در یک مغازه را ذخیره کنید، شما ممکن است یک متغیر با نام *cost* را به سادگی اعلان کنید و بهای آیتم را در این متغیر ذخیره کنید. بعدها، اگر شما به متغیر *cost* مراجعه کنید، مقدار بازیابی شده بهای آیتمی‌خواهد شد که قبلاً در آنجا ذخیره کردید.

نام گذاری متغیرها

شما بایست یک قرارداد نام گذاری برای متغیرها را بپذیرید که به شما کمک می‌کند تا از درهم برهمی‌درباب متغیرهایی که تعریف کرده‌اید، اجتناب کنید. فهرست زیر حاوی برخی توصیه‌های عمومی‌است:

- از زیرخطها در شناسه‌ها استفاده نکنید.
- شناسه‌هایی را ایجاد نکنید که تنها توسط حالت حروفشان با هم فرق دارند. برای مثال، یک متغیر بانام *myVariable* و دیگری با نام *MyVariable* برای استفاده در یک زمان ایجاد نکنید، زیرا خیلی راحت می‌شود آنها را با هم قاطی کرد.

توجه استفاده از شناسه‌هایی که تنها به واسطه حالت حروف متفاوت هستند، می‌تواند قابلیت استفاده مجدد از کلاس‌ها را در برنامه‌های طراحی شده توسط زبان‌های دیگر که نسبت به حالت حساس نیستند مانند ویژوال بیسیک، را محدود کند.

- نام را با حروف کوچک آغاز کنید.
- در یک شناسه چند کلمه‌ای، دومین و هر کلمه مابعد را با یک حرف بزرگ آغاز کنید. (این کار علامت گذاری حالت شتری خوانده می‌شود. مثال : *forPrintOfText*)
- از نمادگذاری مجاری (کولی) استفاده نکنید. (طراحان مایکروسافت ویژوال C++ که درحال خواندن این کتاب هستند، احتمالاً با نمادگذاری مجاری (کولی) آشنا هستند. اگر نمی‌دانید که نمادگذاری مجاری (کولی) چیست، خیلی نگران نباشید!)

مـهم شما بایست با دو توصیه اول پیشین همانند حرکات اجباری برخورد کنید، زیرا آنها به قبول مشخصات زبان عمومی(CLS) مربوط هستند. اگر شما بخواهید برنامه‌هایی بنویسید که با زبان‌های دیگر مانند ویژوال بیسیک .NET. عمل متقابل داشته باشد، باید با این توصیه‌ها مطابقت داشته باشید و انها را برآورید.

برای مثال، *score*، *footballTeam*، *_score* و *FootballTeam* شناسه‌های معتبر هستند، اما تنها دو تای اولی توصیه می‌شوند.

اعلان متغیرها

متغیرها مقادیر را نگه می‌دارند. C# انواع متفاوتی از مقادیر دارد که می‌تواند آنها را ذخیره و پردازش کند--- اعداد صحیح (integerها)، اعداد ممیز شناور و رشته‌هایی از کاراکترها، به منظور اسم بردن از سه نمونه. هرگاه شما یک متغیر را اعلان می‌کنید، باید نوع داده‌ای را که متغیر نگه دارد، تعیین کنید. شما نوع و اسم یک متغیر را در یک دستور اعلان، معرفی می‌کنید. برای مثال، دستور زیر اعلان می‌کند که متغیر با نام *age* مقادیر *int* (صحیح / integer) را نگه می‌دارد. به طوری که همواره، دستور باید با یک نقطه-ویرگول خاتمه یابد.

```
int age;
```

نوع متغیر *int* اسم یکی از انواع اصلی C#، یعنی صحیح/ integer است، که یک عدد کامل است. (بعداً درباره انواع داده اصلی جورواجور در همین فصل یاد خواهید گرفت.)

تـوجه برنامه نویسان ویژوال بیسیک بایست توجه کنند که C# اجازه اعلان ضمنی متغیرها را نمی‌دهد. شما باید به طور صریح همه متغیرها را قبل از استفاده از آنها اعلان کنید.

بعد از این که شما یک متغیر را اعلان کرده‌اید، شما می‌توانید یک مقدار به آن تخصیص دهید. دستور زیر مقدار ۴۲ را به *age* تخصیص می‌دهد. دوباره، شما خواهید دید که نقطه ویرگول لازم می‌شود.

```
age = 42;
```

علامت تساوی (=) عملگر تخصیص است، که مقدار واقع در سمت راستش را به متغیر واقع در سمت چپش تخصیص می‌دهد. بعد از این تخصیص، متغیر *age* می‌تواند در کدتان برای مراجعه به مقداری که نگه می‌دارد، به کار برده شود. دستور بعدی مقدار متغیر *age*، ٤٢، را به کنسول می‌نویسد:

```
Console.WriteLine(age);
```

توضیح اگر اشاره گر ماوس را روی یک متغیر در پنجره ویرایشگر کد و متن ویژوال استودیو ۲۰۰۸ رها کنید، یک صفحه توضیح (ScreenTip) ظاهر می‌شود، در حالی که نوع متغیر را به شما می‌گوید.

کار با انواع داده اصلی

C# تعدادی انوع توکار بانام انواع داده اصلی (*primitive data types*) دارد. جدول زیر انواع داده اصلی در C# را که خیلی عادی به کار می‌روند و دامنه مقادیری که شما می‌توانید در هر کدام ذخیره کنید را فهرست می‌کند.

نوع	توضیح	اندازه (به بیت)	دامنه	نمونه کاربرد
int	اعداد کامل	۳۲	-2^{31} تا $2^{31}-1$	int count; count = 42;
long	اعداد کامل (دامنه بزرگتر)	۶۴	-2^{63} تا $-2^{63}-1$	long wait; wait = 42L;
float	اعداد ممیز شناور	۳۲	$\pm 1.5\times10^{45}$ تا $\pm 3.4\times10^{38}$	float away; away = 0.42F;
double	اعدادممیز شناور با دقت مضاعف (خیلی دقیق تر)	۶۴	$\pm 5.0\times10^{-324}$ تا $\pm 1.7\times10^{308}$	double trouble; trouble = 0.42;
decimal	مقادیر مالی	۱۲۸	۲۸ رقم معنادار	decimal coin; coin = 0.42M;
string	دنباله‌ای از کاراکترها	۱۶ بیت به ازای هر کاراکتر	اعمال نمی‌شود	string vest; vest = “fortytwo”;
char	کاراکتر منفرد	۱۶	0 تا $2^{61}-1$	char grill; grill = ‘x’;
bool	بولی (Boolean)	۸	true یا false	bool teeth; teeth = false;

متغیرهای محلی تخصیص نیافته

هرگاه یک متغیر اعلان می‌کنید، متغیر حاوی یک مقدار تصادفی است تازمانی که شما یک مقدار را به

آن تخصیص دهید. این رفتار منبع غنی از باگ‌ها در برنامه‌های C و C++ بود طوری که یک متغیر را ایجاد می‌کردند و قبل از دادن یک مقدار به آن به طور تصادفی آن را به عنوان یک منبع اطلاعات به کار می‌بردند. C# به شما اجازه نمی‌دهد تا از یک متغیر تخصیص نیافته استفاده کنید. شما قبل از این که بتوانید از متغیر استفاده کنید باید یک مقدار را به متغیر تخصیص دهید؛ در غیر این صورت، برنامه شما ممکن است کامپایل نشود. این الزام قاعده تخصیص معین خوانده می‌شود. برای مثال، دستورات زیر یک خطای زمان کامپایل تولید خواهند کرد زیرا *age* تخصیص نیافته است:

```
int age;
Console.WriteLine(age);        // compile-time error
```

نمایش دادن مقادیر نوع داده اصلی

در فعالیت زیر، شما از یک برنامه C# با نام *PrimitiveDataTypes* استفاده خواهید کرد تا نشان دهید که چگونه انواع داده اصلی کار می‌کنند.

نمایش مقادیر نوع داده اصلی

ویژوال استودیو ۲۰۰۸ را اگر هنوز در حال اجرا نیست، شروع کنید.

اگر درحال استفاده از Visual Studio 2008 Standard Edition یا Visual Studio 2008 Professional Edition هستید، در منوی *File*، ابتدا گزینه *Open* و سپس *Project/Solution* را کلیک کنید.

اگر در حال استفاده Visual C# 2008 Express Edition هستید، در منوی *File*، گزینه *Open Project* را کلیک کنید.

جعبه گفتگوی *Open Project* ظاهر می‌شود.

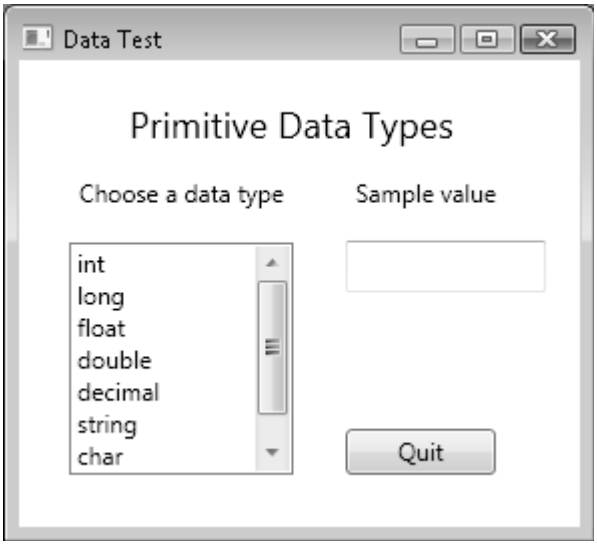
به پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 2\PrimitiveDataTypes در پوشه Documents خودتان بروید. فایل محلول PrimitiveDataTypes را انتخاب کنید و سپس *Open* را کلیک کنید.

محلول بارگذاری می‌شود و *Solution Explorer* (مرورگر محلول) پروژه PrimitiveDataTypes را نمایش می‌دهد.

توجه اسامی فایل محلول پسوند sln. دارند، مانند PrimitiveDataTypes.sln . یک محلول می‌تواند حاوی یک یا چند پروژه باشد. فایل‌های پروژه پسوند csproj. دارند. در صورتی که شما یک پروژه را به جای یک محلول باز کنید، ویژوال استودیو ۲۰۰۸ به طور خودکار یک فایل محلول برای آن ایجاد می‌کند. اگر محلول را بنا کنید، ویژوال استودیو ۲۰۰۸ به طور خودکار هرگونه فایل جدید یا به روز شده را ذخیره می‌کند، از این رو از شما در خواست می‌شود تا یک اسم و موقعیت برای فایل محلول جدید تدارک ببینید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

پنجره برنامه زیر ظاهر می‌شود:



در لیست *Choose a data type*، نوع string (رشته) را کلیک کنید.
مقدار "forty two" در جعبه *Sample value* ظاهر می‌شود.
نوع int (صحیح) را در لیست کلیک کنید.
مقداری برای انجام دادن در جعبه *Sample value* ظاهر می‌شود، نشان دهنده‌اینکه برای نمایش یک مقدار int هنوز لازم است که دستوراتی نوشته شوند.
هر نوع داده را در لیست کلیک کنید. تصدیق کنید که کد لازم برای انواع double و bool نیز باید کامل شوند.
Quit را برای بستن پنجره و توقف برنامه کلیک کنید.
کنترل به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ برمی‌گردد.

استفاده از انواع داده اصلی در کد

در مرورگر محلول (*Solution Explorer*)، *Window1.xaml* را کلیک کنید.
فرم WPF برای برنامه در پنجره نمای طراحی (*Design View*)ظاهر می‌شود.
در هرجای پنجره نمای طراحی (*Design View*) را درحال نمایش فرم Window1.xaml راست کلیک کنید و پس از آن *View Code* را کلیک کنید.
پنجره ویرایشگر کد ومتن در حال نمایش فایل Window1.xaml.cs باز می‌شود.

توجه به یاد داشته باشید که شما می‌توانید از مرورگر محلول (*Solution Explorer*) نیز برای دسترسی به کد استفاده کنید؛ علامت به اضافه (+) را در سمت چپ فایل Window1.xaml کلیک کنید و پس از آن Window1.xaml.cs را دابل کلیک کنید.

در پنجره ویرایشگر کد و متن، متد *showFloatValue* را پیدا کنید.

توضیح برای موقعیت یابی یک آیتم در پروژه خود، منوی *Edit* را کلیک کنید سپس *Find and Replace* و بعد از آن *Quick Find* را انتخاب کنید. یک جعبه گفتگو در حالی که می‌پرسد چه چیزی را می‌خواهید جستجو کنید، باز می‌شود. اسم آیتمی‌را که در حال جستجویش هستید تایپ کنید و سپس *Find Next* را کلیک کنید. به طور پیش فرض، جستجو حساس به حالت حروف نیست. اگر می‌خواهید که یک جستجوی حساس به حالت انجام دهید، دکمه به اضافه، +، را پهلوی برچسب *Find Options* برای نمایش گزینه‌های اضافی، کلیک کنید و جعبه چک *Match Case* را انتخاب کنید. اگر زمان داشته باشید می‌توانید با گزینه‌های دیگر نیز امتحان کنید.

در ضمن شما می‌توانید Ctrl+F را فشار دهید (کلید کنترل را فشرده و سپس F را فشار دهید) تا جعبه گفتگوی *Quick Find* به جای استفاده از منوی *Edit* نمایش یابد. به طور مشابه، شما می‌توانید Ctrl+H را برای نمایش جعبه گفتگوی *Quick Replace* فشار دهید.

هنگامی که نوع *float* را در جعبه لیست کلیک کنید، متد *showFloatValue* اجرا می‌شود. این متد حاوی سه دستور زیر است:

```
float variable;
variable=0.42F;
value.Text = "0.42F";
```

اولین دستور یک متغیر با نام *variable* از نوع *float* اعلان می‌کند.
دستور دوم مقدار 0.42F را به *variable* تخصیص می‌دهد. (F یک پسوند نوع است که تعیین می‌کند که با 0.42 بایست به عنوان یک مقدار *float* برخورد شود. اگر شما F را فراموش کنید، با مقدار 0.42 به عنوان یک *double* برخورد خواهدشد و برنامه شما کامپایل نخواهد شد زیر نمی‌تواند یک مقدار از یک نوع را به یک متغیر از یک نوع متفاوت بدون نوشتن کد اضافی تخصیص دهد--- #C در این رابطه خیلی سخت گیر است.)
سومین دستور مقدار این متغیر را در جعبه متن *value* روی فرم نمایش می‌دهد. این دستور نیازمند اندکی توجه شما است. شیوه‌ای که در آن یک آیتم را در یک جعبه متن نمایش می‌دهید، تنظیم کردن خاصیت *Text* آن است. توجه کنید که به خاصیت یک شیء با استفاده از همان یادداشت نقطه گذاری که برای اجرای یک متد دیدید، دسترسی پیدا می‌کنید. (*Console.WriteLine* را از فصل ۱ به یاد آورید.) داده‌ای که شما در خاصیت *Text* جای می‌دهید باید یک رشته باشد (دنباله‌ای از کاراکترها محصور شده در علائم نقل قول مضاعف) و نه یک عدد. اگر سعی کنید تا یک عدد را به خاصیت *Text* تخصیص دهید، برنامه شما کامپایل نخواهد شد. در این برنامه، دستور به سادگی متن "0.42F" را در جعبه متن نمایش می‌دهد. در یک برنامه دنیای واقعی، شما دستوراتی را اضافه می‌کنید که مقدار متغیر *variable* را به یک رشته (string) تبدیل می‌کنند و سپس آن را در خاصیت *Text* جای می‌دهند، اما قبل از اینکه بتوانید این کار را بکنید، نیاز دارید که اندکی بیشتر درباره #C و .NET Microsoft Framework بدانید.
در پنجره ویرایشگر کد و متن، متد *showIntValue* را پیدا کنید. این متد مانند این به نظر می‌رسد:

```
private void showIntValue()
{
    value.Text = "to do";
}
```

هنگامی که شما نوع *int* را در جعبه فهرست کلیک کنید، متد *showIntValue* فراخوانده می‌شود.

توضیح روش دیگر برای پیدا کردن یک متد در پنجره ویرایشگر کد و متن کلیک کردن لیست پایین افتادنی *Members* در سمت راست بالای پنجره است. این پنجره فهرستی از همه متدها (و آیتم‌های دیگر) در کلاس نمایش یافته در پنجره ویرایشگر کد و متن را نمایش می‌دهد. شما می‌توانید اسم یک عضو را کلیک کنید و به طورمستقیم آن را در پنجره ویرایشگر کد و متن خواهید گرفت.

دو دستور زیر را در سرآغاز متد *showIntValue*، در روی یک سطر جدید بعد از کروشه باز، تایپ کنید، همان طور که با فونت درشت در کد زیر نشان داده شده است:

```
private void showIntValue()
{
  int variable;
  variable = 42;
}
```

در دستور اصلی در این متد، رشته "*to do*" را به "*42*" تغییر دهید.
اکنون متد بایست دقیقاً مانند این به نظر برسد:

```
private void showIntValue()
{
  int variable;
  variable = 42;
  value.Text = "42";
}
```

توجه اگر شما تجربه برنامه نویسی قبلی دارید، شما ممکن است دچار وسوسه شوید که سومین دستور را این گونه تغییر دهید

```
value.Text = variable;
```

به نظر می‌رسد که‌این دستور بایست مقدار متغیر *variable* را در جعبه متن روی فرم نمایش دهد. اما، C# بررسی نوع سخت گیرانه‌ای انجام می‌دهد؛ جعبه متن‌ها تنها می‌توانند مقادیر *string* را نمایش دهند و *variable* یک *int* است، از این رو این دستور کامپایل نخواهد شد. شما بعداً در همین فصل خواهید دید که چگونه مابین مقادیر عددی و مقادیر رشته‌ای تبدیل انجام دهید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید. فرم دوباره ظاهر می‌شود.
نوع *int* را در لیست *Choose a data type* انتخاب کنید. تصدیق کنید که مقدار 42 در جعبه متن *Sample value* نمایش داده می‌شود.
Quit را کلیک کنید تا نجره بسته شده و برنامه متوقف شود.
در پنجره ویریشگر کد و متن، متد *showDoubleValue* را پیدا کنید.
متد *showDoubleValue* را دقیقاً همان طور که در فونت درشت در کد زیر نشان داده شده است، ویرایش کنید:

```
private void showDoubleValue()
{
  double variable;
  variable = 0.42;
  value.Text = "0.42";
}
```

در پنجره ویرایشگر کد و متن، متد *showBoolValue* را پیدا کنید.
متد *showBoolValue* را دقیقاً مانند زیر ویرایش کنید:

```
private void showBoolValue()
{
  bool variable;
  variable = false;
  value.Text = "false";
}
```

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
در لیست *Choose a data type*، انواع *int*، *double* و *bool* را انتخاب کنید. در هر حالت، بررسی کنید که مقدار صحیح در جعبه متن *Sample value* نمایش داده می‌شود.
Quit را برای توقف برنامه کلیک کنید.

استفاده کردن از عملگرهای ریاضیاتی

C# عملیات‌های ریاضیاتی منظمی‌را پشتیبانی می‌کند که شما در کودکی خود فرا گرفته‌اید: علامت جمع (+) برای جمع کردن، علامت منها (–)برای تفریق کردن، ستاره (*) برای ضرب کردن و فوروارد اسلش (/) برای تقسیم کردن. علائم +، –، * و / عملگر خوانده می‌شوند زیرا آنها روی مقادیری برای ایجاد مقادیر جدید عمل می‌کنند. در مثال زیر، متغیر *moneyPaidToConsultant* حاصل ضرب ۷۵۰ (نرخ روزانه) و ۲۰ (تعداد روزهایی که مشاور استخدام شده) را در بردارد :

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```

توجه مقادیری که عملگر روی آنها عمل می‌کند، عملوند خوانده می‌شوند. در عبارت 20 * 750، * عملگر است و 750 و 20 عملوندها هستند.

عملگرها و نوع‌ها

همه عملگرها به همه انواع داده قابل اعمال نیستند. عملگرهایی که شما می‌توانید روی یک مقدار استفاده کنید بستگی به نوع مقدار دارد. برای مثال، شما می‌توانید از همه عملگرهای ریاضیاتی روی مقادیری از نوع *char*، *int*، *long*، *float*، *double* و *decimal* استفاده کنید. اما به استثنای عملگر جمع (+) شما نمی‌توانید از عملگرهای ریاضیاتی روی مقادیری از نوع *string* و *bool* استفاده کنید. از این رو دستور زیر مجاز نیست زیرا نوع *string* عملگر منها را پشتیبانی نمی‌کند (تفریق یک رشته از رشته دیگر بی معنی خواهد شد):

```
// compile-time error
Console.WriteLine("Gillingham" – "Forest Green Rovers");
```

شما می‌توانید از عملگر + برای چسباندن مقادیر رشته استفاده کنید. لازم است که مراقب باشید زیرا این عمل می‌تواند نتایجی را داشته باشد که شما انتظار ندارید. برای مثال، دستور زیر "431" (و نه "44") را به کنسول می‌نویسد:

```
Console.WriteLine("43" + "1");
```

توضیح چارچوب .NET. متدی با نام *Int32.Parse* عرضه می‌کند که اگر نیاز دارید که محاسبات ریاضیاتی روی مقادیر نگه داشته شده به عنوان رشته‌ها انجام دهید، شما می‌توانید از آن برای تبدیل یک مقدار رشته به یک عدد صحیح استفاده کنید.

در ضمن شما بایست بدانید که که نوع نتیجه یک عملیات ریاضیاتی به نوع عملوندهای به کاررفته بستگی دارد. برای مثال، مقدار عبارت 5.0/2.0 برابر 2.5 است؛ نوع هر دو عملوند *double* است (در C# ، لیترال‌های عددی با نقاط اعشاری، به منظور نگه داری بیشترین دقت ممکن، همواره *double* هستند، نه *float*)، از این رو نوع نتیجه نیز *double* است. گرچه مقدار عبارت 5/2 برابر 2 است. در این حالت، نوع هردو عملوند *int* است، از این رو نوع نتیجه نیز *int* است. C# همواره مقادیر را در شرایطی مانند این به پایین گرد می‌کند. اگر شما انواع عملوندها را ترکیب کنید، وضعیت اندکی پیچیده تر می‌شود. برای مثال، عبارت 5/2.0 از یک *int* و یک *double* تشکیل شده است. کامپایلر C# ناهم خوانی را شناسایی می‌کند و کدی تولید می‌کند که قبل از انجام دادن عملیات، *int* را به یک *double* تبدیل می‌کند. از این رو نتیجه عملیات یک *double* است (2.5). اما، با اینکه‌این روش کار می‌کند، به عنوان تمرینی ناچیز برای ترکیب انواع به‌این شیوه مورد توجه قرار می‌گیرد.

انواع عددی و مقادیر نامحدود

در اینجا یک یا دو مشخصه از اعداد در C# موجود است که شما بایست از آنها آگاه باشید. برای مثال، نتیجه تقسیم کردن هر عدد بر صفر بی نهایت است، که خارج از دامنه انواع *int*، *long* و *decimal* است و در نتیجه ارزیابی یک عبارت مانند 5/0 منتج به یک خطا می‌شود. اما، انواع *double* و *float* در واقع یک مقدار ویژه دارند که می‌تواند بیانگر بی نهایت باشد و مقدار عبارت 5.0/0.0 برابر *Infinity* (بی نهایت) است. یک استثنا برای این قاعده مقدار عبارت 0.0/0.0 است. معمولاً، اگر شما صفر را بر هر چیزی تقسیم کنید، نتیجه صفر است، اما اگر هر چیزی را بر صفر تقسیم کنید نتیجه بی نهایت است. عبارت 0.0/0.0 منتج به یک پاردوکس می‌شود--- مقدار باید همزمان صفر و بی نهایت باشد. C# مقدار ویژه دیگری برای این وضعیت دارد که *NaN* نام دارد، که مخفف "not a number" است. بنابر این اگر شما 0.0/0.0 را ارزیابی کنید، نتیجه *NaN* است. *NaN* و *Infinity* از طریق عبارات منتشر می‌شوند. اگر شما *NaN* + 10 را ارزیابی کنید، نتیجه *NaN* است و اگر *Infinity* + 10 را ارزیابی کنید، نتیجه *Infinity* است. یک استثنا برای این قاعده، عبارت *Infinity*0* است، که منتج به صفر می‌شود، در حالی که نتیجه عبارت 0 * *NaN* برابر *NaN* است.

در ضمن C# یک عملگر ریاضیاتی کمتر آشنا را پشتیبانی می‌کند : عملگر باقیمانده، یا قدر مطلق، که با علامت درصد (%) نمایش داده می‌شود. نتیجه *x* % *y* باقیمانده تقسیم *x* بر *y* است. برای مثال، 2 % 9 برابر 1 است، زیرا 9 تقسیم بر 2 برابر 4 و باقیمانده 1 است.

توجه اگر شما با C یا C++ آشنا باشید، شما خواهید دانست که نمی‌توانید در این زبان‌ها از عملگر باقیمانده روی انواع *double* یا *float* استفاده کنید. اما C# این قاعده را ول کرده است. عملگر باقیمانده با تمامی انواع عددی معتبر است و نتیجه لزوماً یک عدد صحیح نیست. برای مثال، نتیجه عبارت 2.4 % 7.0 برابر 2.2 است.

بررسی عملگرهای ریاضیاتی

فعالیت‌های زیر چگونگی استفاده از عملگرهای ریاضیاتی را روی مقادیر *int* با استفاده از یک برنامه از قبل نوشته شده C# با نام MathsOperators، شرح می‌دهد.

کار با عملگرهای ریاضیاتی

پروژه MathsOperators را که در پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 2\MathsOperators در پوشه Documents شما جای گرفته، باز کنید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید. یک فرم روی صفحه ظاهر می‌شود.

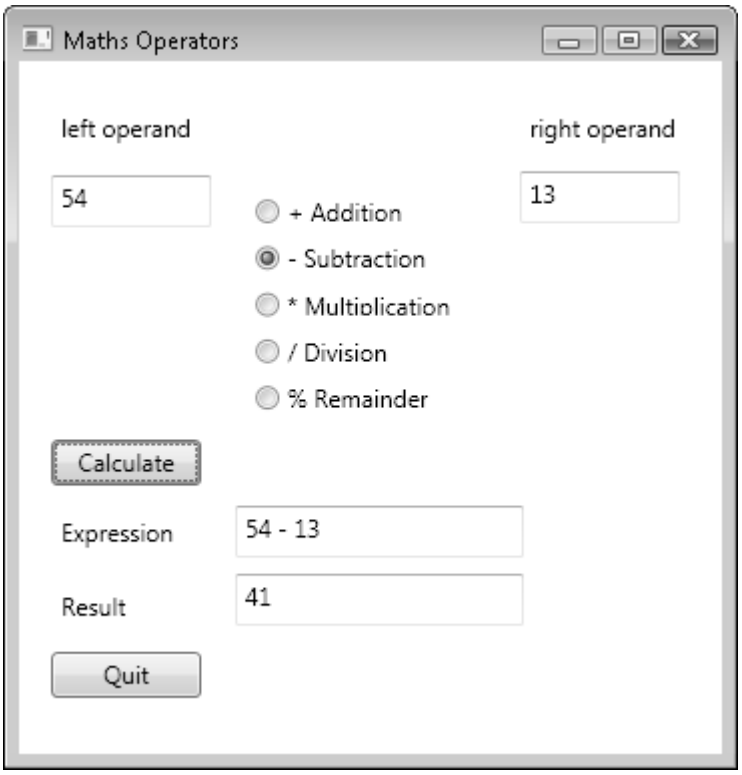
عدد **54** را در جعبه متن *left operand* تایپ کنید.

عدد **13**را در جعبه متن *right operand* تایپ کنید.

اکنون شما می‌توانید هر عملگری را به مقادیر واقع در جعبه‌های متن اعمال کنید.

دکمه *Subtraction* – را کلیک کنید و سپس *Calculate* را کلیک نمایید.

متن در جعبه متن *Expression* به 13 – 54 تغییر می‌کند و مقدار 41 در جعبه *Result* ظاهر می‌شود، همان طور که در تصویر زیر نشان داده شده است:



دکمه */ Division* را کلیک کنید و سپس *Calculate* را کلیک کنید.

متن واقع در عبارت *Expression* به 54/13 تغییر می‌کند و مقدار 4 در جعبه متن *Result* ظاهر می‌شود. در دنیای واقعی، 54/13 برابر 4.153846 است، اما این دنیای واقعی نیست؛ این عمل انجام دادن تقسیم صحیح در C# است و هنگامی که شما یک عدد صحیح را به عدد صحیح دیگری تقسیم می‌کنید، جوابی که شما می‌گیرید یک عدد صحیح است، همان طور که قبلاً توضیح داده شد.

دکمه *% Remainder* را کلیک کنید و سپس *Calculate* را کلیک کنید.

متن واقع در عبارت *Expression* به 54%13 تغییر می‌کند و مقدار 2 در جعبه متن *Result* ظاهر می‌شود. این امر به خاطر این است که باقیمانده تقسیم 54 بر 13 برابر 2 است. اگر شما گرد کردن ریاضیاتی را برای یک عدد صحیح در هر مرحله انجام دهید، ((13 * (54/13)) – 54) برابر 2 است --- معلم قدیمی‌من در مدرسه وحشت زده خواهد شد اگر به او گفته شود که 13 * (54/13) مساوی با 54 نیست!

ترکیب‌های دیگری از اعداد و عملگرها را بررسی کنید. هرگاه کار خود را تمام کردید، *Quit* را برای برگشتن به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ کلیک کنید.

اکنون یک نگاهی به کد برنامه MathsOperators بیاندازید.

بررسی کد برنامه MathsOperators

فرم Window1.xaml را در پنجره *Design View* نمایش دهید (فایل *Window1.xaml* را در مرورگر محلول دابل کلیک کنید).

در منوی *View*، گزینه *Other Windows* و سپس *Document Outline* را کلیک کنید. پنجره *Document Outline* ظاهر می‌شود، درحالی که اسامی و انواع کنترل‌های روی فرم را نشان می‌دهد. اگر هر یک از کنترل‌های روی فرم را کلیک کنید، اسم کنترل در پنجره *Document Outline* پررنگ می‌شود. به طور مشابه، اگر یک کنترل را در پنجره *Document Outline* انتخاب کنید، کنترل متناظر در پنجره *Design View* انتخاب می‌شود.

در روی فرم، دو کنترل *TextBox* که در آنها کاربر اعداد را تایپ می‌کند، کلیک کنید. در پنجره *Document Outline*، بررسی کنید که آنها *lhsOperand* و *rhsOperand* نامیده شده اند. (شما می‌توانید نام یک کنترل را در میان پارانتزها در سمت راست کنترل ببینید.) هرگاه فرم اجرا می‌شود، خاصیت *Text* هر یک از این کنترل‌ها مقادیری را که کاربر وارد می‌کند، نگه می‌دارند.

به طرف پایین فرم، بررسی کنید که کنترل *TextBox* که برای نمایش عبارت ارزیابی شده به کار رفته expression نامیده شده است و کنترل *TextBox* که برای نمایش نتیجه محاسبه به کاررفته است، result نامیده شده است.

پنجره *Document Outline* را ببندید.

کد برای فایل Window1.xaml.cs را در پنجره ویرایشگر کد ومتن نمایش دهید.

در پنجره ویرایشگر کد و متن، متد subtractValues را پیدا کنید. این متد مانند زیر به نظر می‌رسد:

```
private void subtractValues()
{
int lhs = int.Parse(lhsOperand.Text);
int rhs = int.Parse(rhsOperand.Text);
int outcome;
outcome = lhs – rhs;
expression.Text = lhsOperand.Text + " – " + rhsOperand.Text;
result.Text = outcome.ToString();
}
```

دستور اول در این متد یک متغیر *int* با نام *lhs* اعلان می‌کند و آن را با عدد صحیح متناظر با مقدار تایپ شده توسط کاربر در جعبه متن *lhsOperand* مقداردهی می‌کند. به یاد داشته باشید که خاصیت *Text* یک کنترل جعبه متن دربردارنده یک رشته (string) است، از این رو شما قبل از اینکه بتوانید آن را به متغیر *int* تخصیص دهید، باید این رشته را به یک عدد صحیح تبدیل کنید. نوع داده *int* متد *int.Parse* را عرضه می‌کند که به دقت این کار را انجام می‌دهد.

دستور دوم یک متغیر *int* با نام *rhs* اعلان می‌کند و آن را با مقدار واقع در جعبه متن *rhsOperand* بعد از تبدیل کردن آن به *int*، مقداردهی می‌کند.

سومین دستور یک متغیر *int* با نام *outcome* اعلان می‌کند.

چهارمین دستور مقدار متغیر *rhs* را از مقدار متغیر *lhs* کم می‌کند و نتیجه را به *outcome* تخصیص می‌دهد.

دستور چهارم سه رشته را بـــه هـــم الحـــاق می‌کنـد (بـا اسـتفاده از عملگـر +) در حـالی کـه نشـان می‌دهد که محاسبه انجام شده است و نتیجه را به خاصیت *expression.Text* تخصیص مـی‌دهـد. ایـن کار باعث می‌شود که رشته در جعبه متن *expression* در روی فرم نمایش یابد.
دستور ششم نتیجه محاسبه را به واسطه تخصیص دادن آن به خاصیت *Text* جعبه مـتن *result* نمـایش می‌دهد. به یاد داشته باشید که خاصیت *Text* یک رشته (string) است و اینکه نتیجه محاسـبه یـک *int* است، از این رو قبل از تخصیص دادن نتیـجه محاسبه به خاصیت *Text*، شـــــما باید *int* را بـه رشـته (string) تبدیل کنید. این همان کاری است که متد *ToString* نوع *int* انجام می‌دهد.

متد ToString

هر کـلاس در چـارچوب NET. یـک متـد *ToString* دارد. هـدف از *ToString* تبـدیل یـک شـیء بـه نمـــایش رشته اش است. در مثال قبلی، متد *ToString* برای شـیء عـدد صـحیح، *outcome*، بـرای تبـدیل مقـدار صحیح *outcome* به مقدار رشته معـادل بـه کـار بـرده شـده اسـت. ایـن تبـدیل لازم اسـت زیـرا مقـدار در خاصیت *Text* جعبه متن *result* به کـار بـرده می‌شـود--- خاصـیت *Text* تنهـا دربردارنـده رشـته‌ها اسـت. هرگـاه کلاس‌هـای مـال خـود را ایجـاد می‌کنیـد، شـما می‌توانیـد پیـاده سـازی شخصـی خـود را از متـد *ToString* برای تعیین اینکه کلاس شما چگونه بایست بـه عنـوان یـک رشـته نمـایش داده شـود، تعریـف کنید. در فصل ۷، بخش "ایجاد کردن و مدیریت کلاس‌ها واشیاء " درباره‌ایجاد کـردن کلاس‌هـای شخصـی خودتان بیشتر یاد خواهید گرفت.

کنترل کردن حق تقدم

حق تقدم (*Precedence*) ترتیبی را که در آن عملگرهای یک عبارت ارزیـابی مـی‌شـوند، کنتـرل مـی‌کنـد. عبارت زیر را ملاحظه کنید که از عملگرهای + و * استفاده می‌کند:

```
2 + 3 * 4
```

این عبارت به طور بالقوه‌ای مبهم و دو پهلوست؛ شما در ابتدا جمع را انجـام مـی‌دهیـد یـا ضـرب را ؟ بـه عبارت دیگر، آیا عملگر + به سمت چپ 3 می‌چسبد یا عملگـر * بـه سـمت راسـتش ؟ ترتیـب عملیاتهـا مهم است زیرا نتیجه را تغییر می‌دهد:

- اگر شما در ابتدا عملیات جمع و متعاقب آن ضرب را انجام دهیـد، نتیجـه (3 + 2) عملونـد چپـی عملگر * را شکل می‌دهد و نتیجه کامل عبارت برابر 4 * 5 است، یعنی 20 .

- اگر شما ابتـدا ضـرب و متعاقـب آن جمـع را انجـام دهیـد، نتیجـه ضـرب (4 * 3) عملونـد سـمت راستی عملگر + را شکل می‌دهد و نتیجه کامل عبارت برابر 12 + 2 است، یعنی 14 .

در C#، عملگرهای افزاینده (*، / و %) بر همه عملگرهای جمعی (+ و -) حق تقدم دارند، از این رو در عباراتی مانند 4 * 3 + 2، ضرب در ابتدا انجام می‌شود و متعاقب آن عمل جمع انجام می‌شود. از این رو جواب 4 * 3 + 2 برابر 14 است. به محض اینکه هر عملگر جدید در فصول بعدی بحث شود، حق تقدمش توضیح داده خواهد شد.

شما می‌توانید از پارانتزها برای باطل کردن حق تقدم استفاده کنید و عملوندها را وادار کنید که به شیوه‌ای متفاوت به عملگرها مقید شوند. برای مثال، در عبارت زیر، پارانتزها 2 و 3 را وادار می‌کنند که به عملگر + ملتزم شوند (با ایجاد 5) و نتیجه‌این جمع عملوند دست چپی عملگر * را برای تولید مقدار 20 شکل می‌دهد:

```
(2 + 3) * 4
```

توجه عبارت پارانتزها (*parentheses*) یا دوهلال‌ها (*round brackets*) به ( ) اشاره می‌کنند. عبارت آکولادها (*braces*) یا (*curly brackets*) به { } اشاره می‌کنند. عبارت کروشه‌ها (*square brackets*) یا قلاب‌های گوشه دار به [ ] اشاره می‌کنند.

استفاده از شرکت پذیری برای ارزیابی عبارت‌ها

حق تقدم عملگر تنها نیمی‌از داستان است. هنگامی که یک عبارت حاوی عملگرهای متفاوتی است که حق تقدم یکسان دارند، چه اتفاقی می‌افتد ؟ اینجا، همان جایی است که شرکت پذیری مهم می‌شود. شرکت پذیری جهتی است (چپ یا راست) که در آن عملوندهای یک عملگر ارزیابی می‌شوند. به عبارت زیر توجه کنید که از عملگرهای / و * استفاده می‌کند:

```
4 / 2 * 6
```

این عبارت هنوز هم به طور بالقوه‌ای مبهم و دوپهلو است. آیا شما ابتدا تقسیم را انجام می‌دهید یا ضرب را؟ حق تقدم هر دو عملگر یکسان است (آنها هردو عملگرهای افزایشی هستند)، اما ترتیبی که در آن عبارت ارزیابی می‌شود مهم است زیرا شما یکی از دو نتیجه ممکن زیر را کسب می‌کنید:

- اگر در ابتدا تقسیم را انجام دهید، نتیجه تقسیم (4/2) عملوند دست چپی عملگر * را شکل می‌دهد و نتیجه عبارت کامل برابر 6 * (4/2) است، یعنی 12 .

- اگر در ابتدا ضرب را انجام دهید، نتیجه ضرب (6 * 2) عملوند دست راستی عملگر / را شکل می‌دهد و نتیجه عبارت کامل برابر 4/(6 * 2) است، یعنی 4/12 .

در این حالت، شرکت پذیری عملگرها مشخص می‌کند که عبارت چگونه ارزیابی می‌شود. عملگرهای * و / هر دو چپ-شرکت پذیر هستند، به معنای این است که عملوندها از سمت چپ به راست ارزیابی می‌شوند. در این حالت، 4/2 قبل از ضرب شدن در 6 ارزیابی می‌شود، و نتیجه 12 را می‌دهد. به محض اینکه عملگرهای جدیدی در فصول آتی بحث شوند، شرکت پذیری آنها نیز پوشش داده خواهد شد.

شرکت پذیری و عملگر تخصیص

در C#، علامت مساوی (=) یک عملگر است. همه عملگرها یک مقدار برمبنای عملوندهای شان برمی‌گردانند. عملگر تخصیص (=) متفاوت نیست. این عملگر دو عملوند می‌گیرد؛ عملوند دست راستی ارزیابی می‌شود و سپس در عملوند دست چپی ذخیره می‌شود. مقدار عملگر تخصیص مقداری است که به عملوند دست چپی تخصیص داده شده بود. برای مثال، در دستور تخصیص زیر، مقدار برگشتی توسط عملگر تخصیص 10 است، که در ضمن مقداری است که به متغیر *myInt* تخصیص داده شده است:

```
int myInt;
myInt = 10;        //value of assignment expression is 10
```

در این نقطه، شما احتمالاً فکر می‌کنید که‌این کار خیلی عالی و محرمانه است، اما پس چیست ؟ بسیار خوب، از آن جایی که عملگر تخصیص یک مقدار برمی‌گرداند، شما می‌توانید از همان مقدار با رخداد دیگری از دستور تخصیص استفاده کنید، مانند این:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

مقدار تخصیص داده شده به متغیر *myInt2* مقداری است که به *myInt* تخصیص یافته بود. دستور تخصیص مقدار یکسانی را به هر دو متغیر تخصیص می‌دهد. اگر شما بخواهید که چندین متغیر را با مقدار یکسان مقداردهی کنید، این تکنیک خیلی مفید است. این کار خواندن کد شما را برای هر کسی واضح و روشن می‌کند که همه متغیرها باید مقدار یکسان داشته باشند:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

از طریق این بحث، شما احتمالاً نتیجه گرفته‌اید که عملگر تخصیص از راست به چپ مربوط می‌شود. سمت راست ترین تخصیص اول رخ می‌دهد و مقدار تخصیص یافته از طریق متغیرها از راست به چپ منتشر می‌شود. در صورتی که هر متغیری از قبل مقدار داشته باشد، این مقدار با مقدار در حال تخصیص شدن رونویسی می‌شود.

افزایش و کاهش دادن متغیرها (به صورت پله‌ای)

اگر شما بخواهید که 1 را به یک متغیر اضافه کنید، می‌توانید از عملگر + استفاده کنید:

```
count = count + 1;
```

گرچه، اضافه کردن 1 به یک متغیر آن قدر متداول است که C# عملگر شخصی خود را تنها برای این منظور تدارک دیده است: عملگر ++ . برای نمو دادن متغیر *count* با یک واحد، شما می‌توانید دستور زیر را بنویسید:

```
count++;
```

به طور مشابه C# عملگر – – را عرضه کرده است تا شما بتوانید 1 را از یک متغیر کم کنید، مانند این:

```
count--;
```

توجه عملگرهای ++ و – – عملگرهای یکانی (*unary*) هستند، به‌این معنی که آنها تنها یک عملوند یگانه را می‌گیرند. آنها همانند عملگر یکانی !، حق تقدم و شرکت پذیری چپ را به اشتراک می‌برند.

پیشوندی و پسوندی

عملگرهای افزایش، ++، و کاهش، – –، غیرعادی هستند از این حیث که شما می‌توانید آنها یا قبل و یا بعد از متغیر قرار دهید. جای دادن علامت عملگر قبل از متغیر، شکل پیشوندی عملگر خوانده می‌شود و استفاده کردن از عملگر بعد از متغیر، شکل پسوندی خوانده می‌شود. در اینجا مثال‌هایی آورده شده اند:

```
count++;    // postfix increment
++count;    // prefix increment
count--;    // postfix decrement
--count;    // prefix decrement
```

خواه از شکل پیشوندی عملگرهای ++ یا – – استفاده کنید یا شکل پسوندی آنها، برای متغیر تفاوت نمی‌کند که افزایش یابد یا کاهش یابد. برای مثال، اگر شما بنویسید *count++*، مقدار *count*، 1 واحد افزایش می‌یابد و اگر بنویسید *++count*، بازهم مقدار متغیر 1 واحد افزایش می‌یابد. با فهمیدن این موضوع، شما ممکن است نگران شوید که چرا در اینجا دو روش برای نوشتن چیزی یکسان وجود دارد. برای فهمیدن جواب، شما باید به یاد بیاورید که ++ و – – عملگر هستند و اینکه همه آن عملگرها برای ارزیابی یک عبارت که یک مقدار دارد به کار برده می‌شوند. مقدار برگشتی توسط *count++* برابر است با مقدار *count* قبل از اینکه افزایش اتفاق بیافتد، در حالی که مقدار برگشتی توسط *++count* برابر مقدار *count* بعد از اینکه افزایش اتفاق بیافتد. در اینجا یک مثال آورده شده است:

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
```

```
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

روش به خاطر سپاری اینکـــه کدام عمـلوند چه کـاری را انجـام می‌دهد این است که به ترتیب عناصـر (عملوند و عملگر) در یک عبـارت پیشـوندی یـا پسـوندی نگـاه کـرد. در عبـارت *x++*، متـــغیر *x* در اول رخ می‌دهد، پس مقدار آن به عنوان مقدار عبارت قبل از این کـه *x* افـزایش یابـد، بـه کـار بـرده مـی‌شـود. در عبارت *++x* ، عملگر در اول رخ می‌دهد، پس عملیات آن قبل از این که مقدار *x* بـه عنـوان نتیجـه ارزیـابی شود، انجام می‌شود.

این عملگرها، یعنـــی ++ و - -، بـه صـــورت خیلـی متـــداولی در دسـتورات *while* و *do* بـه کـار بـرده می‌شوند، که‌این دستورات در فصل ۵ عرضه می‌شوند. اگر در انزوا در حال استفاده کـردن از عملگرهـای ++ و - - هستید، به شکل پسوندی (*x++*) بچسبید و ثابت قدم باشید.

اعلان کردن متغیرهای محلی نوع دار به طور ضمنی

قبلاً در این فصل، شما دیدید که یک متغیر را با تعیین کردن یک نوع داده و یک شناسه اعـلان مـی‌کنیـد، مانند این :

```
int myInt;
```

هم چنین تذکر داده شد که قبل از این که سعی کنید که از یک متغیر استفاده کنیـد، شـما بایسـت یـک مقـدار بـه آن متغیـر تخصـیص دهیـد. شـما مـی‌توانیـد یـک متغیـر را در دسـتوری یکسـان اعـلان کـرده و مقداردهی کنید، مانند این:

```
int myInt = 99;
```

یا حتی مانند این، با فرض اینکه *myOtherInt* یک متغیر صحیح مقدار دهی شده است:

```
int myInt = myOtherInt * 99;
```

اکنون، به یاد بیاورید که مقداری که شما به یک متغیـر تخصـیص مـی‌دهیـد بایـد از همـان نـوع باشـد کـه متغیر است. برای مثال، شما تنها می‌توانید یک مقدار *int* را به یـک متغیـر *int* تخصـیص دهیـد. کامپـایلر C# می‌تواند به سـرعت نـوع یـک عبـارت بـه کـار رفتـه بـرای مقـداردهی یـک متغیـر را تشـخیص داهـد و چنانچه‌این نوع با نوع متغیر هم خوان نباشد، بـه شـما بگویـد. همچنـین شـما مـی‌توانیـد از کامپـایلر C# بخواهید نوع یک متغیر از یک عبارت را استنباط کند و این نوع را هنگام اعلان کـردن متغیـر بـا اسـتفاده از واژه کلیدی *var* به جای نوع به کار ببرد، مانند این:

```
var myVariable = 99;
```

```
var myOtherVariable = "Hello";
```

متغیرهای *myVariable* و *myOtherVariable* به عنوان متغیرهای نوع دار به طور ضمنی منتسب می‌شوند. واژه کلیدی *var* باعث می‌شود که کامپایلر نوع متغیرها را از نوع عبارت‌هایی که برای مقداردهی آنها به کار رفته اند استنباط کند. در این مثال‌ها، *myVariable* یک *int* است و *myOtherVariable* یک *string* است. این خیلی مهم است که بدانید این صرفاً یک تسهیلات برای اعلان کردن متغیرهاست و اینکه بعد از اینکه یک متغیر اعلان شده باشد، شما تنها می‌توانید مقادیری از نوع استنباط شده به آن تخصیص دهید --- شما نمی‌توانید مقادیر *float*، *double* یا *string* را بعداً در نقطه‌ای دیگر در برنامه خود به *myVariable* تخصیص دهید.

```
var yetAnotherVariable;      // Error - compiler cannot infer type
```

مهم اگر شما در گذشته با ویژوال بیسیک برنامه نویسی کرده‌اید، ممکن است با نوع *Variant* آشنا باشید، که شما می‌توانید از آن برای ذخیره هر نوع مقداری در یک متغیر استفاده کنید. در اینجا و اکنون تأکید می‌کنم که شما بایست هرچیزی را که هنگام برنامه نویسی با ویژوال بیسیک درباره متغیرهای *Variant* فراگرفته‌اید، فراموش کنید. با اینکه واژه‌های کلیدی مشابهی به نظر می‌رسند، *var* و *Variant* چیزهای کاملاً متفاوتی را معنا می‌دهند. هرگاه شما یک متغیر را در C# با استفاده از واژه کلیدی *var* اعلان کنید، نوع مقادیری که شما به متغیر تخصیص می‌دهید نمی‌تواند از طریق آن مقدار که برای مقداردهی متغیر به کار رفته، تغییر کند.

اگر شما یک وسواسی باشید، احتمالاً در این نقطه در حال ساییدن دندان‌های خود هستید و نگران هستید که چرا طراحان یک زبان شسته رفته‌ای مانند C# بایست به مشخصه‌ای مانند *var* اجازه دهند که وابرود. بعد از همه این حرف‌ها، *var* مثل یک بهانه برای نهایت تنبلی در مورد سهم برنامه‌نویسان به نظر می‌رسد و می‌تواند فهمیدن آن چه یک برنامه در حال انجامش است یا ردیابی اشکال‌ها را خیلی سخت کند (و حتی به سادگی اشکالات تازه‌ای را به درون کد شما وارد کند). گرچه، به من اعتماد کنید که *var* یک مقام خیلی قابل اطمینان در C# دارد، هم چنان که هنگام کار کردن در سرتاسر بسیاری از فصول آتی، خواهید دید. اما، عجالتاً، ما به استفاده کردن از متغیرهای صریحاً نوعدار شده می‌چسبیم به جز زمانی که نوعدار کردن ضمنی یک الزام باشد.

- اگر می‌خواهید که به فصل بعدی بروید
  ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۳ مراجعه کنید.

- اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید
  در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۳

نوشتن متدها و اعمال سطح دسترسی

بعد از کامل کردن این فصل شما قادر خواهید شد که:

- ☐ متدها را اعلان کرده و فراخوانی کنید.
- ☐ اطلاعات را به یک متد ارسال کنید.
- ☐ اطلاعات را از متد برگردانید.
- ☐ سطح دسترسی محلی و کلاسی را تعریف کنید.
- ☐ از دیباگر جامع برای بازدید داخل و بیرون متدها هنگامی که اجرا می‌شوند، استفاده کنید.

در فصل ۲، "کارکردن با متغیرها، عملگرها و عبارات"، شما یاد گرفتید که چگونه متغیرها را اعلان کنید، چگونه عبارات را با استفاده از عملگرها ایجاد کنید و چگونه حق تقدم و شرکت پذیری کنترل می‌کنند که عبارات حاوی عملگرهای متعدد به چه صورت ارزیابی می‌شوند. در این فصل، شما درباره متدها یاد خواهید گرفت. در ضمن یاد خواهید گرفت که چگونه از آرگومان‌ها و پارامترها برای ارسال اطلاعات به یک متد استفاده کنید و اطلاعات را از یک متد با استفاده از دستورات برگشتی، عودت دهید. سرانجام، شما خواهید دید که چگونه با استفاده از دیباگر جامع مایکروسافت ویژوال استودیو ۲۰۰۸ در داخل و بیرون متدها بازدید انجام دهید (ردیابی). در صورتی که متدها کاملاً مطابق انتظار شما کار نکنند و شما نیاز داشته باشید که اجرای متدهای خود را ردیابی کنید، این اطلاعات سودمند خواهند بود.

اعلان کردن متدها

یک متد دنباله مشخصی از دستورات است. اگر شما قبلاً با استفاده از زبان‌هایی مانند C یا ویژوال بیسیک برنامه‌نویسی کرده‌اید، می‌دانید که یک متد خیلی شبیه به یک تابع یا یک سابروتین است. یک متد یک اسم و یک بدنه دارد. اسم متد باید یک شناسه با معنی باشد که منظور کلی متد را نشان می‌دهد (مثلاً *CalculateIncomeTax*). بدنه متد دربردارنده دستورات واقعی است که هنگامی که متد فراخوانده می‌شود اجرا می‌شوند. علاوه بر این، برخی داده‌ها می‌توانند به متدها برای پردازش کردن داده شوند و متدها می‌توانند اطلاعاتی را برگردانند، که معمولاً نتیجه پردازش کردن است. متدها یک مکانیزم قدرتمند و بنیادی هستند.

مشخص کردن ترکیب نوشتاری/نحوی اعلان متد

ترکیب نوشتاری/نحوی یک متد مایکروسافت ویژوال #C همانند زیر است:

```
(returnType methodName (parameterList
{
    // method body statements go here
}
```

- returnType اسم یک نوع است و نوع اطلاعاتی که متد به عنوان یک نتیجه از پردازش خود برمی‌گرداند را تعیین می‌کند و می‌تواند هر نوعی باشد، مانند یک int یا string . اگر در حال یک متد هستید که مقداری را برگشت نمی‌دهد، شما باید از واژه کلیدی void به جای نوع برگشتی استفاده کنید.
- methodName اسم به کار رفته برای فراخوانی متد است. اسامی متد از همان قواعد شناسه پیروی می‌کنند که اسامی متغیر از آنها پیروی می‌کنند. برای مثال، addValues یک اسم متد معتبر است، در حالی که add$Values معتبر نیست. فعلاً، شما بایست از قرارداد حالت شتری برای اسامی متد استفاده کنید--- برای مثال، displayCustomer .
- parameterList اختیاری است و انواع و اسامی اطلاعاتی که شما می‌توانید به متد برای پردازش کردن ارسال کنید، توصیف می‌کند. شما پارامترها را انگار که در حال اعلان کردن متغیرها هستید، مابین پارانتزهای باز و بسته می‌نویسید با نام نوع که توسط نام پارامتر پی گرفته می‌شود. اگر متدی که در حال نوشتن آن هستید دو یا چند پارامتر دارد، شما باید آن پارامترها را با ویرگول از هم جدا کنید.
- دستورات بدنه متد سطرهایی از کد هستند که هنگامی که متد فراخوانده می‌شود اجرا می‌شوند. آنها در میان آکولادهای باز و بسته { } محصور می‌شوند.

مهم برنامه نویسان C، C++ و ویژوال بیسیک بایست توجه کنند که #C از متدهای سرتاسری پشتیبانی نمی‌کند. شما باید تمامی متدهای خود را درون یک کلاس بنویسید، یا این که کد شما کامپایل نخواهد شد.

در اینجا تعریف یک متد با نام *addValues* اورده شده است که یک نتیجه *int* برمی‌گرداند و دو پارامتر *int* با نام *leftHandSide* و *rightHandSide* دارد:

```
int addValues(int leftHandSide, int rightHandSide)
{
// ...
```

```
// method body statements go here
// ...
}
```

توجه شما باید به طور صریح نوع هر یک از پارامترها و نوع برگشتی یک متد را تعیین کنید. شما نمی‌توانید از واژه کلیدی *var* استفاده کنید.

در اینجا تعریف یک متد با نام *showResult* آورده شده است که مقداری را برنمی‌گرداند و یک پارامتر *int* یگانه با نام *answer* دارد:

```
void showResult(int answer)
{
// ...
}
```

توجه کنید که استفاده از واژه کلیدی *void* نشان می‌دهد که متد هیچ چیزی را برگشت نمی‌دهد.

مهم برنامه نویسان ویژوال بیسیک بایست توجه کنند که C# از واژه‌های کلیدی متفاوتی برای تمیز دادن مابین یک متد که یک مقدار را برمی‌گرداند (یک تابع) و یک متد که مقداری را برنمی‌گرداند (یک روال یا سابروتین) استفاده نمی‌کند. شما باید همواره یا یک نوع برگشتی را تعیین کنید یا void را .

نوشتن دستورات *return*

اگر شما می‌خواهید که یک متد اطلاعاتی را برگرداند (به عبارت دیگر، نوع برگشتی اش *void* نباشد)، باید یک دستور *return* در داخل متد بنویسید. شما این کار را با استفاده از واژه کلیدی *return* که با یک عبارت که مقدار برگشتی را محاسبه می‌کند و یک نقطه ویرگول پی گرفته می‌شود، انجام می‌دهید. نوع عبارت باید همان نوعی باشد که توسط متد تعیین شده است. به عبارت دیگر، اگر یک متد یک *int* را برمی‌گرداند، دستور *return* نیز باید یک *int* را برگرداند، در غیر این صورت، برنامه شما کامپایل نخواهد شد. در اینجا یک مثال آورده شده است:

```
int addValues(int leftHandSide, int rightHandSide)
{
   // ...
   return leftHandSide + rightHandSide;
}
```

دستور *return* معمولاً در انتهای متد شما مستقر می‌شود زیرا *return* باعث می‌شود که متد خاتمه پذیرد. هر دستوری که بعد از دستور *return* اتفاق بیافتد اجرا نمی‌شود (هرچند در صورتی که دستوراتی را بعد از دستور *return* جای دهید، کامپایلر شما را در مورد این مشکل آگاه می‌کند).

اگر شما نخواهید که متدتان اطلاعاتی را برگرداند (به عبارت دیگر، نوع برگشتی اش *void* باشد)، شما می‌توانید از یک ناپایداری دستور *return* برای واداشتن به یک خروج فوری از متد، استفاده کنید. شما واژه کلیدی *return* را که بلافاصله با یک نقطه ویرگول پی گرفته می‌شود، می‌نویسید. برای مثال:

```
void showResult(int answer)
{
   // display the answer
   ...
   return;
}
```

اگر متد شما هیچ چیزی را برگشت نمی‌دهد، شما می‌توانید از دستور *return* نیز صرف نظر کنید زیرا هنگامی که اجرا به آکولاد بسته در انتهای متد می‌رسد، متد به طور خودکار خاتمه می‌پذیرد. گرچه‌این تکنیک متداول و عمومی‌است، همواره به عنوان یک سبک خوب مورد توجه واقع نمی‌شود.
در فعالیت زیر، شما نسخه دیگری از برنامه MathsOperators از فصل ۲ را بررسی خواهید. این نسخه توسط استفاده دقیق از برخی متدهای کوچک بهینه سازی شده است.

بررسی تعاریف متد

ویژوال استودیو ۲۰۰۸ را اگر از قبل در حال اجرا نیست، شروع کنید.
پروژه *Methods* را در پوشه Microsoft Press\Visual CSharp Step by Step\Chapter 3\Methods واقع در پوشه Documents خود، باز کنید.
در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
ویژوال استودیو ۲۰۰۸ برنامه را ساخت و اجرا می‌کند.
مجدداً خودتان را با برنامه و اینکه چگونه کار می‌کند، آشنا کنید و پس از آن *Quit* را کلیک کنید.
کد را برای Window1.xaml.cs در پنجره ویرایشگر کد و متن نمایش دهید.
در پنجره ویرایشگر کد و متن، متد *addValues* را پیدا کنید. متد مانند این به نظر می‌رسد:

```
private int addValues(int leftHandSide, int rightHandSide)
{
    expression.Text = leftHandSide.ToString() + " + " + rightHandSide.ToString();
    return leftHandSide + rightHandSide;
}
```

متد *addValues* حاوی دو دستور است. دستور اول محاسبه انجام شده را در جعبه متن *expression* در روی فرم نمایش می‌دهد. مقادیر پارامترهای *leftHandSide* و *rightHandSide* به رشته تبدیل می‌شوند (با استفاده از متد *ToString* که در فصل ۲ با آن آشنا شدید) و به یکدیگر متصل می‌شوند با یک نمایش رشته‌ای از عملگر جمع (+) در وسط.

دومین دستور از عملگر + برای جمع کردن مقادیر *leftHandSide* و *rightHandSide* با یکدیگر استفاده می‌کند و نتیجه‌این عملیات را برمی‌گرداند. به یاد داشته باشید که اضافه کردن دو مقدار *int* به یکدیگر، مقدار *int* دیگری را ایجاد می‌کند، از این رو نو ع برگشتی متد *addValues* نیز *int* است.

اگر به متدهای *subtractValues*، *multiplyValues*، *divideValues* و *remainderValues* نگاهی بیاندازید، خواهید دید که آنها از یک الگوی مشابه پیروی می‌کنند.

در پنجره ویرایشگر کد و متن، متد *showResult* را پیدا کنید. متد *showResult* مانند زیر به نظر می‌رسد:

```
private void showResult(int answer)
{
    result.Text = answer.ToString();
}
```

این متد حاوی یک دستور است که یک نمایش رشته‌ای از پارامتر *answer* را در جعبه متن *result* نمایش می‌دهد.

توضیح هیچ طول حداقلی برای یک متد وجود ندارد. اگر یک متد برای پرهیز از تکرار کمک می‌کند و برنامه شما را برای درک کردن آسان تر می‌کند، متد قطع نظر از اینکه تا چه‌اندازه کوچک است، سودمند است.

در ضمن طول حداکثری برای یک متد وجود ندارد، اما معمولاً شما می‌خواهید که متد خود را به‌اندازه کافی کوچک نگه دارید که کار را به اتمام برساند. اگر متد شما بیشتر از یک صفحه طول داشته باشد، برای خوانایی بیشتر آن را به متدهای کوچک تر خرد کنید.

فراخوانی متدها

متدها زندگی می‌کنند تا فراخوانده شوند! شما یک متد را توسط اسمی برای طلبیدن آن به منظور انجام مأموریتش فرامی خوانید. اگر متد نیازمند اطلاعاتی باشد (همان طور که توسط پارامترهایش تعیین شده)، شما باید اطلاعات درخواست شده را فراهم کنید. اگر متد اطلاعاتی را برمی‌گرداند (همان طور که توسط نوع برگشتی اش تعیین شده)، شما بایست سازماندهی کنید تا این اطلاعات را به طریقی به چنگ آورید.

مشخص کردن ترکیب نوشتاری/نحوی فراخوان متد

ترکیب نوشتاری/نحوی یک متد #C همانند زیر است:

```
(result = methodName (argumentList
```

- *methodName* باید دقیقاً با اسم متدی که می‌خواهید فراخوانی کنید هم خوان باشد. به یاد داشته باشید که C# یک زبان حساس به حالت می‌باشد.
- عبارت *result =* اختیاری است. اگر تعیین شود، متغیر مشخص توسط result حاوی مقدار برگشتی توسط متد است. اگر متد void باشد (مقداری را برنگرداند)، شما باید از عبارت *result =* برای دستور صرف نظر کنید.
- *argumentList* اطلاعات اختیاری که متد آنها را قبول می‌کند، فراهم می‌کند. شما باید یک آرگومان برای هر پارامتر فراهم کنید و مقدار هر آرگومان باید با نوع پارامتر متناظرش سازگار باشد. اگر متدی که در حال فراخواندنش هستید، دو یا چند پارمتر داشته باشد، شما باید آرگومان‌ها را با ویرگول از هم جدا کنید.

مهم شما باید پارانتزها را در هر فراخوان متد جای دهید، حتی هنگامی که فراخوانی یک متد هیچ آرگومانی نداشته باشد.

برای روشن شدن این نکات، یک نگاه دوباره‌ای به متد *addValues* بیاندازید:

```
int addValues(int leftHandSide, int rightHandSide)
{
  // ...
}
```

متد *addValues* دو پارامتر *int* دارد، از این رو شما باید آن را با دو آرگومان *int* که با ویرگول از هم جدا شده‌اند، فراخوانی کنید:

```
addValues(39, 3);      // okay
```

هم چنین شما می‌توانید لیترال‌های 39 و 3 را با اسامی متغیرهای *int* جایگزین کنید. در این صورت مقادیر واقع در آن متغیرها به عنوان آرگومان‌های متد به او ارسال می‌شوند، مانند این:

```
int arg1 = 99;
int arg2 = 1;

addValues(arg1, arg2);
```

اگر شما سعی کنید که *addValues* را با برخی شیوه‌های دیگر فراخوانی کنید، به خاطر دلایل توصیف شده در مثال‌های زیر احتمالاً موفق نخواهید شد:

```
addValues;                // compile-time error, no parentheses
addValues();              // compile-time error, not enough arguments
addValues(39);            // compile-time error, not enough arguments
addValues("39", "3");     // compile-time error, wrong types
```

متد *addValues* یک مقدار *int* برمی‌گرداند. این مقدار *int* می‌تواند در هرجایی که یک مقدار *int* می‌تواند به کار برده شود، استفاد ه شود. به‌این مثال‌ها توجه کنید:

```
int result = addValues(39, 3);     // on right-hand side of an assignment
showResult(addValues(39, 3));    // as argument to another method call
```

فعالیت زیر مشاهده برنامه Methods را ادامه می‌دهد. در این زمان شما می‌توانید برخی از فراخوانی‌های متد را بررسی کنید.

بررسی فراخوان‌های متد

به پروژه Methods برگردید. (اگر شما در حال ادامه فعالیت قبلی هستید، این پروژه قبلاً در ویژوال استودیو ۲۰۰۸ باز شده است. اگر این گونه نیست، پروژه را از پوشه Microsoft Press\Visual CSharp Step by Step\Chapter 3\Methods واقع در پوشه Documents خود باز کنید.)
کد را برای Window1.xaml.cs در ویرایشگر کد و متن نمایش دهید.
متد *calculateClick* را پیدا کنید و به دو دستور ابتدایی این متد بعد از دستور *try* و آکولاد باز نگاه کنید. (هدف از دستورات *try* را در فصل ٦، "مدیریت کردن خطاها و اخطارها"، پوشش خواهیم داد. دستورات همانند زیر هستند:

```
int leftHandSide = System.Int32.Parse(lhsOperand.Text);
int rightHandSide = System.Int32.Parse(rhsOperand.Text);
```

این دو دستور دو متغیر *int* با نامهای *leftHandSide* و *rightHandSide* اعلان می‌کنند. گرچه، بخش جذاب، شیوه‌ای است که در آن متغیرها مقداردهی می‌شوند. در هر دو حالت، متد *Parse* از کلاس *System.Int32* فراخوانده می‌شود (*System* یک فضای اسمی است و *Int32* اسم کلاس در این فضای اسمی است). شما این متد را قبلاً دیده‌اید؛ این متد یک پارامتر واحد می‌گیرد و آن را به یک مقدار *int* تبدیل می‌کند. این دو سطر کد آن چه را که کاربر در کنترل‌های جعبه متن *lhsOperand* و *rhsOperand* در روی فرم تایپ کرده است، می‌گیرد و آنها را به مقادیر *int* تبدیل می‌کند.
به دستور چهارم در متد *calculateClick* نگاه کنید (بعد از دستور *if* و آکولاد باز دیگر):

```
calculatedValue = addValues(leftHandSide, rightHandSide);
```

این دستور متد *addValues* را فرامی خواند، در حالی که مقادیر متغیرهای *leftHandSide* و *rightHandSide* را به عنوان آرگومان‌های خود ارسال می‌کند. مقدار برگشتی توسط متد *addValues* در متغیر *calculatedValue* ذخیره می‌شود.
به دستور بعدی نگاه کنید:

```
showResult(calculatedValue);
```

این دستور متد *showResult* را فرامی خواند، در حالی که مقدار واقع در متغیر *calculatedValue* را به عنوان آرگومان خودش ارسال می‌کند. متد *showResult* مقداری را برنمی‌گرداند.
در پنجره ویرایشگر کد و متن، متد *showResult* را که قبلاً به آن نگاه کرده‌اید، پیدا کنید. تنها دستور این متد این است:

```
result.Text = answer.ToString();
```

توجه کنید که فراخوان متد *ToString* با اینکه هیچ آرگومانی وجود ندارد، از پارانتزها استفاده می‌کند.

توضیح شما می‌توانید متدهای متعلق به اشیای دیگر را با پیشوند دار کردن نام متد با نام شیء فراخوانی کنید. در مثال قبلی، عبارت *answer.ToString()* متدی با نام *ToString* متعلق به شیئی با نام *answer* را فراخوانی می‌کند.

اعمال سطح دسترسی (دامنه: Scope)

در برخی از مثال‌ها، شما می‌توانید ببینید که شما می‌توانید متغیرهایی را درون یک متد ایجاد کنید. این متغیرها در نقطه‌ای به موجودیت در می‌آیند که آنها تعریف شده‌اند و پس از آن دستورات متعاقب در همان متد می‌توانند از این متغیرها استفاده کنند؛ یک متغیر تنها بعد از آنکه‌ایجاد شده باشد می‌تواند به کار برده شود. هرگاه متد خاتمه می‌یابد، این متغیرها ناپدید می‌شوند.
اگر یک متغیر بتواند در یک موقعیت به خصوص در یک برنامه به کاربرده شود، گفته می‌شود که متغیر در میدان دید (سطح دسترسی: *scope*) آن موقعیت قرار دارد. به عبارت دیگر، سطح دسترسی یک متغیر در حقیقت ناحیه‌ای از برنامه است که در آن ناحیه آن متغیر قابل استفاده است. سطح دسترسی به متدها به همان اندازه متغیرها اعمال می‌شود. سطح دسترسی یک شناسه (برای یک متد یا یک متغیر) به موقعیت اعلانی که شناسه را در برنامه معرفی می‌کند پیوند داده می‌شود، هم چنان که اکنون یاد خواهید گرفت.

تعریف سطح دسترسی محلی

آکولادهای باز و بسته که بدنه یک متد را شکل می‌دهند، یک سطح دسترسی (میدان دید) را تعریف می‌کنند. هر متغیر را که شما در میان بدنه یک متد اعلان می‌کنید در سطح دسترسی آن متد قرار می‌گیرد؛ هنگامی که متد خاتمه می‌یابد آنها ناپدید می‌شوند و تنها می‌توانند توسط کد در حال اجرا در آن متد قابل دسترسی باشند. این متغیرها، متغیرهای محلی (*local variables*) خوانده می‌شوند زیرا آنها محدود به متد هستند، جایی که در آن اعلان می‌شوند؛ آنها در میدان دید هیچ متد دیگری قرار ندارند. این آرایش به معنای این است که شما نمی‌توانید از متغیرهای محلی برای اشتراک اطلاعات مابین متدها استفاده کنید. به‌این مثال توجه کنید:

```
class Example
{
void firstMethod()
{
int myVar;
...
}
void anotherMethod()
{
myVar = 42;    // error – variable not in scope
...
}
}
```

این کد برای کامپایل شدن نافرجام خواهد ماند زیرا *anotherMethod* در حال تلاش برای استفاده از متغیر *myVar* است که در سطح دسترسی اش نیست. متغیر *myVar* تنها برای دستورات واقع در *firstMethod* و آنهایی که بعد از سطری از کد که *myVar* را اعلان می‌کند، اتفاق می‌افتند در دسترس است.

تعریف کردن سطح دسترسی کلاسی

آکولادهای باز و بسته که بدنه یک کلا را تشکیل می‌دهند هم یک سطح دسترسی (میدان دید) شکل می‌دهند. هر متغیری که در داخل بدنه یک کلاس (اما نه در داخل یک متد) اعلان می‌کنید، در میدان دید و سطح دسترسی آن کلاس قرار دارند. نام مخصوص C# برای متغیرهای تعریف شده توسط یک کلاس یک فیلد (*field*) است. در تباین با متغیرهای محلی، شما می‌توانید از فیلدها برای اشتراک اطلاعات مابین متدها استفاده کنید. در اینجا یک مثال آورده شده است:

```
class Example
{
void firstMethod()
{
```

```
myField = 42; // ok
...
}
void anotherMethod()
{
myField++; // ok
...
}
int myField = 0;
}
```

متغیر *myField* در کلاس تعریف شده است اما خارج از متدهای *firstMethod* و *anotherMethod* . از این رو *myField* سطح دسترسی کلاس را دارد و برای استفاده توسط همه متدها در این کلاس در دسترس است.

یک نکته برای توجه نشان دادن در این مثال وجود دارد. در یک متد، شما بایست یک متغیر را قبل از اینکه بتوانید از آن استفاده کنید، اعلان کنید. فیلدها اندکی متفاوت هستند. یک متد می‌تواند از یک فیلد قبل از دستوری که فیلد را اعلان می‌کند، استفاده کند--- کامپایلر جزییات را برای شما دسته بندی می‌کند!

سربارگذاری متدها

اگر دو شناسه اسم یکسانی داشته باشند و در میدان دید یکسان اعلان شده باشند، گفته می‌شود که آنها سربارگذاری شده (*overloaded*) باشند. اغلب یک شناسه سربارگذاری شده یک اشکال است که به عنوان یک خطای زمان کامپایل به تله می‌افتد. برای مثال، اگر دو متغیر محلی با نام یکسان در متد یکسان اعلان کنید، شما یک خطای زمان کامپایل می‌گیرید. به طور مشابه، اگر شما دو فیلد با نام یکسان در کلاس یکسان یا دو متد یکسان در کلاس یکسان اعلان کنید، باز هم یک خطای زمان کامپایل دریافت می‌کنید. معلوم است که‌این حقیقت که هر چیزی که تا این اندازه نتیجه شده باشد که یک خطای زمان کامپایل می‌شود، ممکن است به زحمت برای تذکر دادن با ارزش به نظر برسد. گرچه روشی وجود دارد که می‌توانید یک شناسه را سربارگذاری کنید و این روش هم سودمند و هم با اهمیت است.

به متد *WriteLine* از کلاس *Console* توجه کنید. شما قبلاً از این متد برای خروجی دهی یک رشته به صفحه نمایش استفاده کردید. گرچه، هنگامی که شما *WriteLine* را هنگام نوشتن کد C# در پنجره ویرایشگر کد و متن تایپ می‌کنید، متوجه خواهید شد که IntelliSense به شما ۱۹ انتخاب متفاوت می‌دهد! هر نسخه از متد *WriteLine* مجموعه‌ای متفاوت از پارامترها را می‌گیرد؛ یک نسخه هیچ پارامتری نمی‌گیرد و به سادگی یک سطر خالی را به خروجی می‌دهد، نسخه دیگری یک پارامتر *bool* می‌گیرد و یک رشته بیانگر مقدار خودش (*true* یا *false*) را به خروجی می‌دهد، با این حال یک پیاده سازی دیگر یک پارامتر *decimal* می‌گیرد و آن را به عنوان رشته به خروجی می‌دهد و الی آخر. در زمان

کامپایلر، کامپایلر به انواع پارامترهایی که شما ارسال کرده‌اید نگاه می‌کند و پس از آن نسخه‌ای از متد را که یک مجموعه پارامتر همخوان دارد فراخوانی می‌کند. در اینجا یک مثال آورده شده است:

```
static void Main()
{
   Console.WriteLine("The answer is ");
   Console.WriteLine(42);
}
```

هنگامی که شما نیاز به انجام دادن عملیات یکسان روی انواع داده متفاوتی دارید، سربارگذاری اصولاً سودمند می‌باشد. هنگامی که پیاده سازی‌های متفاوتی از متد مجموعه‌های مختلفی از پارامترها را دارند، شما می‌توانید یک متد را سربارگذاری کنید؛ یعنی، هنگامی که متدها نام یکسان اما تعداد متفاوتی از پارامترها را دارند یا هنگامی که انواع پارامترها تفاوت می‌کند. هنگامی که یک متد را فراخوان می‌کنید، این قابلیت جایز است به طوری که شما می‌توانید لیستی از آرگومان‌ها که با ویرگول از هم جدا شده‌اند، فراهم کنید و تعداد و نوع آرگومان‌ها توسط کامپایلر برای انتخاب یکی از متدهای سربارگذاری شده به کار برده می‌شوند. اما، توجه کنید که با این که می‌توانید پارامترهای یک متد را سربارگذاری کنید، شما نمی‌توانید نوع برگشتی یک متد را سربار گذاری کنید. به عبارت دیگر، شما نمی‌توانید دو متد را با نام یکسان اعلان کنید که تنها نوع برگشتی آنها فرق می‌کند. (کامپایلر باهوش است، اما نه آن باهوشی که شما می‌شناسید.)

نوشتن متدها

در فعالیت‌های زیر، شما یک متد ایجاد خواهید کرد که محاسبه می‌کند که تا چه‌اندازه یک مشاور برای تعدادی از روزهای مشاوره معلوم در یک نرخ روزانه ثابت هزینه خواهد کرد. شما با طراحی منطق لازم برای برنامه شروع خواهید کرد و پس از آن از Generate Method Stub Wizard استفاده خواهید کرد تا به شما در نوشتن متدهایی که در این منطق به کار برده می‌شوند، کمک کند. بعد از این، شما این متدها را در یک دیباگر اجرا خواهید کرد تا درون وبرون فراخوان‌های متد را همان طور که اجرا می‌شوند، ردیابی کنید.

طراحی منطق لازم برای برنامه

با استفاده از ویژوال استودیو ۲۰۰۸، پروژه DailyRate واقع در پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 3\DailyRate در پوشه Documents خود را باز کنید.

در مرورگر محلول (*Solution Explorer*)، فایل *Program.cs* را برای نمایش کد لازم برای برنامه در پنجره ویرایشگر کد و متن، دابل کلیک کنید.

دستورات زیر را به بدنه متد *run*، مابین آکولادهای باز و بسته، اضافه کنید:

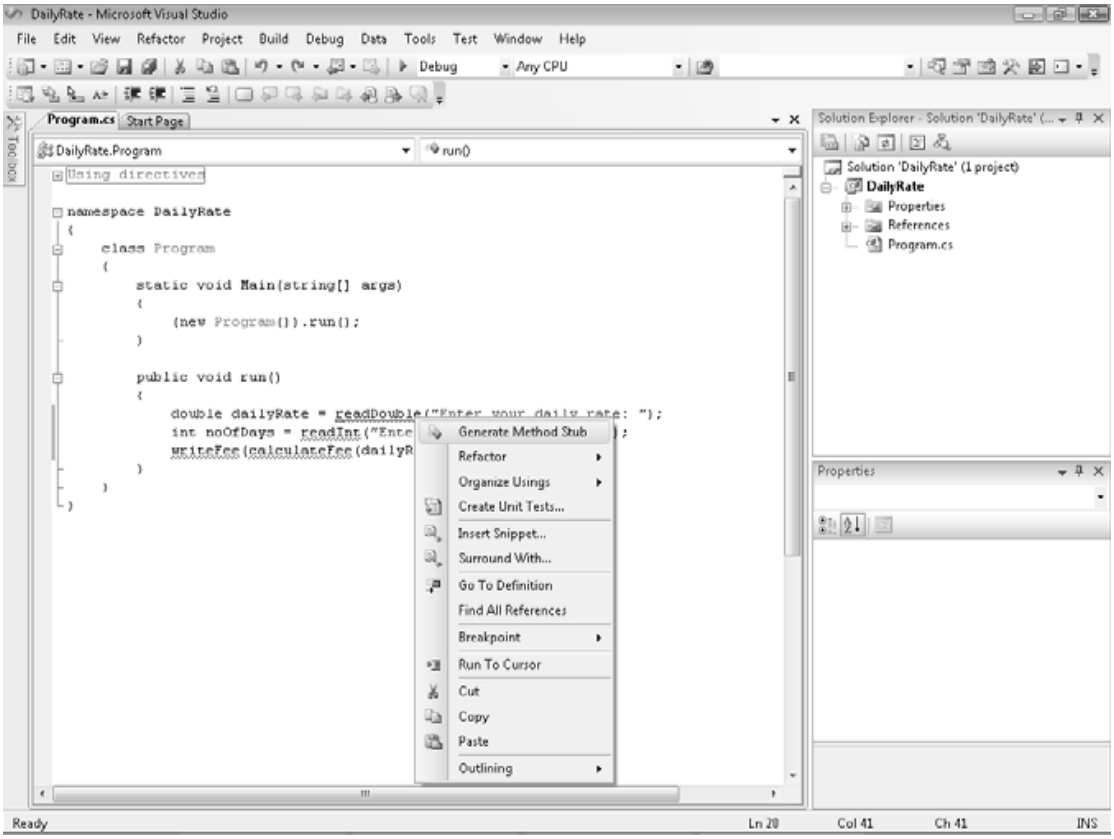
```
double dailyRate = readDouble("Enter your daily rate: ");
int noOfDays = readInt("Enter the number of days: ");
writeFee(calculateFee(dailyRate, noOfDays));
```

هنگامی که برنامه شروع می‌شود، متد *run* توسط متد *Main* فراخوانی می‌شود.
بلوکی از کد که شما اندکی قبل به متد *run* اضافه کرده‌اید، متد *readDouble* را (که شما به زودی خواهید نوشت) به منظور تقاضا از کاربر برای نرخ روزانه برای مشاور، فرامی خواند. دستور بعدی متد *readInt* را (که شما به زودی خواهید نوشت) برای گرفتن تعداد روزها، فرامی خواند. بالاخره، متد *writeFee* (نوشته خواهد شد) برای نمایش نتایج روی صفحه فراخوانده می‌شود. توجه کنید که مقدار ارسال شده به *writeFee* مقدار برگشتی توسط متد *calculateFee* (آخرین متد که خواهید نوشت) است، که‌این متد نرخ روزانه و تعداد روزها را می‌گیرد و دست مزد قابل پرداخت را محاسبه می‌کند.

توجه شما هنوز متدهای *readDouble*، *readInt*، *writeFee* یا *calculateFee* را ننوشته‌اید، از این رو هنگامی که شما این کد را تایپ می‌کنید، IntelliSense این متدها را نمایش نمی‌دهد. هنوز سعی نکنید که برنامه را بنا کنید، زیر که نافرجام خواهد ماند.

نوشتن متدها با استفاده از جادوگر Generate Method Stub

در پنجره ویرایشگر کد و متن، فراخوان متد *readDouble* را در متد *run* راست کلیک کنید.
یک منوی میان بر ظاهر می‌شود که حاوی فرمان‌های مفیدی برای تولید و ویرایش کردن کد است، همان طور که در اینجا نشان داده شده است:



در منوی میان بر، *Generate Method Stub* را کلیک کنید.

جادوگر Generate Method Stub فراخوانی به متد *readDouble* را بررسی می‌کند، نوع پارمترها و نوع مقدار برگشتی آن را معلوم می‌کند و یک متد با یک پیاده سازی پیش فرض تولید می‌کند، مانند این:

```
private double readDouble(string p)
{
throw new NotImplementedException();
}
```

متد جدید با تصریح کننده *private* (خصوصی) ایجاد می‌شود. بدنه متد الساعه به طور جاری یک *NotImplementedException* صادر می‌کند. (استثناها در فصل ۶ بحث می‌شوند.) در گام بعدی شما بدنه را با کد شخصی خود جایگزین خواهید کرد.

دستور *throw new NotImplementedException();* را از متد *readDouble* پاک کنید و آن را با سطرهای کد زیر جایگزین کند:

```
Console.Write(p);
string line = Console.ReadLine();
return double.Parse(line);
```

این بلوک کد رشته واقع در متغیر *p* را به صفحه نمایش خروجی می‌دهد. این متغیر پارامتر رشته‌ای است که هنگامی که متد فراخوانی می‌شود، ارسال می‌شود و حاوی یک پیغام برانگیزاننده برای کاربر است تا نرخ روزانه را تایپ و وارد کند.

توجه متد *Console.Write* خیلی شبیه به دستور *Console.WriteLine* است که شما آن را در فعالیت‌های قبلی به کار برده‌اید، به استثنای اینکه‌این متد یک کاراکتر سطرجدید را بعد از پیغام به خروجی نمی‌دهد.

کاربر یک مقدار را تایپ می‌کند، که با استفاده از متد *ReadLine* به درون یک *string* خوانده می‌شود و با استفاده از متد *double.Parse* به یک *double* تبدیل می‌شود. نتیجه به عنوان مقدار برگشتی فراخوان متد به عقب ارسال می‌شود.

توجه متد *ReadLine* متد هم نشین برای متد *WriteLine* است؛ متد *ReadLine* ورودی کاربر را از صفحه کلید می‌خواند، در حالی که هنگامی که کاربر کلید Enter را فشار می‌دهد، پایان می‌پذیرد. متن تایپ شده توسط کاربر به عنوان مقدار برگشتی به عقب ارسال می‌شود.

فراخوانی به متد *readInt* واقع در متد *run* را راست کلیک کنید و پس از آن *Generate Method Stub* را برای تولید متد *readInt* کلیک کنید.
متد *readInt* تولید می‌شود، مانند این:

```
private int readInt(string p)
{
throw new NotImplementedException();
}
```

دستور *throw new NotImplementedException();* را در بدنه متد *readInt* با کد زیر جایگزین کنید:

```
Console.Write(p);
string line = Console.ReadLine();
return int.Parse(line);
```

این بلوک کد مشابه با کد متد *readDouble* است. تنها تفاوت این است که متد یک مقدار *int* برمی‌گرداند، از این رو *string* تایپ شده توسط کاربر، با استفاده از متد *int.Parse* به یک عدد تبدیل می‌شود.

فراخوانی به متد *calculateFee* واقع در متد *run* را راست کلیک کنید و پس از آن *Generate Method Stub* را برای تولید متد *calculateFee* کلیک کنید.
متد *calculateFee* تولید می‌شود، مانند این:

```
private object calculateFee(double dailyRate, int noOfDays)
{
throw new NotImplementedException();
}
```

توجه کنید که جادوگر Generate Method Stub از اسامی آرگومان‌های ارسال شده برای تولید اسامی لازم برای پارامترها استفاده می‌کند. (البته شما می‌توانید اسامی پارامترها را در صورتی که مناسب نباشند، تغییر دهید.) چیزی که خیلی فریبنده است نوع برگشت داده شده توسط متد است، که *object* (شیء) است. جادوگر Generate Method Stub عاجز از این است که از متنی که در آن فراخوانی انجام می‌شود، تشخیص دهد که دقیقاً کدام نوع از مقدار بایست توسط متد برگشت داده شود. نوع *object* تنها به معنای یک "چیز" است و شما هنگامی که کد لازم را به متد اضافه می‌کنید، بایست آن را به نوعی که نیاز دارید تغییر دهید.

تعریف متد *calculateFee* را تغییر دهید به طوری که یک *double* را برگرداند، همان طور که با فونت درشت در اینجا نشان داده شده است:

```
private double calculateFee(double dailyRate, int noOfDays)
{
throw new NotImplementedException();
}
```

بدنه متد *calculateFee* را با دستور زیر جایگزین کنید، که‌این دستور دست مزد قابل پرداخت را توسط ضرب کردن دو پارامتر در یکدیگر محاسبه می‌کند و آن را برمی‌گرداند:

```
return dailyRate * noOfDays;
```

فراخوانی به متد *writeFee* واقع در متد *run* را راست کلیک کنید و پس از آن *Generate Method Stub* را برای تولید متد *writeFee* کلیک کنید.

توجه کنید که جادوگر Generate Method Stub از تعریف متد *calculateFee* برای سر در آوردن از این که پارامترش بایست یک *double* باشد، استفاده می‌کند. هم چنین، فراخوان متد از یک مقدار برگشتی استفاده نمی‌کند، از این رو نوع متد *void* است:

```
private void writeFee(double p)
{
...
}
```

توضیح اگر به قدر کافی با ترکیب نوشتاری/نحوی (syntax) احساس راحتی می‌کنید، شما در ضمن می‌توانید متدها را به طور مستقیم با تایپ کردن آنها در پنجره ویرایشگر کد و متن بنویسید. شما همیشه نبایست از گزینه منوی *Generate Method Stub* استفاده کنید.

دستور زیر را در داخل متد *writeFee* تایپ کنید:

```
Console.WriteLine("The consultant's fee is: {0}", p * 1.1);
```

توجه این نسخه از متد *WriteLine* استفاده از یک فرمت رشته را نمایش می‌دهد. متن *{0}* در رشته به کار رفته به عنوان اولین آرگومان برای متد *WriteLine* یک نگه دارنده مکان است که با مقدار عبارت پیرو رشته (*p * 1.1*) بعد از این که در زمان اجرا ارزیابی شد، جایگزین می‌شود. استفاده از این تکنیک ارجح بر گزینه‌های جایگزین، مانند تبدیل مقدار عبارت *p * 1.1* به یک رشته و استفاده از عملگر + برای چسباندن آن به پیغام است.

در منوی *Build*، گزینه *Build Solution* را کلیک کنید.

عاملیت مجدد کد

یک مشخصه بسیار سودمند ویژوال استودیو ۲۰۰۸ توانایی برای عاملیت مجدد کد است. برخی اوقات، در خواهید یافت که شما خوتان کد یکسانی (یا مشابهی) را در بیشتر از یک جای برنامه بنویسید. هرگاه‌این امر اتفاق بیافتد، بلوک کدی را که‌اندکی قبل تایپ کرده‌اید، پر رنگ کنید، و روی منوی *Refactor*، گزینه *Extract Method* را کلیک کنید. جعبه گفتگوی *Extract Method* ظاهر می‌شود، در حالی که اسم یک متد جدید را که حاوی این کد است برای ایجاد کردن از شما می‌خواهد. اسم را تایپ کنید و OK را کلیک کنید. متد جدید در حالی که حاوی کد شماست ایجاد می‌شود و کدی که شما تایپ کرده‌اید با یک فراخوان به‌این متد جایگزین می‌شود . ضمناً *Extract Method* به قدری کافی باهوش است که سر در آورد که‌ایا متد بایست هر گونه پارامتری را بگیرد و یک مقدار را برگرداند یا نه.

آزمایش برنامه

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

ویژوال استودیو ۲۰۰۸ برنامه را بنا می‌کند و سپس آن را اجرا می‌کند. یک پنجره کنسول ظاهر می‌شود.

در اعلان فوری *Enter your daily rate*، مقدار **525** را تایپ کنید و سپس Enter را فشار دهید.

در اعلان فوری *Enter the number of days*، مقدار **17** را تایپ کنید.

برنامه پیغام زیر را به پنجره کنسول می‌نویسد:

```
The consultant's fee is: 9817.5
```

کلید Enter را برای بستن برنامه و بازگشت به محیط برنامه‌نویسی ویژوال استودیو ۲۰۰۸ فشار دهید.

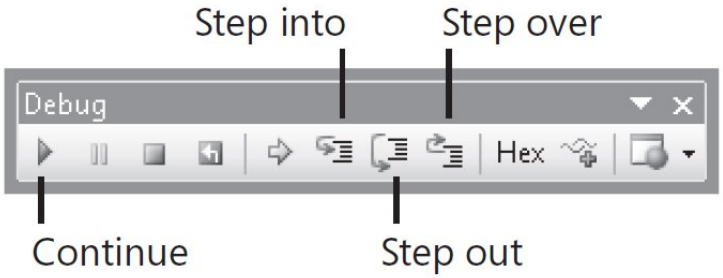
در فعالیت نهایی، شما از دیباگر ویژوال استودیو ۲۰۰۸ برای اجرای برنامه خود در حرکت آهسته استفاده خواهید کرد. خواهید دید که چه موقع هر یک متدها فراخوانده می‌شود (این عمل به عنوان *stepping into the method* منتسب می‌شود) و خواهید دید که چگونه دستور *return*، کنترل را به فراخواننده منتقل می‌کند (که به عنوان *stepping out of the method* نیز شناخته می‌شود). در حالی که در حال ردیابی به داخل و به بیرون از متدها هستید، شما از ابزار روی میله ابزار *Debug* استفاده خواهید کرد. گرچه، هنگامی که یک برنامه در حال اجرا به سبک Debug است، همان فرمان‌ها نیز روی منوی *Debug* در دسترس هستند.

قدم زدن در میان متدها با استفاده از دیباگر ویژوال استودیو ۲۰۰۸

در پنجره ویرایشگر کد و متن، متد *run* را پیدا کنید.
ماوس را به اولین دستور در متد *run* حرکت دهید:

```
double dailyRate = readDouble("Enter your daily rate: ");
```

در هر جایی روی این سطر راست کلیک کنید و در منوی میان بر، *Run To Cursor* را کلیک کنید.
برنامه شروع شده و اجرا می‌شود تا اینکه به اولین دستور در متد *run* برسد و سپس مکث می‌کند.
یک پیکان زرد رنگ در حاشیه سمت چپی پنجره ویرایشگر کد و متن، دستور جاری را نشان می‌دهد، که در ضمن این دستور با یک پس زمینه زرد رنگ پررنگ شده است.
در منوی *View*، ابتدا *Toolbars* را کلیک کنید و سپس مطمئن شوید که میله ابزار *Debug* انتخاب شده است.
اگر میله ابزار *Debug* از قبل قابل مشاهده نباشد، باز می‌شود. این میله ابزار ممکن است در حالی ظاهر شود که به میله‌های ابزار دیگر مهار شده باشد. اگر نمی‌توانید میله ابزار را ببینید، سعی کنید که با استفاده از فرمان *Toolbars* در منوی *View* آن را مخفی کنید و توجه کنید که کدام دکمه‌ها ناپدید می‌شوند. سپس میله ابزار را دوباره نمایش دهید. میله ابزار مانند این به نظر می‌رسد (میله ابزار اندکی مابین ویژوال استودیو ۲۰۰۸ و Microsoft Visual C# 2008 Express Edition فرق می‌کند):



	توضیح برای ظاهر کردن میله ابزار *Debug* در پنجره شخصی خودش، از دستگیره واقع در انتهای سمت چپ میله ابزار برای کشیدن میله ابزار در روی پنجره ویرایشگر کد و متن، استفاده کنید.

روی میله ابزار *Debug*، دکمه *Step Into* را کلیک کنید. (این دکمه، ششمین دکمه از سمت چپ است.)

این عمل باعث می‌شود که دیباگر در داخل متدی که فراخوانده شده قدم بردارد. مکان نمای زرد رنگ به آکولاد باز در ابتدای متد *readDouble* پرش می‌کند.

Step Into را دوباره کلیک کنید. مکان نما به طرف اولین دستور جلو می‌رود:

```
Console.Write(p);
```

	توضیح هم چنین شما می‌توانید به جای کلیک کردن مکرر *Step Into* در روی میله ابزار *Debug*، F11 را فشار دهید.

روی میله ابزار *Debug*، دکمه *Step Over* را کلیک کنید. (این دکمه هفتمین دکمه از سمت چپ است.)

این عمل باعث می‌شود که متد دستور بعدی را بدون اشکال زدایی آن اجرا کند (با گام زدن به داخل آن). مکان نمای زرد رنگ به دومین دستور متد جابه جا می‌شود و برنامه قبل از برگشتن به ویژوال استودیو ۲۰۰۸ اعلان فوری *Enter your daily rate* را در یک پنجره کنسول نمایش می‌دهد. (پنجره کنسول ممکن است در پشت ویژوال استودیو پنهان شده باشد.)

	توضیح هم چنین شما می‌توانید به جای کلیک کردن *Step Over* در روی میله ابزار *Debug*، F10 را فشار دهید.

روی میله ابزار *Debug*، دکمه *Step Over* را کلیک کنید.

در این لحظه، مکان نمای زرد رنگ ناپدید می‌شود و پنجره کنسول فوکوس را می‌گیرد زیرا برنامه در حال اجرای متد *Console.ReadLine* است و منتظر این است که شما چیزی را تایپ کنید.

در پنجره کنسول **525** را تایپ کنید و Enter را فشار دهید.

کنترل به ویژوال استودیو ۲۰۰۸ برمی‌گردد. مکان نمای زرد رنگ روی سومین سطر متد ظاهر می‌شود.

بدون کلیک کردن، ماوس را روی ارجـاع به متـغیر *line* یا روی دوم یا سـطر سوم متـد، جابـه جـا کنیـد (مهم نیست که کدام باشد).

یک ScreenTip (صفحه کوچکی حاوی کمکی آنی) ظـاهر می‌شود، در حـالی کـه مقـدار جـاری متغیر *line* ("525") را نشان می‌دهـد. شـما مـی‌توانیـد از ایـن ویژگـی هنگـام قـدم زدن در میـان متـدها، بـرای مطمئن شدن از اینکه یک متغیر به یک مقدار مورد انتظار تنظیم شده باشد، استفاده کنید.

روی میلـه ابـزار *Debug*، دکمـه *Step Out* را کلیـک کنیـد. (ایـن دکمـه هشـتمین دکمـه از سـمت چـپ است.)

این عمل باعث می‌شود که متد جاری اجرا را بدون وقفه تا انتهـای خـود ادامـه دهـد. متـد *readDouble* خاتمه می‌پذیرد و مکان نمای زرد رنگ رو به عقب در اولین دستور متد *run* مستقر می‌شود.

تـوضیـح شما هم چنین می‌توانیـد بـه جـای کلیـک کـردن *Step Out* در روی میلـه ابـزار *Debug* ، Shift+F11 را فشار دهید.

روی میله ابزار *Debug*، دکمه *Step Into* را کلیک کنید.

مکان نمای زرد رنگ به دومین دستور در متد *run* جا به جا می‌شود:

```
int noOfDays = readInt("Enter the number of days: ");
```

روی میله ابزار *Debug*، دکمه *Step Over* را کلیک کنید.

در این لحظه، شما انتخاب کرده‌اید که متد را بدون گام زدن در میان آن اجرا کنیـد. پنجـره کنسـول دوبـاره ظاهر می‌شود، در حالی که منتظر است که شما تعداد روزها را وارد کنید.

در پنجره کنسول، **17** را تایپ کنید و سپس Enter را فشار دهید.

کنترل به ویژوال استودیو ۲۰۰۸ برمی‌گردد. مکان نمای زرد رنگ بـه سـومین دسـتور متـد *run* جـا بـه جـا می‌شود.

```
writeFee(calculateFee(dailyRate, noOfDays));
```

روی میله ابزار *Debug*، دکمه *Step Into* را کلیک کنید.

مکان نمای زرد رنگ به آکـولاد باز در شـروع متـد *calculateFee* پـرش مـی‌کنـد. ایـن متـد اول فراخوانـده مـی‌شـود، قبـل از *writeFee*، زیـرا مقـدار برگشـت داده شـده توسـط ایـن متـد بـه عنـوان پـارامتر بـرای *writeFee* به کار برده می‌شود.

روی میله ابزار *Debug*، دکمه *Step Out* را کلیک کنید.

مکان نمای زرد رنگ به سومین دستور متد *run* پرش به عقب می‌کند.

روی میله ابزار *Debug*، دکمه *Step Into* را کلیک کنید.

در این لحظه، مکان نمای زرد رنگ به آکولاد باز در شروع متد *writeFee* پرش می‌کند.

ماوس را روی متغیر *p* واقع در تعریف متد قرار دهید.

مقدار *p*، 8925.0، در یک ScreenTip (صفحه‌ای کوچک حاوی یک کمک آنی) نمایش داده می‌شود.

روی میله ابزار *Debug*، دکمه *Step Out* را کلیک کنید.

پیغام *The consultant's fee is: 9817.5* در پنجره کنسول نمایش می‌یابد. (اگر پنجره کنسول در پشت سر ویژوال استودیو ۲۰۰۸ پنهان شده باشد، شاید نیاز داشته باشید که آن را برای نمایش به پیش زمینه برسانید.) مکان نمای زرد رنگ به سومین دستور در متد *run* برمی‌گردد.
روی میله ابزار *Debug*، دکمه *Continue* را کلیک کنید (که اولین دکمه روی میله ابزار است) تا باعث شوید که برنامه اجرا را بدون متوقف شدن در هر دستوری، ادامه دهد.

توضیح هم چنین شما می‌توانید برای ادامه اجرا در دیباگر، F5 را فشار دهید.

برنامه کامل می‌شود و اجرا کردن خاتمه می‌دهد.
تبریک می‌گویم! شما با موفقیت متدها را نوشته و فراخواندید و از دیباگر ویژوال استودیو ۲۰۰۸ برای قدم زدن به داخل و خارج متدها، هنگامی که اجرا می‌شوند، استفاده کردید.

- اگر می‌خواهید که به فصل بعدی بروید
  ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۴ مراجعه کنید.

- اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید
  در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۴

استفاده از دستورات داوری

بعد از کامل کردن این فصل، شما قادر خواهید بود که:

- متغیرهای بولی (Boolean) را اعلان کنید.
- از عملگرهای بولی برای ایجاد عباراتی که برآمد آنها یا صحیح (True) است یا غلط (False)، استفاده کنید.
- دستورات if را برای اتخاذ تصمیماتی بر مبنای نتیجه یک عبارت بولی، بنویسید.
- دستورات switch را برای اتخاذ تصمیمات و داوری‌های پیچیده تر بنویسید.

در فصل ۳، "نوشتن متدها و اعمال سطح دسترسی"، یاد گرفتید که چگونه دستورات مربوط به هم را درون یک متد گروه بندی کنید. هم چنین یاد گرفتید که چگونه از پارامترها برای ارسال اطلاعات به یک متد استفاده کنید و چگونه از دستورات *return* برای ارسال اطلاعات به بیرون از یک متد استفاده کنید. تقسیم کردن یک برنامه به مجموعه ای از متدهای جدا از هم، که هر کدام از متدها برای انجام یک وظیفه یا محاسبه به خصوص طراحی شده، یک استراتژی طراحی ضروری است. بسیاری از برنامه‌ها نیاز دارند که مسائل بزرگ و پیچیده ای را حل کنند. خرد کردن برنامه به درون متدها به شما کمک می‌کند تا این مسائل را درک کنید و روی اینکه چگونه در یک زمان آنها را یک خرده باز کنید، تمرکز کنید. هم چنین شما نیاز دارید که قادر باشید متدهایی را بنویسید که به طور گزینشی اعمال متفاوتی را بسته به شرایط و مقتضیات انجام دهند. در این فصل، خواهید دید که چگونه این وظیفه را به انجام برسانید.

اعلان کردن متغیرهای بولی (Boolean)

در دنیای برنامه‌نویسی (برخلاف دنیای واقعی)، هر چیزی یا سیاه است یا سفید، حق یا نا حق، درست یا غلط. برای مثال، اگر شما یک متغیر با نام *x* اعلان می‌کنید، مقدار 99 را به *x* تخصیص دهید و سپس بپرسید " آیا *x* حاوی مقدار 99 است؟ " جواب قطعاً آری است. اگر شما بپرسید " آیا *x* کمتر از 10 است؟ " جواب قطعاً خیر است. اینها مثال‌هایی از عبارت‌های بولی هستند. یک عبارت بولی همواره به درست (true) یا غلط (false) ارزیابی می‌شود.

توجه پاسخ این سؤال‌ها برای تمامی زبان‌های برنامه‌نویسی، صریح و قطعی نیستند. یک متغیر تخصیص داده نشده یک مقدار تعریف نشده دارد و برای مثال، شما نمی‌توانید بگویید که قطعاً کمتر از 10 است. موضوعاتی مانند این یکی، یک منشأ عمده از خطاها در برنامه‌های C و C++ هستند. کامپایلر ویژوال C# این مشکل را قبل از بررسی متغیر با مطمئن شدن از اینکه شما همواره یک مقدار را به آن متغیر تخصیص می‌دهید، حل می‌کند. در صورتی که سعی کنید محتویات یک متغیر تخصیص داده نشده را بررسی کنید، برنامه شما کامپایل نخواهد شد.

ویژوال C# یک نوع داده با نام *bool* عرضه می‌کند. یک متغیر *bool* می‌تواند یکی از دو مقادیر *true* یا *false* را نگه دارد. برای مثال، سه دستور زیر یک متغیر *bool* با نام *areYouReady* اعلان کرده، *true* را به متغیر تخصیص داده و سپس مقدار آن را به کنسول می‌نویسد:

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // writes True
```

استفاده از عملگرهای بولی (Boolean)

یک عملگر بولی، عملگری است که محاسبه ای را انجام می‌دهد که نتیجه اش یا *true* (صحیح) یا *false* (غلط) است. C# چندین عملگر بولی خیلی سودمند دارد، که ساده ترین آنها عملگر NOT است، که با علامت تعجب (!) نمایش داده می‌شود. عملگر ! یک مقدار بولی را نفی می‌کند، با دادن معکوس آن مقدار. در مثال قبلی، اگر مقدار متغیر *areYouReady* برابر *true* باشد، مقدار عبارت !*areYouReady* برابر *false* است.

شناخت عملگرهای تساوی و رابطه ای

دو عملگر بولی که شما بارها آنها را به کار خواهید برد، عملگرهای برابری (==) و نابرابری (!=) هستند. شما از این عملگرهای باینری استفاده می‌کنید تا دریابید که آیا یک مقدار با مقدار دیگری از همان نوع برابر است یا نه. جدول زیر با استفاده از یک متغیر *int* با نام *age* به عنوان یک مثال، به طور خلاصه می‌گوید که چگونه این عملگرها کار می‌کنند.

عملگر	مصداق	مثال	اگر *age* برابر 42 باشد، نتیجه
==	برابر است با	age == 100	False

!=	نا مساوی است با	age != 0	True

عملگرهایی که به طور نزدیکی به این دو عملگر مربوط هستند، عملگرهای رابطه ای (*relational*) هستند. شما از این عملگرها استفاده می‌کنید تا دریابید که آیا یک مقدار کمتر از یا بزرگتر از مقدار دیگری از همان نوع است یا نه. جدول زیر نشان می‌دهد که چگونه از این عملگرها استفاده کنید.

عملگر	مصداق	مثال	اگر *age* برابر 42 باشد، نتیجه
<	کمتر از	age < 21	False
<=	کمتر از یا مساوی با	age <= 18	False
>	بزرگتر از	age > 16	True
>=	بزرگتر از یا مساوی با	age >= 30	True

توجه عملگر *برابری* == را با عملگر *تخصیص* =، اشتباه نگیرید. عبارت x==y، x را با y مقایسه می‌کند و اگر مقادیر آنها یکسان باشد، مقدار *true* را دارد. عبارت x=y، مقدار y را به x تخصیص می‌دهد.

شناخت عملگرهای منطقی شرطی

ضمناً C# دو عملگر بولی دیگر عرضه می‌کند: عملگر AND منطقی، که با علامت && نمایندگی می‌شود، و عملگر OR منطقی، که با علامت || نمایندگی می‌شود. مجموعاً این عملگرها، به عنوان عملگرهای منطقی شرطی شناخته می‌شوند. هدف آنها ترکیب کردن دو عبارت یا مقدار بولی درون یک نتیجه رشته ای واحد است. این عملگرهای باینری شبیه عملگرهای برابری و رابطه ای هستند از این حیث که عبارت‌هایی که در آنها ظاهر می‌شوند یا صحیح (*true*) هستند یا غلط (*false*)، اما آنها فرق می‌کنند چون که مقادیری که روی آنها عمل می‌کنند باید یا true باشند یا false.

برآمد عملگر && برابر *true* است اگر و فقط اگر هردو عبارت بولی که عملگر روی انها عمل می‌کند *true* باشند. برای مثال، دستور زیر مقدار *true* را به *validPercentage* تخصیص می‌دهد اگر و تنها اگر مقدار *percent* بزرگتر از یا مساوی با 0 باشد و مقدار *percent* کمتر از یا مساوی با 100 باشد.

```
bool validPercentage;
validPercentage = (percent >= 0) && (percent <= 100);
```

توضیح یک خطای عمده تازه کارها تلاش برای ترکیب دو آزمایش با نام گذاری متغیر *percent* تنها برای یک مرتبه است، مانند این:

```
percent >= 0 && <= 100 // this statement will not compile
```

استفاده از پارانتزها کمک می‌کند تا از این نوع اشتباه پرهیز کنید و در ضمن منظور از عبارت را روشن سازید. برای مثال، این دو عبارت را مقایسه کنید:

```
validPercentage = percent >= 0 && percent <= 100
```

و

```
validPercentage = (percent >= 0) && (percent <= 100)
```

هر دو عبارت مقدار یکسانی را برمی‌گردانند زیرا حق تقدم عملگر && کمتر از >= و <= است. گرچه، دومین دستور منظور خود را با یک شیوه خواناتری می‌رساند.

برآمد عملگر || برابر *true* است در صورتی که یکی از عبارت‌های بولی که عملگر روی آنها عمل می‌کند، *true* باشد. شما از عملگر || استفاده می‌کنید تا تعیین کنید که آیا هرگونه ترکیبی از عبارات بولی *true* است یا نه. برای مثال، دستور زیر مقدار *true* را به *invalidPercentage* تخصیص می‌دهد، اگر مقدار *percent* کمتر از 0 باشد یا مقدار *percent* بزرگتر از 100 باشد:

```
bool invalidPercentage;
invalidPercentage = (percent < 0) || (percent > 100);
```

ارزیابی اتصال کوتاه (short-circuiting)

عملگرهای && و || هر دو یک ویژگی با نام اتصال کوتاه (*short-circuiting*) را ارائه می‌کنند. برخی اوقات، هنگام معلوم کردن نتیجه یک عبارت شرطی منطقی، لازم نیست که هر دو عملوند ارزیابی شوند. برای مثال، اگر عملوند دست چپی عملگر && به *false* ارزیابی شود، علیرغم مقدار عملوند دست راستی، نتیجه سرتاسر عبارت باید *false* باشد. به طور مشابه اگر مقدار عملوند دست چپی عملگر || به *true* ارزیابی شود، صرف نظر از مقدار عملوند دست راستی، نتیجه سرتاسر عبارت باید *true* باشد. در این موارد، عملگرهای && و || ارزیابی عملوند دست راستی را کنار می‌گذارند. در اینجا برخی مثال‌ها آورده شده است:

```
(percent >= 0) && (percent <= 100)
```

در این عبارت، اگر مقدار *percent* کمتر از صفر باشد، عبارت بولی سمت چپ عملگر && به *false* ارزیابی می‌شود. این مقدار به معنای این است که نتیجه سراسر عبارت باید *false* باشد و عبارت بولی سمت راست عملگر && ارزیابی نمی‌شود.

```
(percent < 0) || (percent > 100)
```

در این عبارت، اگر مقدار *percent* کمتر از صفر باشد، عبارت بولی سمت چپ عملگر || به *true* ارزیابی می‌شود. این مقدار به معنای این است که نتیجه سراسر عبارت باید *true* باشد و عبارت بولی سمت راست عملگر || ارزیابی نمی‌شود.

اگر با دقت عباراتی را طراحی می‌کنید که از عملگرهای منطقی شرطی استفاده می‌کنند، شما می‌توانید کارایی کد خودتان را با اجتناب از کارهای اضافی بالا ببرید. عبارات بولی ساده را که می‌توانند به سادگی ارزیابی شوند به راحتی در سمت چپ یک عملگر منطقی شرطی جای دهید و عبارات پیچیده تر را در سمت راست بگذارید. در خیلی از موارد، شما درخواهید یافت که برنامه نیاز به ارزیابی کردن عبارات خیلی پیچیده ندارد.

جمع بندی حق تقدم و شرکت پذیری عملگر

جدول زیر حق تقدم و شرکت پذیری تمامی عملگرهایی که تا اینجا درباره آنها آموخته‌اید، جمع بندی می‌کند. عملگرها واقع در رده یکسان حق تقدم یکسان دارند. عملگرهای واقع در رده‌های بالاتر جدول به همه عملگرهای واقع در رده‌های پایین تر حق تقدم دارند.

رده	عملگرها	توضیح	شرکت پذیری
مقدماتی	((	لغو حق تقدم	چپ
	++	نمو پسوندی	
	- -	کاهش پسوندی	
یکانی	!	NOT منطقی	چپ
	+	جمع	
	-	تفریق	
	++	نمو پیشوندی	
	- -	کاهش پیشوندی	
افزاینده	*	ضرب	چپ
	/	تقسیم	
	%	باقیمانده تقسیم (قدر مطلق)	
جمعی	+	جمع	چپ
	-	تفریق	
رابطه ای	<	کمتر از	چپ
	=<	کمتر از یا مساوی با	
	>	بزرگتر از	
	>=	بزرگتر از یا مساوی با	

برابری	==	برابر است با	چپ
	=!	نامساوی است با	
AND شرطی	&&	AND منطقی	چپ
OR شرطی	||	OR منطقی	چپ
تخصیص	=		راست

استفاده از دستورات if برای تصمیم گیری و داوری‌ها

هرگاه بخواهید که مابین اجرا شدن دو بلوک مختلف کد، بسته به نتیجه یک عبارت بولی انتخاب انجام دهید، شما می‌توانید از یک دستور *if* استفاده کنید.

آشنایی با ترکیب نوشتاری/نحوی دستور *if*

ترکیب نحوی/نوشتاری یک دستور *if* همانند زیر است (*if* و *else* واژه‌های کلیدی C# هستند):

```
(if (booleanExpression
statement-1;
else
statement-2;
```

اگر *booleanExpression* به *true* ارزیابی شود، *statement-1* اجرا می‌شود؛ در غیر این صورت *statement-2* اجرا می‌شود. واژه کلیدی *else* و *statement-2* پیرو آن اختیاری هستند. اگر هیچ شرط *else* موجود نباشد و *booleanExpression* برابر *false* باشد، اجرا با هر آن چه کدی که پس از دستور *if* می‌آید، ادامه می‌یابد.

برای مثال، در اینجا یک دستور *if* آورده شده است که یک متغیر، بیانگر ثانیه شمار یک کرونومتر (فی الحال از دقیقه صرف نظر می‌شود)، را یک واحد افزایش می‌دهد. اگر مقدار متغیر *seconds* برابر 59 باشد، به مقدار صفر بازنشانی می‌شود؛ در غیر این صورت، با استفاده از عملگر ++ یک واحد افزایش می‌یابد:

```
int seconds;
...
```

```
if (seconds == 59)
seconds = 0;
else
seconds++;
```

فقط عبارت‌های بولی، لطفاً!

عبارت واقع در دستور *if* باید در میان پارانتزها محصور شده باشد. به طور کلی، عبارت باید یک مقدار بولی باشد. در برخی زبان‌های دیگر (به خصوص C و C++)، شما می‌توانید یک عبارت عدد صحیح را بنویسید و کامپایلر به طور بی صدا مقدار عدد صحیح را به *true* (غیر صفر) یا *false* (صفر) تبدیل می‌کند. C# این رفتار را پشتیبانی نمی‌کند و اگر یک چنین عبارتی بنویسید کامپایلر یک خطا گزارش می‌کند.

اگر در یک دستور *if* شما تصادفاً عملگر تخصیص، =، را به جای عملگر بررسی تساوی، ==، به کار ببرید، کامپایلر اشتباه شما را تشخیص می‌دهد و از کامپایل کردن کد شما سر باز می‌زند. برای مثال:

```
int seconds;
...
if (seconds = 59) // compile-time error
...
if (seconds == 59) // ok
```

تخصیصات تصادفی منبع عمده دیگری از اشکال‌ها در برنامه‌های C و C++ بودند، که به طور بی صدایی می‌توانستند مقدار تخصیص داده شده (59) را به یک عبارت بولی (هر عدد غیر صفر *true* در نظر گرفته می‌شد) تبدیل کنند، با این نتیجه که کد متعاقب دستور *if* می‌توانست هر زمان اجرا شود.

اتفاقاً، شما می‌توانید یک متغیر بولی را به عنوان عبارت لازم برای یک دستور *if* به کار ببرید، اگرچه این متغیر هنوز هم باید در میان پارانتزها محصور شده باشد، همان طور که در این مثال نشان داده شده است:

```
bool inWord;
...
if (inWord == true) // ok, but not commonly used
...
if (inWord) // better
```

استفاده از بلوک‌ها برای گروه بندی دستورات

توجه کنید که ترکیب نوشتاری/نحوی دستور *if* که قبلاً نشان داده شده، یک دستور یگانه را بعد از *if* (*booleanExpression*) و یک دستور یگانه را بعد از واژه کلیدی *else* تعیین می‌کند. برخی اوقات، در صورتی که یک عبارت بولی صحیح باشد، شما می‌خواهید که بیشتر از یک دستور را اجرا کنید. شما می‌توانید دستورات را در یک متد جدید جمع کنید و سپس متد جدید را فرا خوانی کنید، اما یک راه حل ساده تر گروه بندی دستورات درون یک *بلوک* است. یک بلوک در اصل دنباله ای از دستورات جمع شده در میان یک آکولاد باز و بسته است. در ضمن بلوک یک سطح دسترسی (میدان دید) تازه ای را شروع می‌کند. شما می‌توانید متغیرها را در میان یک بلوک تعریف کنید، اما این متغیرها در انتهای بلوک ناپدید می‌شوند.

در مثال زیر، دو دستور که متغیر *seconds* را به صفر بازنشانی کرده و متغیر *minutes* را یک واحد افزایش می‌دهند درون یک بلوک جمع گروه بندی شده‌اند و در صورتی که مقدار *seconds* برابر 59 باشد، تمامی بلوک اجرا می‌شود:

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
seconds = 0;
minutes++;
}
else
seconds++;
```

مهم اگر از آکولادها صرف نظر کنید، کامپایلر C# تنها اولین دستور (seconds = 0;) را با دستور *if* پیوند می‌دهد. هنگامی که برنامه کامپایل می‌شود، دستور بعدی (minutes++;) توسط کامپایلر به عنوان بخشی از دستور *if* شناسایی نمی‌شود. از این گذشته، هنگامی که کامپایلر به واژه کلیدی *else* می‌رسد، آن را با دستور *if* قبلی مرتبط نمی‌کند و در عوض یک خطای ترکیب نوشتاری/نحوی را گزارش می‌کند.

آبشاری کردن دستورات if

شما می‌توانید دستورات *if* را در میان دستورات *if* دیگر تودرتو کنید. با این شیوه، می‌توانید دنباله ای از عبارات بولی را با یکدیگر زنجیر کنید، که یکی بعد از دیگری امتحان می‌شوند تا اینکه یکی از آنها به *true* ارزیابی شود. در مثال زیر، اگر مقدار *day* برابر 0 باشد، اولین بررسی به *true* ارزیابی می‌شود و رشته *"Sunday"* به *dayName* تخصیص می‌یابد. اگر مقدار *day* صفر نباشد، اولین بررسی نافرجام می‌ماند و کنترل به شرط *else* منتقل می‌شود، که دومین دستور *if* را اجرا می‌کند و مقدار *day* را با 1

مقایسه می‌کند. دومین تنها در صورتی به دست می‌آید که بررسی اول *false* باشد. به طور مشابه سومین دستور *if* تنها در صورتی به دست می‌آید که بررسی‌های اول و دوم *false* باشند.

```
if (day == 0)
dayName = "Sunday";
else if (day == 1)
dayName = "Monday";
else if (day == 2)
dayName = "Tuesday";
else if (day == 3)
dayName = "Wednesday";
else if (day == 4)
dayName = "Thursday";
else if (day == 5)
dayName = "Friday";
else if (day == 6)
dayName = "Saturday";
else
dayName = "unknown";
```

در فعالیت زیر، متدی خواهید نوشت که از یک دستور *if* آبشاری برای مقایسه دو تاریخ استفاده می‌کند.

نوشتن دستورات if

ویژوال استودیو ۲۰۰۸ را اگر ازقبل در حال اجرا نباشد، شروع کنید.

پروژه Selection را که در پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 4\Selection واقع در پوشه Documents شما جای گرفته است، باز کنید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

ویژوال استودیو ۲۰۰۸ برنامه را بناکرده و اجرا می‌کند. فرم حاوی دو کنترل *DateTimePicker* با نام‌های *first* و *second* است. (این کنترل‌ها یک تقویم را نشان می‌دهند که به شما اجازه می‌دهد زمانی که پیکان پایین افتادنی را کلیک کردید یک تاریخ را انتخاب کنید.) هر دو کنترل در ابتدا به تاریخ جاری تنظیم شده‌اند.

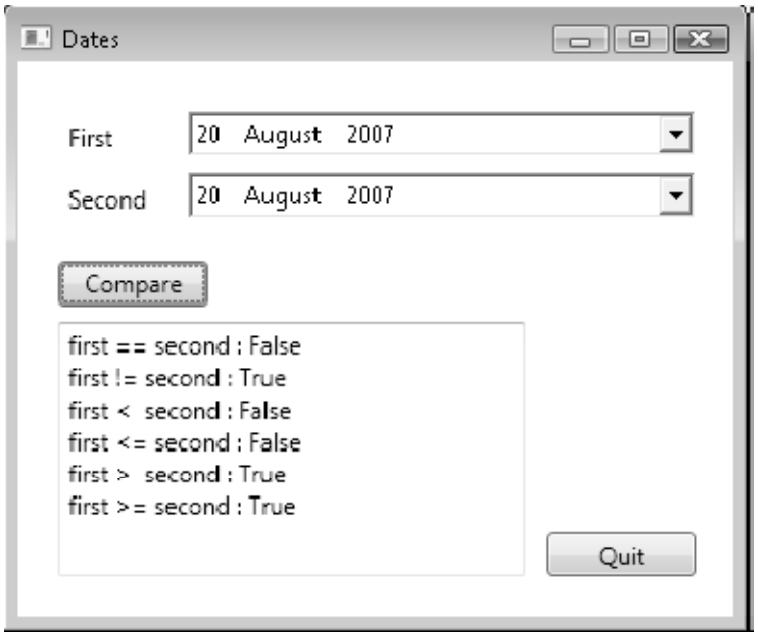
Compare را کلیک کنید.

متن زیر در جعبه متن ظاهر می‌شود:

```
first == second: False
first != second: True
first < second: False
first <= second: False
first > second: True
```

```
first >= second: True
```

عبارت بولی first == second باید *true* شود زیرا هم *first* و هم *second* به تاریخ جاری تنظیم شده‌اند. در واقع، به نظر می‌رسد که تنها عملگر کمتر از (<) و عملگر بزرگتر از یا مساوی با (=>) به درستی کار کنند.



Quit را کلیک کنید تا به محیط برنامه‌نویسی ویژوال استودیو ۲۰۰۸ برگردید.

کد لازم برای Window1.xaml.cs را در پنجره ویرایشگر کد و متن نمایش دهید.

متد *compareClick* را پیدا کنید، که مانند این به نظر می‌رسد:

```
private int compareClick(object sender, RoutedEventArgs e)
{
int diff = dateCompare(first.Value, second.Value);
info.Text = "";
show("first == second", diff == 0);
show("first != second", diff != 0);
show("first < second", diff < 0);
show("first <= second", diff <= 0);
show("first > second", diff > 0);
show("first >= second", diff >= 0);
}
```

هر زمان که کاربر دکمه *Compare* را روی فرم کلیک کند، این متد اجرا می‌شود. این متد مقدار تاریخ‌های نمایش داده شده در کنترل‌های *first* و *second* در روی فرم را بازیابی می‌کند و متد دیگری با

نـام *dateCompare* را بـرای مقایسـه آنهـا فرامـی خوانـد. متـد *dateCompare* را در گـام بعـدی بررسـی خواهید کرد.

متد *show* نتایج مقایسه را در کنترل جعبه متن *info* در روی فرم، جمع بندی می‌کند.

متد *dateCompare* را پیدا کنید، که مانند این به نظر می‌رسد:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
// TO DO
return 42;
}
```

این متد به طور جاری هر زمان که فراخوانی شود، بسته به مقـادیر پارامترهـایش بـه جـای 0، 1- یـا 1+، مقدار یکسانی را برمی‌گرداند. این امر روشـن می‌کند که چرا برنامه همان طـور کـه انتظـار مـی‌رفـت کـار نمی‌کند.

هـدف از ایـن متـد، بررسـی آرگومان‌هـای آن و برگردانـدن یـک مقـدار صـحیح بسـته بـه مقـادیر نسـبی آرگومان‌ها است؛ در صورتی که آنها مقادیر یکسان داشته باشند، متد بایـد صـفر را برگردانـد، اگـر مقـدار آرگومان اول کمتر از مقدار آرگومان دوم باشد متـد بایـد 1- را برگردانـد و اگـر مقـدار آرگومـان اول بزرگتـر از مقدار آرگومان دوم باشد متد باید 1+ را برگرداند. (اگر یک تاریخ بـه ترتیـب زمـانی بعـد از یـک تـاریخ دیگـر بیاید، بزرگتر در نظر گرفته می‌شود.) شما لازم است که منطـق ایـن متـد را بـرای مقایسـه دو تـاریخ بـه درستی پیاده سازی کنید.

توضیح TO DO // و دستور *return* را از متد *dateCompare* حذف کنید.

دستورات زیر را که با فونت درشت نشان داده شده‌اند، به بدنه متد *dateCompare* اضافه کنید:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
int result;
if (leftHandSide.Year < rightHandSide.Year)
result = -1;
else if (leftHandSide.Year > rightHandSide.Year)
result = +1;
}
```

اگر عبارت leftHandSide.Year < rightHandSide.Year برابـر *true* باشـد، تـاریخ واقـع در *leftHandSide* بایـد قبل تر از تاریخ واقع در *rightHandSide* باشـد، از این رو برنامـه متغیـر *result* را بـه 1- تنظـیم مـی‌کنـد. در غیر این صورت، اگر عبارت leftHandSide.Year > rightHandSide.Year به *true* ارزیابی شود، تـاریخ واقـع در *leftHandSide* بایـد بعـد از تـاریخ واقـع در *rightHandSide* باشـد و برنامـه متغیـر *result* را بـه 1+ تنظـیم می‌کند.

اگـر عبـارت leftHandSide.Year < rightHandSide.Year برابـر *false* باشـد و عبـارت leftHandSide.Year > rightHandSide.Year نیز برابر *false* باشـد، خاصیت *Year* هردو تـاریخ بایـد یکسـان باشـد، از ایـن رو برنامـه نیاز دارد که ماه‌ها را در هر تاریخ با هم مقایسه کند.

دستورات زیر را که در فونت درشت نشان داده شده‌اند به بدنه متد *dateCompare* اضافه کنید، بعد از کدی که در گام قبلی وارد کردید:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
...
else if (leftHandSide.Month < rightHandSide.Month)
result = -1;
else if (leftHandSide.Month > rightHandSide.Month)
result = +1;
}
```

این دستورات از منطق مشابهی برای مقایسه ماه‌ها استفاده می‌کند که در گام قبلی برای مقایسه سال‌ها استفاده کرد.

اگر عبارت leftHandSide.Month < rightHandSide.Month به *false* ارزیابی شود و عبارت leftHandSide.Month > rightHandSide.Month نیز به *false* ارزیابی شود، خاصیت *Month* هردو تاریخ باید یکسان باشند، از این رو برنامه نیاز دارد که روزها را در هر تاریخ با هم مقایسه کند.

دستورات زیر را به بدنه متد *dateCompare* اضافه کنید، بعد از کدی که در دو گام قبلی وارد کردید:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
...
else if (leftHandSide.Day < rightHandSide.Day)
result = -1;
else if (leftHandSide.Day > rightHandSide.Day)
result = +1;
else
result = 0;
return result;
}
```

اکنون بایست الگوی این منطق را تشخیص دهید.

اگر leftHandSide.Day < rightHandSide.Day و leftHandSide.Day > rightHandSide.Day هردو *false* باشند، مقادیر خاصیت *Day* برای هردو متغیر باید یکسان باشند. مقادیر *Month* و مقادیر *Year* نیز به ترتیب باید یکسان باشند، چون که منطق برنامه به این دوردست دست یافته، پس دو تاریخ باید یکسان باشند و برنامه مقدار *result* را به 0 تنظیم می‌کند.

دستور نهایی مقدار ذخیره شده در متغیر *result* را برمی‌گرداند.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

برنامه مجدداً بنا شده و اجرا می‌شود. یک مرتبه دیگر، دو کنترل *DateTimePicker*، یعنی *first* و *second*، به تاریخ جاری تنظیم شده‌اند.

Compare را کلیک کنید.

متن زیر در جعبه متن ظاهر می‌شود:

```
first == second: True
first != second: False
first < second: False
first <= second: True
first > second: False
first >= second: True
```

اینها نتایج صحیح برای تاریخ‌های مساوی هستند.

پیکان پایین افتادنی را برای کنترل *DateTimePicker* دومی‌کلیک کنید، و سپس تاریخ فردا را کلیک کنید.

Compare را کلیک کنید.

متن زیر در جعبه متن ظاهر می‌شود:

```
first == second: False
first != second: True
first < second: True
first <= second: True
first > second: False
first >= second: False
```

دوباره، هنگامی که تاریخ اول زودتر از تاریخ دوم باشد، اینها نتایج صحیح هستند.

برخی تاریخ‌های دیگر را نیز بررسی کنید، و تحقیق کنید که نتایج همان طور که شما انتظار داشتید هستند. هنگامی که به کار خود پایان دادید، *Quit* را کلیک کنید.

مقایسه تاریخ‌ها در برنامه‌های دنیای واقعی

اکنون که دیده‌اید که چگونه از یک سری دستورات *if* و *else* نسبتاً بلند و پیچیده استفاده کنید، بایست متذکر شوم که این تکنیکی نیست که شما برای مقایسه تاریخ‌ها در برنامه دنیای واقعی به کار می‌برید. در کتابخانه کلاس چارچوب .NET، تاریخ‌ها با استفاده از یک نوع به خصوص با نام *DateTime* نگه داری می‌شوند. اگر به متد *dateCompare* که در فعالیت قبلی نوشته‌اید، نگاهی بیندازید، خواهید دید که دو پارامتر *leftHandSide* و *rightHandSide* مقادیر *DateTime* هستند. منطقی که شما نوشته‌اید تنها بخش تاریخ این متغیرها را مقایسه می‌کند--- در آنجا یک عنصر زمان نیز وجود دارد. برای اینکه دو مقدار *DateTime* مساوی در نظر گرفته شوند، آنها صرفاً نباید تاریخ یکسان داشته باشند بلکه باید زمان یکسانی نیز داشته باشند. مقایسه کردن تاریخ‌ها و زمانها عملیات متداولی است که نوع *DateTime* یک متد توکار با نام *Compare* فقط برای انجام این کار دارد. متد *Compare* دو آرگومان تاریخ و زمان می‌گیرد و آنها را با هم مقایسه می‌کند، با برگرداندن یک مقدار که نشان می‌دهد که آیا آرگومان

اول کمتر از آرگومان دوم است، در این حالت نتیجه منفی است؛ و بر گردان یک مقدار که نشان می‌دهد که آیا آرگومان اول بزرگتر از آرگومان دوم است، که در این حالت نتیجه مثبت خواهد شد؛ و برگرداندن یک مقدار که نشان می‌دهد که آیا هر دو آرگومان تاریخ و زمان یکسانی را بیان می‌کنند، و در این حالت نتیجه صفر خواهد شد.

استفاده کردن از دستورات switch

برخی اوقات، هنگامی که یک دستور *if* آبشاری را می‌نویسید، همه دستورات *if* شبیه به نظر می‌رسند زیرا آنها همگی یک عبارت یکسان را ارزیابی می‌کنند. تنها تفاوت این است که هر *if* نتیجه عبارت را با یک مقدار متفاوت ارزیابی می‌کند. برای مثال، به بلوک کد زیر توجه کنید که از یک دستور *if* برای آزمایش مقدار واقع در متغیر *day* استفاده می‌کند نتیجه می‌گیرد که *day* کدام روز هفته می‌باشد:

```
if (day == 0)
dayName = "Sunday";
else if (day == 1)
dayName = "Monday";
else if (day == 2)
dayName = "Tuesday";
else if (day == 3)
...
else
dayName = "Unknown";
```

در وضعیت‌ها، اغلب اوقات شما می‌توانید دستور *if* آبشاری را به صورت یک دستور *switch* بازنویسی کنید تا برنامه شما را کارآمدتر و خواناتر سازد.

شناخت ترکیب نوشتاری/نحوی دستور switch

ترکیب نوشتاری/نحوی *switch* همانند زیر است (*switch*، *case* و *default* واژه‌های کلیدی هستند):

```
(switch (controllingExpression
{
:case constantExpression
statements
break;
```

```
:case constantExpression
statements
break;
...
:default
statements
break;
}
```

controllingExpression فقط یک مرتبه ارزیابی می‌شود. پس از آن کنترل به بلوک کد مشخص شده با *constantExpression* که مقدارش برابر با نتیجه *controllingExpression* است، پرش می‌کند. (شناسه یک برچسب حالت (*case label*) خوانده می‌شود.) اجرا ادامه پیدا می‌کند تا آنجا که به دستور *break* برسد، که در این نقطه دستور *switch* خاتمه می‌یابد و برنامه با اولین دستور بعد از آکولاد بسته دستور *switch*، ادامه پیدا می‌کند. اگر هیچ یک از مقادیر *constantExpression* برابر با مقدار *controllingExpression* نباشد، دستورات پایین برچسب اختیاری *default* اجرا می‌شوند.

توجه هر مقدار *constantExpression* باید منحصر به فرد باشد، پس *controllingExpression* تنها با یکی از آنها مطابقت خواهد کرد. اگر مقدار *controllingExpression* با هیچ مقدار *constantExpression* هم خوانی نکند و هیچ برچسب *default* هم در آنجا نباشد، اجرای برنامه با اولین دستور بعد از آکولاد بسته دستور *switch*، ادامه پیدا می‌کند.

برای مثال، شما می‌توانید دستور *if* آبشاری قبلی را همانند دستور *switch* زیر بازنویسی کنید:

```
switch (day)
{
:case 0
dayName = "Sunday";
break;
:case 1
dayName = "Monday";
break;
:case 2
dayName = "Tuesday";
break;
...
:default
dayName = "Unknown";
break;
}
```

پیروی از قواعد دستور switch

دستور *switch* خیلی سودمند است، اما متأسفانه شما همیشه نمی‌توانید هر زمان که دوست دارید از آن استفاده کنید. هر دستور *switch* که می‌نویسید باید به قواعد زیر وفادار بماند:

- شما می‌توانید دستور *switch* را تنها در مورد انواع داده اصلی، مانند int یا string به کاربرید. با هر نوع دیگری (شامل float و double)، مجبور خواهید شد که از یک دستور if استفاده کنید.
- برچسب‌های case باید عبارت‌های ثابت باشند مانند 42 یا "42". اگر لازم است که مقادیر برچسب case را در زمان اجرا محاسبه کنید، باید از یک دستور if استفاده کنید.
- برچسب‌های case باید عبارت‌های منحصر به فرد باشند. به عبارت دیگر دو برچسب case نمی‌توانند مقدار یکسان داشته باشند.
- می‌توانید تعیین کنید که می‌خواهید دستورات یکسانی را برای بیشتر از یک مقدار با فراهم کردن لیستی از برچسب‌های case و بدون هیچ دستور حائلی اجرا کنید، که در این حالت کد مربوط به برچسب نهایی در لیست برای همه حالات در آن لیست اجرا می‌شود. گرچه، اگر یک برچسب یک یا چند دستور مربوط داشته باشد، اجرا نمی‌تواند به برچسب‌های بعدی نزول کند و کامپایلر یک خطا تولید می‌کند. برای مثال:

```
switch (trumps)
{
:case Hearts
case Diamonds: // Fall-through allowed – no code between labels
color = "Red"; // Code executed for Hearts and Diamonds
break;
:case Clubs
color = "Black";
case Spades: // Error – code between labels
color = "Black";
break;
}
```

توجه دستور *break* شیوه خیلی متداولی برای متوقف کردن عدم نتیجه گیری نزول به برچسب بعدی است، اما شما می‌توانید از یک دستور *return* یا یک دستور *throw* نیز استفاده کنید. دستور *throw* در فصل ٦، "مدیریت کردن خطاها و اخطارها"، توضیح داده می‌شود.

قواعد عدم نتیجه گیری و نزول به برچسب بعدی برای switch

از آن جایی که اگر در آنجا هرگونه کد حائلی وجود داشته باشد، شما نمی‌توانید به طور تصادفی از یک برچسب *case* به برچسب دیگری نزول کنید، می‌توانید به طور آزادانه بخش‌های یک دستور *switch* را بدون تأثیر گذاشتن بر فحوای آن بازآرایی کنید (شامل برچسب *default*، که به طور قراردادی معمولاً به عنوان آخرین برچسب جای داده می‌شود اما نبایستی این گونه باشد).

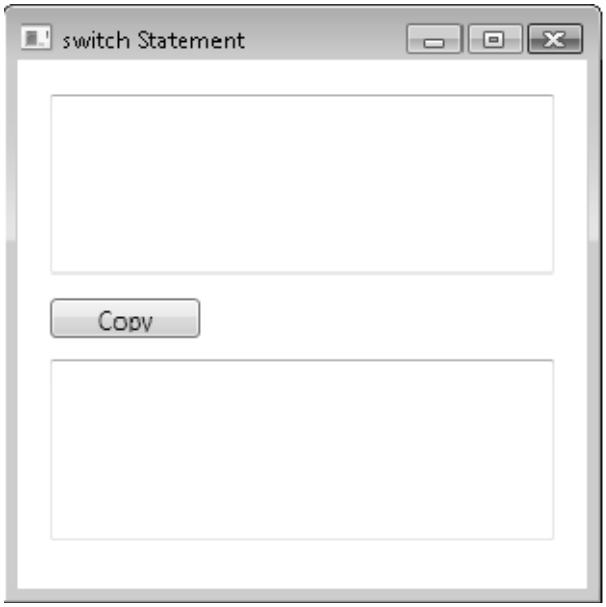
برنامه نویسان C و C++ بایست توجه کنند که دستور *break* برای هر حالت (case) در یک دستور *switch* اجباری است (حتی حالت *default*). این الزام چیز خوبی است؛ فراموش کردن دستور *break* در برنامه‌های C یا C++ امر خیلی شایعی است، که به اجرا اجازه می‌دهد که به برچسب بعدی نزول کند و منجر به اشکالاتی شود که تشخیص دادن آنها خیلی مشکل است.

اگر واقعاً بخواهید، شما می‌توانید با استفاده از یک دستور *goto* برای رفتن به برچسب *case* یا *default* بعدی، از نزول C/C++ در C# تقلید کنید. گرچه به طور کلی استفاده از دستور *goto* توصیه نمی‌شود و این کتاب به شما نشان نمی‌دهد که از ان استفاده کنید!

در فعالیت زیر، شما یک برنامه را کامل خواهید کرد که کاراکترهای یک رشته را می‌خواند و هر کاراکتر را به نمایش XML اش نگاشت می‌کند. برای مثال، کاراکتر >، یک معنای ویژه در XML دارد. اگر شما داده ای داشته باشید که حاوی این کاراکتر باشد، باید به متن "<" برگردانده شود به طوری که یک پردازنده XML می‌داند که آن داده است و نه بخشی از یک دستورالعمل XML . قواعد مشابهی به کاراکترهای <، امپرسند (&)، علامت نقل قول منفرد (') و علامت نقل قول مضاعف (") اعمال می‌شود. یک دستور *switch* خواهید نوشت که مقدار کاراکتر را بررسی می‌کند و کاراکترهای مخصوص XML را به عنوان برچسب‌های *case* به دام می‌اندازد.

نوشتن دستورات switch

ویژوال استودیو ۲۰۰۸ را اگر از قبل در حال اجرا نیست، آغاز کنید.
پروژه SwitchStatement قرار گرفته در پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 4\SwitchStatement واقع در پوشه Documents خود را باز کنید.
در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
ویژوال استودیو ۲۰۰۸ برنامه را بنا کرده و اجرا می‌کند. برنامه یک فرم حاوی دو جعبه متن را که توسط یک دکمه *Copy* از هم جدا شده‌اند، نمایش می‌دهد.



متن ساده زیر را در جعبه متن بالایی تایپ کنید:

```
inRange = (lo <= number) && (hi >= number);
```

Copy را کلیک کنید.

دستور کلمه به کلمه میان جعبه متـــن پایینی کپی مـی‌شـود و هـیچ ترجمـه ای از >، & یـا < رخ نمی‌دهد.

فرم را ببندید و به ویژوال استودیو ۲۰۰۸ برگردید.

کد مربوط به Window1.xaml.cs را در پنجره ویرایشگر کد و مـتن نمـایش دهیـد و متـد *copyOne* را پیـدا کنید.

متد *copyOne* کاراکتر مشخص شده به عنوان پارامتر ورودی اش را به انتهای متن نمایش داده شـده در جعبه متن پایینی کپی می‌کند. در حال حاضر، *copyOne* حاوی یک دستور *switch* با یک بخـش *default* یگانه است. در چند قدم بعدی، شما این دستور *switch* را ویرایش خواهید کرد تا کاراکترهـایی را کـه در XML معنادار هستند به نگاشت XML آنها تبدیل کند. برای مثال، کاراکتر < به رشته "<" تبدیل خواهـد شد.

دستورات زیر را به دستور *switch* بعـد از آکـولاد بـاز دسـتور و مسـتقیماً قبـل از برچسـب *default* اضـافه کنید:

```
:case '<'
target.Text += "&lt;";
break;
```

اگر کاراکتر جاری که در حال کپی شدن است < باشد، این کد رشته "<" را به متن خروجـی در مکـان آن الحاق می‌کند.

دستورات زیر را به دستور *switch* بعد از دستور *break* که شما اندکی قبل اضافه کرده‌اید و بالای برچسب *default* اضافه کنید:

```
:case '>'
target.Text += "&gt;";
break;
:case '&'
target.Text += "&amp;";
break;
:case '\"'
target.Text += "&#34;";
break;
:case '\''
target.Text += "&#39;";
break;
```

توجه علامت نقل قول منفرد (') و علامت نقل قول مضاعف (") در C# و نیز در XML معنای ویژه ای دارند--- آنها برای محدود کردن ثابت‌های کاراکتر و رشته به کار می‌روند. بک اسلش (\) در دو برچسب *case* نهایی یک کاراکتر گریز است که با عث می‌شود که کامپایلر C# با این کاراکترها به جای حائل به عنوان لیترال برخورد کند.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

متن زیر را در جعبه متن بالایی تایپ کنید:

```
inRange = (lo <= number) && (hi >= number);
```

Copy را کلیک کنید.

دستور به میان جعبه متن پایینی کپی می‌شود. اکنون، هر کاراکتر متحمل نگاشت XML پیاده سازی شده در دستور *switch* می‌شود. جعبه متن هدف متن زیر را نمایش می‌دهد:

```
inRange = (lo &lt;= number) &amp;&amp; (hi &gt;= number)
```

با رشته‌های دیگر آزمایش کنید و تحقیق کنید که تمامی کاراکترهای ویژه (<، >، " ، & و ') به درستی اداره می‌شوند.

فرم را ببندید.

- اگر می‌خواهید که به فصل بعدی بروید
ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۵ مراجعه کنید.

- اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید
در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۵

استفاده کردن از تخصیص مرکب و دستورات تکرار

بعد از کامل کردن این فصل، قادر خواهید بود که:

- مقدار یک متغیر را با استفاده از عملگرهای تخصیص مرکب، به روز کنید.
- دستورات تکرار *while*، *for* و *do* را بنویسید.
- در میان یک دستور *do* چرخیده و ببینید که چگونه مقادیر متغیرها تغییر می‌کنند.

در فصل ۴، " استفاده از دستورات داوری"، یاد گرفتید که چگونه از ساختارهای *if* و *switch* برای اجرای دستورات به طور انتخابی، استفاده کنید. در این فصل، خواهید دید که چگونه از دستورات تکرار (یا ایجاد حلقه) گوناگونی برای اجرای یک یا چند دستور به طور تکراری استفاده کنید. هرگاه دستورات تکرار را می‌نویسید، معمولاً نیازمند این هستید که تعداد تکرارهایی را که می‌خواهید انجم دهید، کنترل کنید. شما می‌توانید، با استفاده از یک متغیر، به هنگام سازی مقدار آن با هر تکرار و متوقف کردن فرایند تکرار وقتی که متغیر به یک مقدار به خصوص می‌رسد، به این منظور دست یابید. ضمناً شما درباره عملگرهای تخصیص ویژه ای که باید آنها را برای به هنگام سازی مقدار یک متغیر در این شرایط به کار ببرید، مطالبی را یاد خواهید گرفت.

استفاده از عملگرهای تخصیص مرکب

شما قبلاً دیده‌اید که چگونه از عملگرهای ریاضیاتی برای ایجاد مقادیر جدید استفاده کنید. برای مثال، دستور زیر از عملگر به اضافه (+) برای نمایش یک مقدار در پنجره کنسول که از متغیر *answer* به‌اندازه 42 بزرگتر است، استفاده می‌کند:

```
Console.WriteLine(answer + 42);
```

هم چنین شما قبلاً دیده‌اید که چگونه از دستورات تخصیص برای تغییر مقدار یک متغیر استفاده کنید. دستور زیر از عملگر تخصیص برای تغییر مقدار *answer* به 42 استفاده می‌کند:

```
answer = 42;
```

در صورتی که می‌خواهید 42 را به مقدار یک متغیر اضافه کنید، شما می‌توانید عملگر تخصیص و عملگر جمع را ترکیب کنید. برای مثال، دستور زیر مقدار 42 را به *answer* اضافه می‌کند. بعد از این که دستور اجرا می‌شود، مقدار *answer* از آن چه قبلاً بوده است به‌اندازه 42 بیشتر می‌شود:

```
answer = answer + 42;
```

با این که این دستور کار می‌کند، احتمالاً هرگز نخواهد دید که یک برنامه نویس با تجربه کدی مانند این را بنویسد. اضافه کردن یک مقدار به یک متغیر آن قدر متداول است که C# به شما اجازه می‌دهد که با استفاده از عملگر =+، این کار را به شیوه ی تند نویسی بنویسید.

```
answer += 42;
```

شما می‌توانید از این تند نویسی برای ترکیب هر عملگر ریاضیاتی با عملگر تخصیص استفاده کنید، همان طور که در جدول زیر نشان داده شده است. این عملگرها مجموعاً به عنوان عملگرهای تخصیص مرکب شناخته می‌شوند.

این را بنویسید	این را ننویسید
variable *= number;	variable = variable * number;
variable /= number;	variable = variable / number;
variable %= number;	variable = variable % number;
variable += number;	variable = variable + number;
variable -= number;	variable = variable - number;

توضیح عملگرهای تخصیص مرکب، حق تقدم و شرکت پذیری یکسانی را با عملگرهای تخصیص ساده شریک هستند.

عملگر =+ روی رشته‌ها نیز عمل می‌کند؛ این عملگر یک رشته را به انتهای رشته ای دیگر می‌چسباند. برای مثال، کد زیر "Hello John" را روی کنسول نمایش می‌دهد:

```
string name = "John";
string greeting = "Hello ";
```

```
greeting += name;
Console.WriteLine(greeting);
```

شما نمی‌توانید هیچ یک از عملگرهای تخصیص مرکب دیگر را روی رشته‌ها، به کار ببرید.

توجه هنگام افزایش یا کاهش دادن یک متغیر به‌اندازه یک واحد، از عملگرهای افزایش دادن (++) و کاهش دادن (- -) به جای یک عملگر تخصیص مرکب استفاده کنید. برای مثال، دستور

```
count += 1;
```

را با

```
count++;
```

جایگزین کنید.

نوشتن دستورات while

شما می‌توانید در حالی که برخی شروط صحیح (true) باشند، از دستور *while* برای اجرای یک دستور به صورت مکرر استفاده کنید. ترکیب نوشتاری/نحوی دستور *while* همانند زیر است:

```
(while (booleanExpression
statement
```

عبارت بولی (booleanExpression) ارزیابی می‌شود و اگر true باشد، statement اجرا می‌شود و پس از آن عبارت بولی دوباره ارزیابی می‌شود. در صورتی که عبارت هنوز هم صحیح (true) باشد، دستور تکرار می‌شود و پس از آن عبارت بولی (booleanExpression) دوباره ارزیابی می‌شود. این فرایند تا زمانی که عبارت بولی به false ارزیابی شود، ادامه می‌یابد. پس از آن اجرا با اولین دستور پس از دستور *while* ادامه پیدا می‌کند. یک دستور *while* مشابهت‌های نحوی بسیاری را با یک دستور *if* شریک می‌شود (در واقع، ترکیب نوشتاری یکسان است به استثنای واژه کلیدی):

- ☐ عبارت باید یک عبارت بولی باشد.
- ☐ عبارت بولی باید در میان پارانتزها نوشته می‌شود.
- ☐ اگر عبارت بولی اولین بار که ارزیابی شد، به false ارزیابی شود، دستور اجرا نمی‌شود.
- ☐ اگر می‌خواهید دو یا چند دستور را تحت کنترل یک دستور *while* انجام دهید، شما باید از آکولادها---{ }--- برای گروه بندی کردن آن دستورات در یک بلوک استفاده کنید.

در اینجا یک دستور *while* آورده شده است که مقادیر 0 تا 9 را به کنسول می‌نویسد:

```
int i = 0;
while (i < 10)
```

```
{
Console.WriteLine(i);
i++;
}
```

همه دستورات *while* باید در نقطه معینی خاتمه یابند. یک اشتباه شایع مبتدی‌ها این است که فراموش می‌کنند که یک دستور را جای دهند که باعث شود عبارت بولی سرانجام به false ارزیابی شود و حلقه را خاتمه دهد، که این اشتباه منجر می‌شود که یک برنامه تا ابد اجرا شود. در مثال بالا، دستور i++ این نقش را انجام می‌دهد.

توجه متغیر *i* در حلقه *while* تعداد تکرارهایی را که حلقه انجام می‌دهد، کنترل می‌کند. این یک شیوه متداولی است و متغیری که این نقش را انجام می‌دهد برخی اوقات متغیر دیده بان (*Sentinel*) خوانده می‌شود.

در فعالیت زیر، شما یک حلقه *while* می‌نویسید که محتوای یک فایل متنی را، هر بار یک سطر، بازیابی کرده و هر سطر را به یک جعبه متن واقع در روی یک فرم می‌نویسد.

نوشتن دستور while

با استفاده از ویژوال استودیو ۲۰۰۸، پروژه WhileStatement واقع در پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement قرار گرفته در پوشه Documents خود را باز کنید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.

ویژوال استودیو ۲۰۰۸ برنامه را بنا کرده و اجرا می‌کند. برنامه یک ناظر فایل متنی ساده است که شما می‌توانید از ان برای انتخاب یک فایل و نمایش محتویات آن استفاده کنید.

Open File را کلیک کنید.

جعبه گفتگوی *Open* باز می‌شود.

به پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement\WhileStatement واقع در پوشه Documents خود بروید.

فایل Window1.xaml.cs را انتخاب کنید و *Open* را کلیک کنید.

اسم فایل، Window1.xaml.cs، در جعبه متن کوچک در روی فرم ظاهر می‌شود، اما محتویات فایل در جعبه متن بزرگتر ظاهر نمی‌شود. این امر به خاطر این است که شما هنوز کدی را که محتویات فایل را خوانده و نمایش می‌دهد، پیاده سازی نکرده‌اید. شما این عملکرد را در گامهای زیر اضافه خواهید کرد.

فرم را ببندید و به ویژوال استودیو ۲۰۰۸ برگردید.

کد مربوط به فایل Window1.xaml.cs را در پنجره ویرایشگر کد و متن نمایش دهید و متد *openFileDialogFileOk* را پیدا کنید.

این متد هنگامی که کاربر بعد از انتخاب یک فایل در جعبه گفتگوی *Open*، دکمه *Open* را کلیک می‌کند، اجرا می‌شود. بدنه متد که به طور جاری پیاده سازی شده، همانند زیر است:

```
private void openFileDialogFileOk(object sender, System.ComponentModel.
CancelEventArgs e)
{
string fullPathname = openFileDialog.FileName;
FileInfo src = new FileInfo(fullPathname);
filename.Text = src.Name;
// add while loop here
}
```

اولین دستور یک متغیر *string* با نام *fullPathname* اعلان می‌کند و آن را با خاصیت *FileName* شیء *openFileDialog* مقداردهی می‌کند. این خاصیت حاوی نام کامل (از جمله پوشه) فایل منبع انتخاب شده در جعبه گفتگوی *Open* است.

توجه شیء *openFileDialog* یک وهله از کلاس *openFileDialog* است. این کلاس متدهایی را عرضه می‌کند که شما می‌توانید از آنها برای نمایش جعبه گفتگوی *Windows Open* استاندارد، انتخاب یک فایل و بازیابی نام و مسیر فایل انتخاب شده استفاده کنید.

دومین دستور یک متغیر *FileInfo* با نام *src* اعلان می‌کند و آن را با یک شیء مقداردهی می‌کند که بیانگر فایل انتخاب شده در جعبه گفتگوی *Open* است. (*FileInfo* کلاس عرضه شده توسط Microsoft .NET Framework است که شما می‌توانید از آن برای دست کاری فایل‌ها استفاده کنید.)
سومین دستور خاصیت *Text* کنترل *filename* را به خاصیت *Name* متغیر *src* تخصیص می‌دهد. خاصیت *Name* متغیر *src* اسم فایل انتخاب شده در جعبه گفتگوی *Open* را نگه می‌دارد، اما بدون اسم پوشه.
این دستور اسم فایل را در جعبه متن واقع در روی فرم نمایش می‌دهد.
توضیح *// add while loop* را با دستور زیر جایگزین کنید:

```
source.Text = "";
```

متغیر *source* به جعبه متن بزرگ در روی فرم اشاره می‌کند. تنظیم کردن خاصیت *Text* متغیر *source* به رشته تهی ("") هر متنی را که به طور جاری در این جعبه متن نمایش یافته، پاک می‌کند.
دستور زیر را بعد از سطری که‌اندکی قبل به متد *openFileDialogFileOk* اضافه کردید، تایپ کنید:

```
TextReader reader = src.OpenText();
```

این دستور یک متغیر *TextReader* با نام *reader* اعلان می‌کند. *TextReader* کلاس دیگر عرضه شده توسط Microsoft .NET Framework است که شما می‌توانید از آن برای خواندن جریان‌های کاراکتر از منابعی مانند فایل، استفاده کنید. این کلاس در فضای اسمی *System.IO* جای گرفته است. کلاس *FileInfo* متد *OpenText* را برای باز کردن یک فایل برای خواندن، عرضه می‌کند. این دستور فایل انتخاب شده توسط کاربر در جعبه گفتگوی *Open* را باز می‌کند به طوری که متغیر *reader* می‌تواند محتویات این فایل را بخواند.

دستور زیر را بعد از سطر قبلی که شما به متد *openFileDialogFileOk* اضافه کردید، اضافه کنید:

```
string line = reader.ReadLine();
```

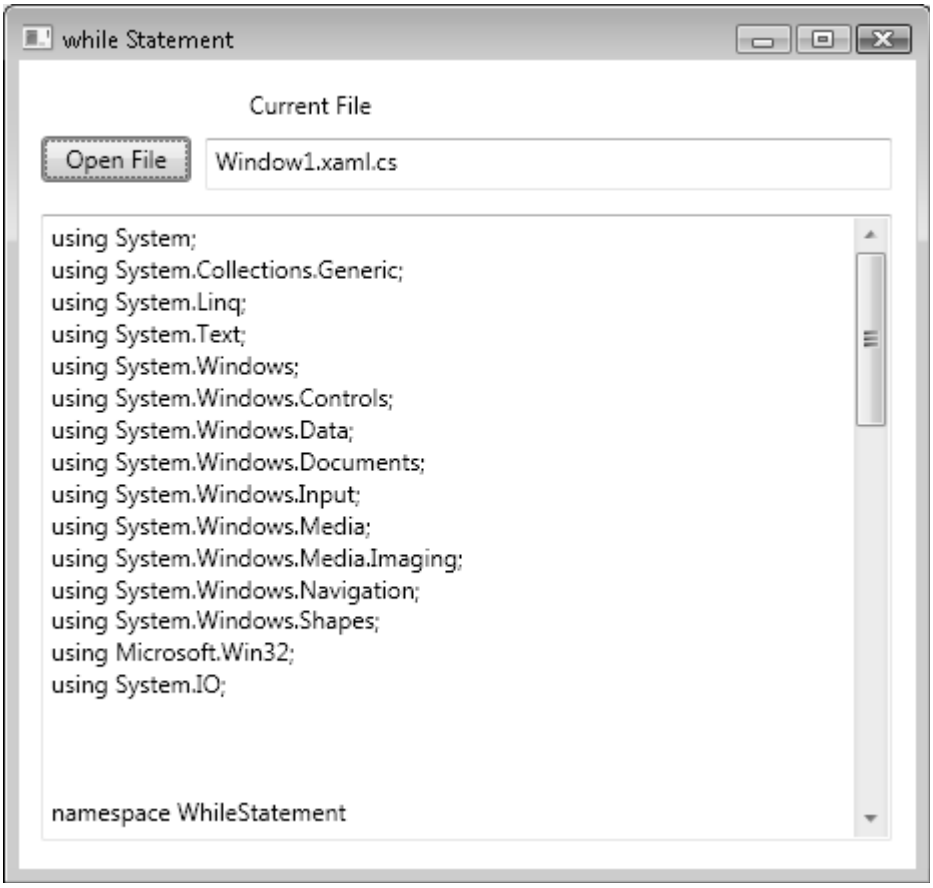
این دستور یک متغیر *string* با نام *line* اعلان می‌کند و متد *reader.ReadLine* را برای خواندن اولین سطر از فایل به درون این متغیر فراخوانی می‌کند. این متد یا سطر بعدی را برمی‌گرداند یا اگر در آنجا هیچ سطر دیگری برای خواندن موجود نبود، یک مقدار ویژه با نام *null* را برمی‌گرداند. (اگر در آنجا در اصل هیچ سطری موجود نباشد، فایل بایستی تهی باشد.)
دستورات زیر را بعد از کدی که‌اندکی قبل وارد کرده‌اید، به متد *openFileDialogFileOk* اضافه کنید:

```
while (line != null)
{
source.Text += line + '\n';
line = reader.ReadLine();
}
```

این یک حلقه *while* است که در هر بار یک سطر فایل را بازیابی می‌کند تا زمانی که هیچ سطر دیگری در آنجا در دسترس نباشد.
عبارت بولی در شروع حلقه *while* مقدار واقع در متغیر *line* را بررسی می‌کند. اگر این مقدار *null* نباشد، بدنهی حلقه سطر جاری متن را با چسباندن آن به انتهای خاصیت *Text* جعبه متن *source*، همراه با یک کاراکتر سطر جدید ('\n' --- متد *ReadLine* از شیء *TextReader* به محض اینکه هر سطر را می‌خواند، کاراکترهای سطر جدید را بیرون می‌دهد، از این رو کد نیازمند این است که آن را دوباره به عقب اضافه کند). پس از ان حلقه *while* سطر بعدی متن را قبل از انجام دادن تکرار بعدی، می‌خواند. هنگامی که در آنجا هیچ سطر دیگری در فایل موجود نباشد و متد *ReadLine* مقدار null را برگرداند، حلقه *while* خاتمه می‌یابد.
دستور زیر را بعد از آکولاد بسته واقع در انتهای حلقه *while* اضافه کنید:

```
reader.Close();
```

این دستور فایل را می‌بندد.
در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
هرگاه فرم ظاهر شد، *Open File* را کلیک کنید.
در جعبه گفتگوی *Open File*، به پوشه \Microsoft Press\Visual CSharp Step by Step\Chapter 5\WhileStatement\WhileStatement واقع در پوشه Documents خود بروید. فایل Window1.xaml.cs را انتخاب کنید و *Open* را کلیک کنید.
اکنون محتویات فایل انتخاب شده در جعبه متن ظاهر می‌شود--- شما باید کدی را که‌اندکی قبل ویرایش کرده‌اید، تشخیص دهید:



سراسر متن را مرور کنید و متد *openFileDialogFileOk* را پیدا نمایید. بررسی کنید که این متد حاوی کدی است که شما اندکی قبل اضافه کردید.

فرم را ببندید و به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ برگردید.

نوشتن دستورات for

اغلب دستورات *while* ساختار عمومی‌زیر را دارند:

```
initialization
while (Boolean expression)
{
statement
update control variable
}
```

با یک دستور *for*، شما می‌توانید یک نسخه رسمی تر از این نوع ساختار را با ترکیب کردن initialization، Boolean expression و update (مقدمه چینی و عملیات پاک سازی حلقه) بنویسید. شما

دستور *for* را بسیار سودمند خواهید یافت زیرا خیلی سخت است که هر یک از این سه بخش را فراموش کنید. در اینجا ترکیب نوشتاری/نحوی یک دستور *for* آورده شده است:

```
for (initialization; Boolean expression; update control variable)
statement
```

شما می‌توانید حلقه *while* را که قبلاً نشان داده شده، مجدداً جمله بندی کنید تا اعداد صحیح را از 0 تا 9 به صورت حلقه *for* زیر نمایش دهد:

```
for (int i = 0; i < 10; i++)
{
Console.WriteLine(i);
}
```

initialization (مقداردهی) یک مرتبه در شروع حلقه رخ می‌دهد. پس از آن، اگر Boolean expression (عبارت بولی) به *true* ارزیابی شود، statement (دستور) اجرا می‌شود. به هنگام‌سازی control variable (متغیر کنترل) رخ می‌دهد و پس از آن Boolean expression (عبارت بولی) دوباره ارزیابی می‌شود. اگر شرط هنوز هم *true* باشد، دستور دوباره اجرا می‌شود، متغیر کنترل به هنگام سازی می‌شود، Boolean expression دوباره ارزیابی می‌شود و الی آخر.

توجه کنید که initialization تنها یک مرتبه اتفاق می‌افتد، دستور واقع در بدنه حلقه همواره قبل از اینکه به هنگام سازی (update) اتفاق بیافتد، اجرا می‌شود و اینکه update قبل از اینکه Boolean expression (عبارت بولی) مجدداً ارزیابی شود، رخ می‌دهد.

شما می‌توانید از هر یک از سه بخش یک دستور *for* صرف نظر کنید. اگر از Boolean expression صرف‌نظر کنید، مقدار پیش فرضش *true* خواهد بود. دستور *for* زیر برای همیشه اجرا می‌شود:

```
for (int i = 0; ;i++)
{
Console.WriteLine("somebody stop me!");
}
```

اگر از بخش‌های initialization و update صرف نظر کنید، شما یک حلقه *while* خواهید داشت که به طور عجیب غریبی املا شده است:

```
int i = 0;
(for (; i < 10;
{
Console.WriteLine(i);
i++;
}
```

توجه بخش‌های initialization، Boolean expression و update control variable از یک دستور *for* باید همواره توسط نقطه ویرگول از هم جدا شوند، حتی زمانی که آنها صرف نظر شده باشند.

اگر لازم باشد، شما می‌توانید initializationها و updateهای متعددی را در یک حلقه *for* تدارک ببینید (شما تنها می‌توانید یک Boolean expression داشته باشید). برای رسیدن به این منظور، initializationها و updateها را با ویرگول از هم جدا کنید، همان طور که در مثال زیر نشان داده شده است:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
...
}
```

به عنوان یک مثال پایانی، در این جا حلقه *while* از فعالیت قبلی آورده شده است که به صورت یک حلقه *for* از نو طرح ریزی شده است.

```
for (string line = reader.ReadLine(); line != null; line = reader.ReadLine())
{
source.Text += line + '\n';
}
```

توضیح این سبک خوبی است که از آکولادها برای مشخص کردن بلوک دستور مربوط به بدنه دستورات *if*، *while* و *for* استفاده کرد حتی اگر بلوک تنها حاوی یک دستور باشد. با نوشتن بلوک، اضافه کردن دستورات را به بدنه بلوک در آینده راحت تر می‌سازید. بدون بلوک، برای اضافه کردن دستوری دیگر، شما باید به یاد داشته باشید که هم دستور اضافی را اضافه کنید و هم آکولادها را، و در این حالت آکولادها به راحتی فراموش می‌شوند.

شناخت دامنه دستور for

شما ممکن است فهمیده باشید که می‌توانید یک متغیر را در بخش initialization (مقداردهی اولیه) یک دستور *for* اعلان کنید. میدان دید و دامنه‌ی آن متغیر محدود به بدنه دستور *for* است و هرگاه دستور *for* خاتمه می‌یابد، ناپدید می‌شود. اولاً شما نمی‌توانید از آن متغیر بعد از اینکه دستور *for* خاتمه یافت، استفاده کنید زیرا دیگر در میان دامنه نیست. در اینجا یک مثال آورده شده است:

```
for (int i = 0; i < 10; i++)
{
```

```
...
}
Console.WriteLine(i); // compile-time error
```

دوماً، شما می‌توانید دو یا چند دستور *for* را در کنار یکدیگر بنویسید که از اسم متغیر یکسان دوباره استفاده کنند زیرا هر متغیر در یک دامنه متفاوت قرار دارد. در اینجا مثالی آورده شده است:

```
for (int i = 0; i < 10; i++)
{
...
}
for (int i = 0; i < 20; i += 2) // okay
{
...
}
```

نوشتن دستورات do

دستورات *while* و *for* هر دو عبارت بولی را در شروع حلقه تست می‌کنند. این سخن به معنای این است که اگر در همان بررسی اول، عبارت به *false* ارزیابی شود، بدنه حلقه اجرا نمی‌شود، حتی یک مرتبه. دستور *do* متفاوت است؛ عبارت بولی این دستور بعد از هر تکرار ارزیابی می‌شود، از این رو بدنه همواره دست کم یک بار اجرا می‌شود.

ترکیب نوشتاری/نحوی دستور *do* همانند زیر است (نقطه ویرگول نهایی را فراموش نکنید):

```
do
statement
while (booleanExpression);
```

در صورتی که بدنه حلقه بیشتر از یک دستور را در بر داشته باشد، باید از یک بلوک *statement* (دستور) استفاده کنید. در اینجا یک نسخه از مثالی که مقادیر 0 تا 9 را به کنسول می‌نویسد آورده شده است، اکنون این مثال با استفاده از یک دستور *do* ساخته شده است.

```
int i = 0;
do
{
Console.WriteLine(i);
i++;
}
```

```
while (i < 10);
```

دستورات break و continue

در فصل ۴، دستور *break* را دیدید که برای پرش کردن به خارج از یک دستور *switch* به کار برده می‌شد. ضمناً شما می‌توانید از یک دستور *break* برای پرش به خارج از بدنه یک دستور تکرار استفاده کنید. هرگاه که یک حلقه را در هم می‌شکنید، اجرا بلافاصله از حلقه خارج شده و از اولین دستور بعد از حلقه ادامه پیدا می‌کند. نه به هنگام سازی (update) حلقه و نه شرط تمدید حلقه دوباره اجرا نمی‌شوند.

در مقابل، دستور *continue* باعث می‌شود که برنامه بلافاصله تکرار بعدی حلقه را انجام دهد (بعد از ارزیابی مجدد عبارت بولی). در این جا نسخه ای دیگر از مثالی که مقادیر 0 تا 9 را به کنسول می‌نویسد، در این لحظه با استفاده از دستورات *break* و *continue* آورده شده است:

```
int i = 0;
while (true)
{
  Console.WriteLine("continue " + i);
  i++;
  if (i < 10)
  continue;
  else
  break;
}
```

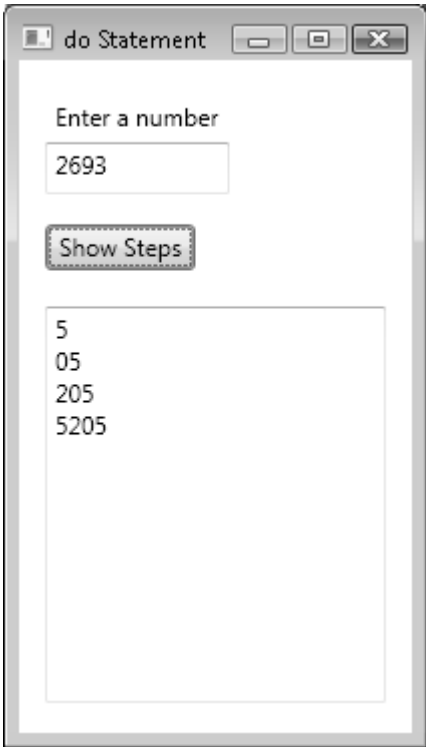
این کد کاملاً مخوف می‌باشد. در بسیاری از رهنمودهای برنامه نویسی توصیه شده که از *continue* با احتیاط استفاده شود یا اصلاً استفاده نشود زیرا این دستور اغلب با کدی که فهمیدن آن سخت است، مرتبط است. در ضمن رفتار *continue* کاملاً ظریف و دقیق است. برای مثال، اگر یک دستور *continue* را از درون یک دستور *for* اجرا کنید، بخش update قبل از انجام دادن تکرار بعدی حلقه اجرا می‌شود.

در فعالیت زیر، شما یک دستور *do* برای تبدیل یک عدد کامل مثبت به نمایش رشته ای آن عدد در شیوه هشت- هشتی (منظور مبنای هشت: octal) خواهید نوشت.

بررسی دستور do

با استفاده از ویژوال استودیو ۲۰۰۸ پروژه DoStatement واقع در پوشه Microsoft Press\Visual CSharp Step by Step\Chapter 5\DoStatement قرار گرفته در پوشه Documents خود را باز کنید.

در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
برنامه یک فرم را نمایش می‌دهد که دو جعبه متن و یک دکمه با نام *Show Steps* دارد. هرگاه یک عدد صحیح مثبت را (برنامه با اعداد صحیح منفی کار نمی‌کند) در جعبه متن بالایی تایپ کنید و *Show Steps* را کلیک کنید، برنامه عددی را که تایپ کرده‌اید می‌گیرد و آن را به نمایش رشته ای مقدار اکتال (مبنای هشت) همان عدد تبدیل می‌کند. برنامه از یک الگوریتم مشهور استفاده می‌کند که به طور تکراری یک عدد را بر 8 تقسیم می‌کند و باقیمانده هر مرحله را محاسبه می‌کند. جعبه متن پایینی گامهای به کار رفته برای ساخت این نمایش اکتال (مبنای ۸) را نشان می‌دهد.
عدد 2693 را در جعبه متن بالایی تایپ کنید و *Show Steps* را کلیک کنید.
جعبه متن پایینی، گامهای به کار رفته برای ایجاد نمایش اکتال 2693 (یعنی 5205) را نمایش می‌دهد.



پنجره را ببندید و به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ برگردید.
کد مربوط به فایل Window1.xaml.cs را در پنجره ویرایشگر کد و متن نمایش دهید.
متد *showStepsClick* را پیدا کنید. این متد هنگامی که کاربر دکمه *Show Steps* را روی فرم کلیک می‌کند، اجرا می‌شود.
این متد حاوی دستورات زیر است:

```
int amount = int.Parse(number.Text);
steps.Text = "";
string current = "";
do
{
int nextDigit = amount % 8;
```

```
int digitCode = '0' + nextDigit;
char digit = Convert.ToChar(digitCode);
current = digit + current;
steps.Text += current + "\n";
amount /= 8;
}
while (amount != 0);
```

اولین دستور، مقدار رشته واقع در خاصیت *Text* جعبه متن *number* را با استفاده از متد *Parse* برای نوع *int*، به *int* تبدیل می‌کند:

```
int amount = int.Parse(number.Text);
```

دومین دستور متن نمایش یافته در جعبه متن پایینی (با نام *steps*) را با تنظیم کردن خاصیت *Text* آن به رشته تهی، پاک می‌کند:

```
steps.Text = "";
```

سومین دستور یک متغیر *string* با نام *current* اعلان می‌کند و آن را با رشته تهی مقداردهی می‌کند:

```
string current = "";
```

کار واقعی در این متد توسط دستور *do* انجام می‌شود، که از دستور چهارم شروع می‌شود:

```
do
{
...
}
while (amount != 0);
```

الگوریتم به طور تکراری محاسبات صحیح را برای تقسیم متغیر *amount* بر 8 انجام می‌دهد و باقیمانده را تعیین می‌کند؛ باقیمانده بعد از هر تقسیم متوالی رقم بعدی در رشته در حال ایجاد شدن را تشکیل می‌دهد. سرانجام، هرگاه *amount* به 0 کاهش یافت، حلقه خاتمه می‌یابد. توجه کنید که بدنه باید دست کم یک مرتبه اجرا شود. این رفتار دقیقاً آن چیزی است که لازم می‌شود زیرا که حتی عدد 0 هم یک رقم اکتال (مبنای هشت) دارد.

اگر خیلی با دقت به کد نگاه کنید خواهید دید که اولین دستور در میان حلقه *do* این است:

```
int nextDigit = amount % 8;
```

این دستور یک متغیر *int* با نام *nextDigit* را اعلان می‌کند و آن را با باقیمانده ی تقسیم مقدار واقع در *amount* بر 8، مقداردهی می‌کند. این مقدار عددی مابین 0 و 7 خواهد شد.
دستور بعدی این است:

```
int digitCode = '0' + nextDigit;
```

این دستور نیازمند اندکی توضیح است! کاراکترها یک کد منحصر به فرد مطابق با مجموعه کاراکتر به کار برده شده توسط سیستم عامل دارند. در مجموعه کاراکترهایی که خیلی اوقات توسط سیستم عامل ویندوز به کار برده می‌شوند، کد مربوط به کاراکتر '0' مقدار صحیح 48 را دارد. کد مربوط به کاراکتر '1' برابر 49 است، کد مربوط به کاراکتر '2' برابر 50 است و ... تا کد مربوط به کاراکتر '9' که مقدار صحیح 57 را دارد. C# به شما اجازه می‌دهد تا با یک کاراکتر همانند یک عدد صحیح برخورد کرده و محاسبات را روی آن انجام دهید، اما هرگاه شما چنین کاری را انجام دهید، C# از کد کاراکتر به عنوان مقدار استفاده می‌کند. از این رو عبارت '0' + nextDigit در واقع منجر به مقداری مابین 48 و 55، متناظر با کد مربوط به رقم اکتال معادل ، می‌شود (به یاد داشته باشید که *nextDigit* مابین 0 و 7 نوشته خواهد شد).
سومین دستور در میان حلقه do این است:

```
char digit = Convert.ToChar(digitCode);
```

این دستور یک متغیر *char* با نام *digit* اعلان می‌کند و آن را با نتیجه فراخوان متد *Convert.ToChar(digitCode)* مقداردهی می‌کند. متد *Convert.ToChar* یک عدد صحیح که یک کد کاراکتر را نگه می‌دارد، می‌گیرد و کاراکتر متناظر را برمی‌گرداند. از این رو، مثلاً در صورتی که *digitCode* مقدار 54 را داشته باشد، *Convert.ToChar(digitCode)* کاراکتر '6' را برمی‌گرداند. به منظور جمع بندی، سه دستور اول در حلقه *do* نمایش کاراکتری کم اهمیت ترین(سمت راست‌ترین) رقم اکتال (مبنای ۸) متناظر با عددی را که کاربر تایپ می‌کند، مشخص کرده‌اند. کار بعدی این است که این رقم را به ابتدای رشته ای که نتیجه خواهد شد، الحاق کند، مانند این:

```
current = digit + current;
```

دستور بعد در میان حلقه *do* این است:

```
steps.Text += current + "\n";
```

این دستور رشته حاوی ارقامی را که تا اینجا برای نمایش اکتال عدد تولید شده است به جعبه متن *Steps* اضافه می‌کند.
دستور نهایی در میان *do* این است

```
amount /= 8;
```

این یک دستور تخصیص مرکب است و معادل است با نوشتن این دستور amount = amount / 8; . در صورتی که مقدار *amount* برابر 2693 باشد، بعد از اینکه این دستور اجرا می‌شود مقدار *amount* برابر 336 می‌شود.

بالاخره، شرط واقع در عبارت *while* در انتهای رشته ارزیابی می‌شود:

```
while (amount != 0)
```

از آن جایی که مقدار *amount* هنوز هم 0 نیست، حلقه تکرار بعدی را انجام می‌دهد.

در فعالیت نهایی، شما از دیباگر ویژوال استودیو ۲۰۰۸ برای گام زدن در میان دستور do پیشین برای کمک به خود برای درک این که این دستور چگونه کار می‌کند، استفاده خواهید کرد.

ردیابی دستور do

در پنجره ویرایشگر کد و متن، در حالی که فایل Window1.xaml.cs را نمایش می‌دهد، مکان نمای ماوس را به اولین دستور متد *showStepsClick* حرکت دهید:

```
int amount = int.Parse(number.Text);
```

در هر جای اولین دستور راست کلیک کنید و پس از آن *Run To Cursor* را کلیک کنید.

هر گاه فرم ظاهر شد، 2693 را در جعبه متن بالایی تایپ کنید و پس از آن *Show Steps* را کلیک کنید.

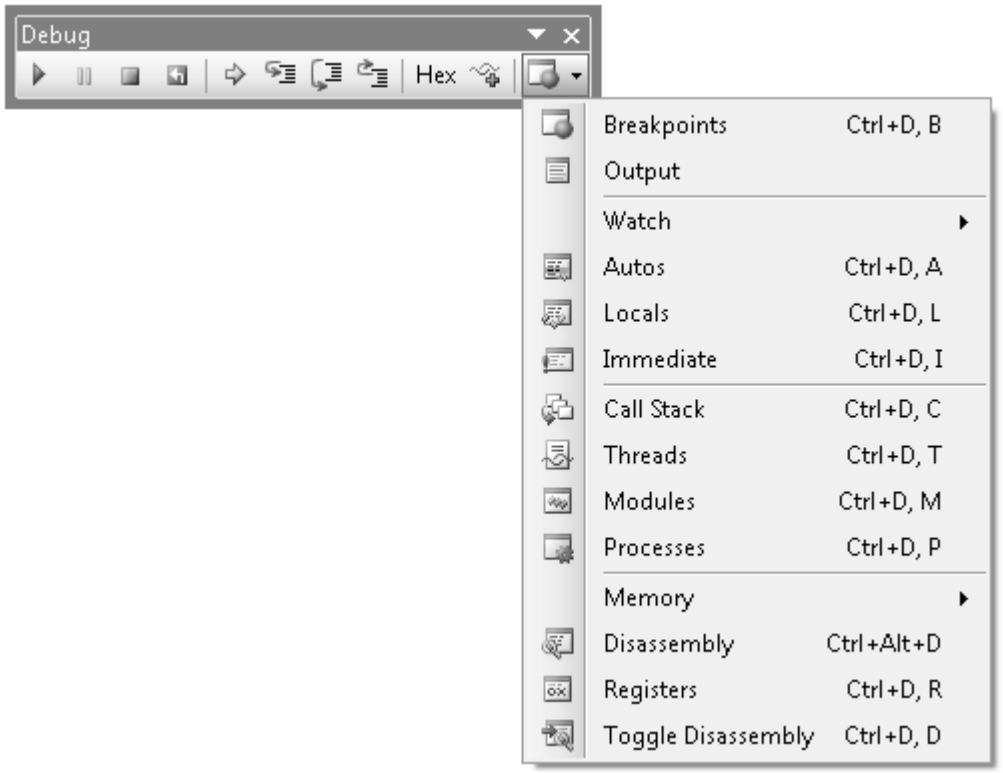
برنامه متوقف می‌شود و شما در ویژوال استودیو ۲۰۰۸ در حالت دیباگ مستقر می‌شوید. یک پیکان زرد رنگ در حاشیه سمت چپی پنجره ویرایشگر کد و متن دستور جاری را نشان می‌دهد.

میله ابزار *Debug* را اگر قابل مشاهده نباشد، نمایش دهید. (در میله منو، View|Toolbar|Debug را انتخاب کنید.)

در میله ابزار *Debug*، پیکان پایین افتادنی *Windows* را کلیک کنید.

توجه آیکون *Windows* سمت راست ترین آیکون در میله ابزار *Debug* است.

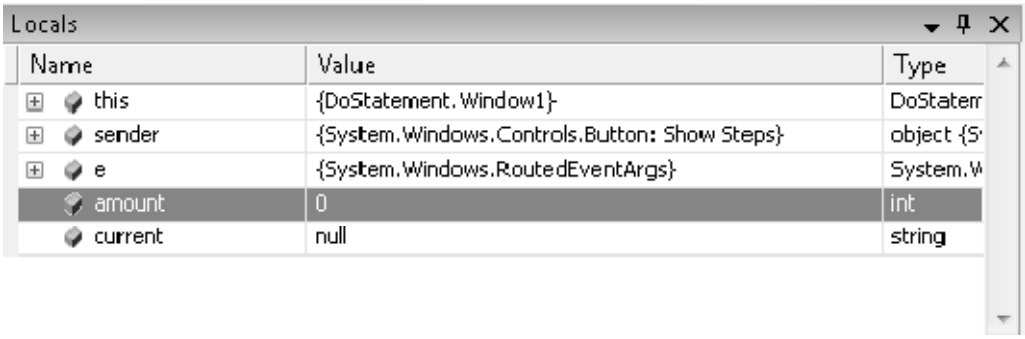
منوی زیر ظاهر می‌شود:



توجه در صورتی که در حال استفاده از Microsoft Visual C# 2008 Express Edition هستید، منوی میان بر حاوی یک زیر مجموعه از این آنهایی است که در تصویر نشان داده شده است.

در منوی پایین افتادنی، گزینه *Locals* را کلیک کنید.

پنجره *Locals* ظاهر می‌شود (در صورتی که از قبل باز نباشد). این پنجره اسم، مقدار و نوع متغیرهای محلی واقع در متد جاری، از جمله متغیر محلی *amount* را نمایش می‌دهد. توجه کنید که مقدار *amount* به طور جاری برابر 0 است:



در منوی *Debug*، دکمه *Step Into* را کلیک کنید.

دیباگر دستور زیر را اجرا می‌کند:

```
int amount = int.Parse(number.Text);
```

مقدار *amount* در پنجره *Locals* به 2693 تغییر می‌کند و پیکان زرد رنگ به دستور بعدی حرکت می‌کند.
Step Into را دوباره کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
steps.Text = "";
```

این دستور پنجره *Locals* را متأثر نمی‌کند زیرا *steps* یک کنترل واقع در روی فرم است و یک متغیر محلی نیست. پیکان زرد رنگ به دستور بعدی حرکت می‌کند.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
string current = "";
```

پیکان زرد رنگ به آکولاد باز در شروع حلقه *do* حرکت می‌کند.
Step Into را کلیک کنید.
پیکان زرد رنگ به اولین دستور در میان حلقه do حرکت می‌کند. حلقه do حاوی سه متغیر محلی متعلق به خودش است: *nextDigit*، *digitCode* و *digit*. توجه کنید که این سه متغیر محلی در پنجره *Locals* ظاهر می‌شود و این که مقدار همه این متغیرها برابر 0 است.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
int nextDigit = amount % 8;
```

مقدار *nextDigit* در پنجره *Locals* به 5 تغییر پیدا می‌کند. این مقدار باقیمانده تقسیم 2693 بر 8 است.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
int digitCode = '0' + nextDigit;
```

مقدار *digitCode* در پنجره *Locals* به 53 تغییر پیدا می‌کند. این مقدار کد کاراکتر '5' (5 + 48) است.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
char digit = Convert.ToChar(digitCode);
```

مقدار *digit* در پنجره *Locals* به '5' تغییر پیدا می‌کند. پنجره *Locals* مقادیر *char* را هم با مقدار عددی متضمن (در این حالت، 53) و هم نمایش کاراکتری ('5') نمایش می‌دهد.
توجه کنید که در پنجره *Locals*، مقدار متغیر *current* برابر "" (تهی) است.

Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
current = current + digit;
```

مقدار *current* در پنجره *Locals* به "5" تغییر پیدا می‌کند.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
steps.Text += current + "\n";
```

این دستور متن "5" را در جعبه متن *steps* نمایش می‌دهد که با یک کاراکتر سطر جدید پی گرفته می‌شود تا باعث شود که خروجی بعدی در روی سطر بعدی در جعبه متن نمایش داده شود. (فرم فعلاً در پشت ویژوال استودیو ۲۰۰۸ پنهان است، از این رو شما نمی‌توانید آن را ببینید.)
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
amount /= 8;
```

مقدار *amount* در پنجره *Locals* به *336* تغییر پیدا می‌کند. پیکان زرد رنگ به آکولاد در انتهای حلقه *do* حرکت می‌کند.
Step Into را کلیک کنید.
پیکان زرد رنگ به دستور *while* منتقل می‌شود.
Step Into را کلیک کنید.
دیباگر دستور زیر را اجرا می‌کند:

```
while (amount != 0);
```

مقدار *amount* برابر 336 است و عبارت 0 =! 336 به *true* ارزیابی می‌شود، از این رو حلقه *do* تکرار بعدی را انجام می‌دهد. پیکان زرد رنگ به آکولاد باز در شروع حلقه *do* پرش می‌کند.
Step Into را کلیک کنید.
پیکان زرد رنگ به اولین دستور در میان حلقه *do* جابه جا می‌شود.
به طور مکرر *Step Into* را کلیک کنید تا در میان سه تکرار بعدی حلقه *do* گام زنید و ببینید که چگونه مقادیر متغیرها در پنجره *Locals* تغییر پیدا می‌کنند.
در انتهای چهارمین تکرار حلقه، مقدار *amount* فعلاً 0 است و مقدار *current* برابر "5205" است.
پیکان زرد رنگ روی شرط تمدید حلقه *do* است:

```
while (amount != 0);
```

مقدار *amount* اکنون 0 است، از این رو عبارت amount != 0 به *false* ارزیابی می‌شود و حلقه *do* خاتمه می‌یابد.

Step Into را کلیک کنید.

دیباگر دستور زیر را اجرا می‌کند:

```
while (amount != 0);
```

همان طور که پیش بینی می‌شد، حلقه do خاتمه می‌یابد و پیکان زرد رنگ به آکولاد بسته در انتهای متد *showStepsClick* حرکت می‌کند.

دکمه *Continue* را در میله ابزار *Debug* کلیک کنید.

فرم ظاهر می‌شود در حالی که چهار مرحله به کار رفته برای ایجاد نمایش اکتال (مبنای ۸) عدد 2693 را نمایش می‌دهد: "5"، "05"، "205" و "5205".

فرم را ببندید تا به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ بازگردید.

تبریک می‌گم! شما با موفقیت دستورات بامعنای *while* و *do* را نوشته‌اید و دیباگر ویژوال استودیو ۲۰۰۸ را برای ردیابی دستور *do* استفاد کرده‌اید.

- ☐ اگر می‌خواهید که به فصل بعدی بروید
  ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۶ مراجعه کنید.

- ☐ اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید
  در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۶

مدیریت خطاها و استثناها

بعد از تمام کردن این فصل، قادر خواهید بود که:

- استثناها را با استفاده از دستورات *try*، *catch* و *finally* اداره کنید.
- با استفاده از واژه‌های کلیدی *checked* و *unchecked* سرریز عدد صحیح را کنترل کنید.
- با استفاده از واژه کلیدی *throw* استثناها را از متدهای مال خودتان برانگیزانید.
- با استفاده از یک بلوک *finally* مطمئن شوید که کد همواره اجرا می‌شود، حتی بعد از اینکه یک استثنا رخ داده باشد.

اکنون شما دستورات هسته ای ویژوال C# را دیده‌اید که برای نوشتن و خواندن متدها، اعلان متغیرها، استفاده از عملگرها برای ایجاد مقادیر، نوشتن دستورات *if* و *switch* برای اجرای دستورات به طور انتخابی و نوشتن دستورات *while*، *for* و *do* برای اجرای کد به طور تکراری، نیاز دارید که بدانید. گرچه، فصول قبلی احتمال (امکان) چیزهایی را که می‌توانند بد کار کنند مورد توجه قرار نداده‌اند. مطمئن شدن از این که یک قطعه از کد همواره همان طور که مورد انتظار است کار می‌کند، خیلی سخت است. ناکامی ها و شکست‌ها به دلایل بسیاری می‌توانند رخ دهند، که بسیاری از آنها خارج از کنترل شما به عنوان یک برنامه نویس هستند. هر برنامه ای که می‌نویسید باید قادر باشند که ناکامی ها و شکست‌ها را شناسایی کنند و به روشی مطلوب و برازنده به آنها رسیدگی کنند. در فصل نهایی بخش I، شما یاد خواهید گرفت که چگونه استثناها را برانگیزانید تا علامت دهید که یک خطا رخ داده است و چگونه از دستورات *try*، *catch* و *finally* برای گرفتن و بررسی خطاهایی که این استثناها آنها را نمایندگی می‌کنند، استفاده کنید. در انتهای این فصل، یک زیربنای یکپارچه از C# خواهید داشت، که روی آن بخش II، " شناخت زبان C# "، را بنا خواهید کرد.

کنار آمدن با خطاها

این واقعیتی از زندگی است که چیزهای بد گاهی اتفاق می‌افتند. تایرها پنچر می‌شوند، باتری‌ها کهنه می‌شوند، آچار پیچ گوشتی‌ها هرگز در جایی که رها کرده‌اید، نیستند و کاربران برنامه‌های شما به شیوه ای غیر قابل پیش بینی رفتار می‌کنند. هنگامی که یک برنامه اجرا می‌شود، خطاها می‌توانند تقریباً در هر مرحله ای اتفاق بیافتند، پس شما چگونه آنها را تشخیص می‌دهید و برای ترمیم آنها تلاش می‌کنید؟ در طی سالیان، تعدادی از مکانیزم‌ها استنتاج شده‌اند. هر زمان که یک متد بی نتیجه می‌ماند، یک رویکرد نوعی بهینه شده توسط سیستم‌های قدیمی‌مانند UNIX با برنامه ریزی سیستم عامل به منظور تنظیم یک متغیر سرتاسری ویژه درگیر بود. سپس، بعد از هر فراخوان به یک متد، شما متغیر سرتاسری را بررسی می‌کردید تا ببینید که آیا متد موفق بوده است یا نه.

C# و خیلی از زبان‌های شیء گرای مدرن دیگر خطاها را به این شیوه اداره نمی‌کنند. این روش خیلی پردردسر است. این زبان‌ها در عوض از استثناها (*exceptions*) استفاده می‌کنند. در صورتی که می‌خواهید برنامه‌های نیرومند C# را بنویسید، نیازمند این هستید که مطالبی را درباره استثناها بیاموزید.

سنجیدن کد و در اختیار گرفتن استثناها

C# با استفاده از استثناها و گرداننده‌های استثنا، جداسازی کد رسیدگی به استثنا را از کدی که جریان اصلی برنامه را پیاده سازی می‌کند، آسان می‌سازد. برای نوشتن برنامه‌های با اطلاع از استثناها، شما نیازمند این هستید که دو چیز را بدانید:
کد خود را در میان یک بلوک *try* بنویسید (*try* یک ویژه کلیدی C# است). هرگاه کد اجرا می‌شود، تلاش می‌کند که همه دستورات واقع در میان بلوک *try* را اجرا کند و اگر هیچ یک از دستورات استثنای تولید نکنند، یکی بعد از دیگری تا آخر اجرا می‌شوند. گرچه، در صورتی که یک وضعیت خطا اتفاق بیافتد، استثنا به بیرون از بلوک *try* و به دورن قطعه دیگری از کد که برای گرفتن و اداره استثناها طراحی شده، پرش می‌کند--- یعنی یک گرداننده *catch* .

بلافاصله بعد از بلوک *try*، یک یا چند گرداننده *catch* برای رسیدگی به هر وضعیت امکان پذیر خطا بنویسید (*catch* واژه کلیدی دیگری برای C# است). یک گرداننده *catch* نامزد دریافت و اداره یک نوع مخصوص استثنا است و شما می‌توانید بعد از یک بلوک *try* چندین گرداننده *catch* داشته باشید، که هر یک از آنها برای به دام انداختن و پردازش یک استثنا ویژه طراحی شده‌اند به طوری که می‌توانید گردانندهای متفاوتی برای خطاهای متفاوتی که می‌توانند در بلوک *try* رخ دهند، تدارک ببینید.

در اینجا یک مثال از کدی واقع در یک بلوک *try* که آورده شده است که سعی می‌کند که رشته‌هایی را که یک کاربر در جعبه متن‌های معینی روی یک فرم تایپ کرده است، به مقادیر صحیح تبدیل کند، یک متد را برای محاسبه یک مقدار فراخوان کند و نتیجه را به جعبه متن دیگری بنویسد. تبدیل یک رشته به یک عدد صحیح نیازمند این است که رشته حاوی یک نمایش معتبر و نه دنباله ی قراردادی معینی از کاراکترها می‌باشد. در صورتی که رشته حاوی کاراکترهای نامعتبری باشد، متد *int.Parse* یک

FormatException را می‌تاباند و اجـرا بـــه گردانـنده *catch* متنـاظر منتقـل مـی‌شـود. هرگـاه گردانـنده *catch* خاتمه می‌یابد، برنامه با اولین دستور بعد از گردانندە ادامه می‌یابد:

```
try
{
int leftHandSide = int.Parse(lhsOperand.Text);
int rightHandSide = int.Parse(rhsOperand.Text);
int answer = doCalculation(leftHandSide, rightHandSide);
result.Text = answer.ToString();
}
catch (FormatException fEx)
{
// Handle the exception
...
}
```

اداره کردن یک استثنا

یک گردانندە *catch* از ترکیب نوشتاری/نحوی مشابه با آن چه که یک پارامتر متـد بـرای تعیـین اسـتثنای که باید گرفته شود، استفاده می‌کند. در مثال پیشین، هرگاه یک *FormatException* صادر شـود، متغیـر *fEx* با یک شیء حاوی جزییات اسـتثنا پـر مـی‌شـود. نـوع *FormatException* تعـدادی خاصـیت دارد کـه شما می‌توانید بررسی کنید تا علت واقعی استثنا را مشخص کنید. خیلی از این خاصـیت‌ها بـرای همـه استثناها مشترک هستند. برای مثال، خاصیت *Message* حاوی یک توضیح متنـی از خطـایی اسـت کـه باعث بروز استثنا شـــده است. شما می‌توانید از این اطلاعات هنگام اداره کردن اسـتثنا اسـتفاده کنیـد، شاید با ثبت کردن جزییات در یک فایل log یا نمایش دادن یک پیغام معنادار به کاربر و پرسش از کاربر بـه منظور سنجش دوباره.

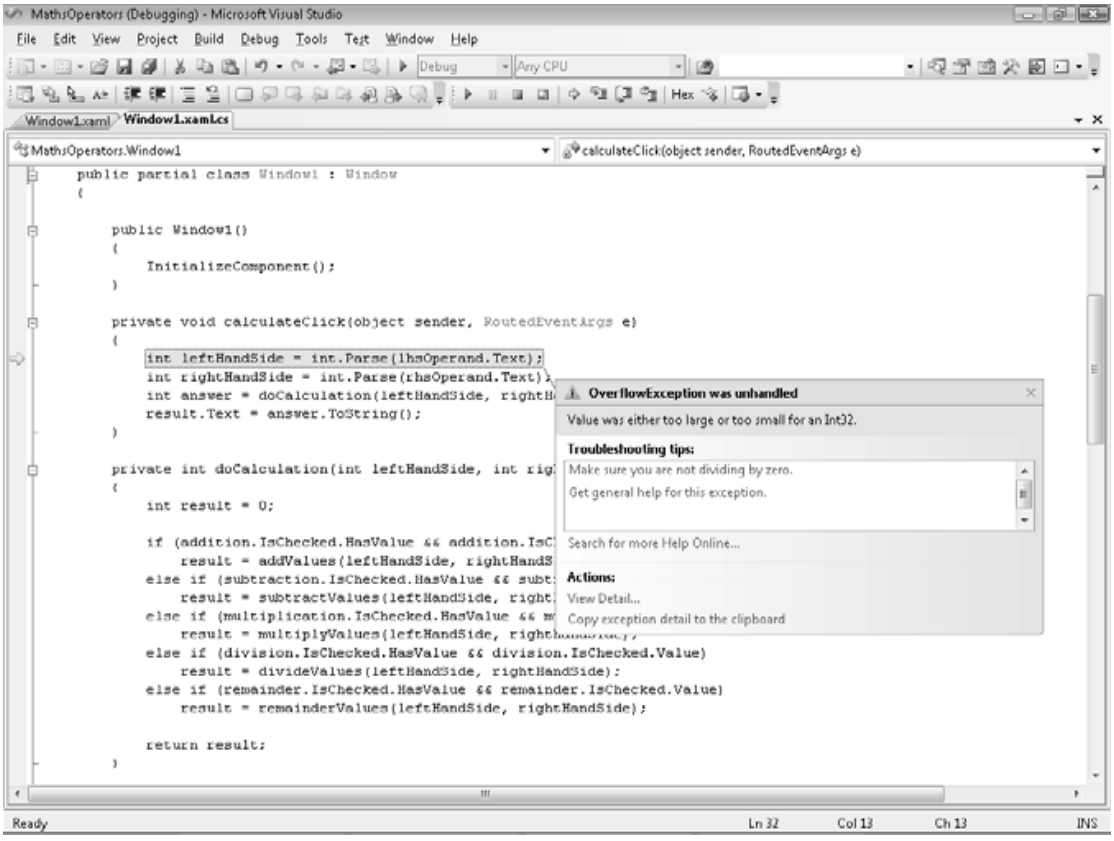
استثناهای اداره نشده

در صورتی که یک بلـوک *try* یـک اسـتثنا را صـادر کنـد و در آنجـا هـیچ گردانـنده *catch* متنـاظری موجـود نباشد، چه اتفاقی می‌افتد؟ در مثال قبلی، ممکن اسـت کـه جعبـه مـتن *lhsOperand* حـاوی نمـایش رشته یک عدد صحیح معتبر باشد، اما عـدد صحیحی کـه او بیـان مـی‌کنـد خـارج از دامنـه اعـداد صـحیح پشتیبانی شده توسـط C# اسـت (بـرای مثـال، "2147483648"). در ایـن حالـت، دسـتور *int.Parse* یـک *OverflowException* صـادر مـی‌کنـد، کـه ایـن اسـتثنا توسـط گردانـنده *catch* بـا نـام *FormatException* گرفته نمی‌شود. اگر این حالت اتفاق بیافتد، در صـورتی کـه بلـوک *try* بخشـی از یـک متـد باشـد، اجـرا بلافاصله از متد خارج مـی‌شـود و بـه متـد فراخوانـنده برمـی‌گـردد. اگـر متـد فراخوانـنده از یـک بلـوک *try* اسـتفاد ه مـی‌کنـد، runtime تـلاش مـی‌کنـد تـا یـک تطـابق گردانـنده *catch* بعـد از بلـوک *try* در متـد فراخواننده پیدا کند و آن را اجرا کند. اگر متد فراخواننده از یک بلوک *try* استفاده نکنـد، یـا در آن جـا هـیچ

تطابق گرداننده *catch* موجود نباشد، اجرا بلافاصله از متد فراخواننده خارج می‌شود و به فراخواننده بازمی‌گردد، جایی که فرایند تکرار می‌شود. اگر سرانجام یک تطابق گرداننده *catch* یافت شود، گرداننده اجرا می‌شود و اجرا با اولین دستور بعد از گرداننده *catch* در متد جاذب ادامه می‌یابد.

مهم توجه کنید که بعد از گرفتن یک استثنا، اجرا در متد حاوی بلوکی که استثنا گرفته شده است، ادامه می‌یابد. اگر اجرا در یک متدی غیر از متدی که حاوی گرداننده *catch* است، رخ داده باشد، کنترل به متدی که باعث بروز استثنا شده برنمی‌گردد.

اگر، بعد از بازگشت به عقب در میان لیست فراخوانی متدها به صورت آبشاری، runtime عاجز از یافتن یک تطابق گرداننده *catch* باشد، برنامه با یک استثنا اداره نشده خاتمه می‌یابد. در صورتی که در حال اجرای برنامه در حالت دیباگ در ویژوال استودیو ۲۰۰۸ هستید (شما گزینه *Start Debugging* را در منوی *Debug* برای اجرای برنامه انتخاب کرده‌اید)، جعبه گفتگوی اطلاعات زیر ظاهر می‌شود و برنامه در میان دیباگر می‌افتد، در حالی که به شما اجازه می‌دهد که علت استثنا را تعیین کنید:



استفاده از گردانندههای catch متعدد

بحث قبلی، این که چگونه خطاهای متفاوت انواع متفاوتی از استثناها را برای بیان انواع متفاوتی از شکست‌ها بازتاب می‌دهند، پررنگ می‌کند. برای کنار آمدن با این وضعیت‌ها، شما می‌توانید گرداننده‌های *catch* متعددی، یکی بعد از دیگری، تدارک ببینید، مانند این:

```
try
{
int leftHandSide = int.Parse(lhsOperand.Text);
int rightHandSide = int.Parse(rhsOperand.Text);
int answer = doCalculation(leftHandSide, rightHandSide);
result.Text = answer.ToString();
}
catch (FormatException fEx)
{
//...
}
catch (OverflowException oEx)
{
//...
}
```

در اختیار گرفتن استثناهای متعدد

مکانیزم در اختیار گرفتن استثنا، عرضه شده توسط C# و Microsoft .NET Framework کاملاً جامع و فراگیر است. استثناهای متفاوت بسیاری در .NET Framework تعریف شده‌اند و هر برنامه ای که می‌نویسید قادر خوهد بود تا بسیاری از آنها را بازتاباند! خیلی بعید است که بخواهید برای هر استثنا ممکنی که کد شما می‌تواند آن را بازتاب دهد، گرداننده *catch* بنویسید. پس چگونه مطمئن می‌شوید که برنامه‌های شما همه استثناهای ممکنه را گرفته و اداره می‌کنند؟

پاسخ به این پرسش به شیوه ای که استثناهای متفاوت به یکی دیگر مرتبط می‌شوند، تکیه می‌کند. استثناها در خانواده‌هایی با نام سلسله مراتب توارث سازماندهی می‌شوند. *FormatException* و *OverflowException* هردو به خانواده ای با نام *SystemException* تعلق دارند، همان طور که تعداد دیگری از استثناها این گونه هستند. به جای گرفتن هر یک از این استثناها به طور مجزا، شما می‌توانید یک گرداننده ایجاد کنید که *SystemException* را دریافت کند. *SystemException* در اصل عضوی از یک خانواده با نام *Exception* است. *Exception* نیای نهایی همه استثناهاست. در صورتی که *Exception* را دریافت کنید، گرداننده هر استثنا ممکنی را که می‌تواند رخ دهد به دام می‌اندازد.

توجه خانواده *Exception* حاوی تنوع گسترده ای از استثناهاست، که بسیاری از آنها نامزد استفاده توسط بخش‌های متنوعی از .NET Framework هستند. برخی از این اینها تا حدی مبهم هستند، اما هنوز هم سودمند است که بدانیم که چگونه آنها را دریافت کنیم.

مثال بعدی نشان می‌دهد که چگونه همه استثناهای امکان پذیر سیستم دریافت می‌شوند:

```
try
{
int leftHandSide = int.Parse(lhsOperand.Text);
int rightHandSide = int.Parse(rhsOperand.Text);
int answer = doCalculation(leftHandSide, rightHandSide);
result.Text = answer.ToString();
}
catch (Exception ex) // this is a general catch handler
{
//...
}
```

توضیح در صورتی که بخواهید *Exception* را دریافت کنید، شما در واقع می‌توانید نام آن را از گرداننده *catch* حذف کنید زیرا به استثنا پیش فرض پیش فرض شده است:

```
catch
{
// ...
}
```

اگر چه، این امر همواره توصیه نمی‌شود. شیء استثنا ارسال شده به گرداننده *catch* می‌تواند حاوی اطلاعاتی در خصوص استثنا باشد، که هنگام استفاده از این نسخه از ساختار *catch* قابل دسترس نمی‌باشد.

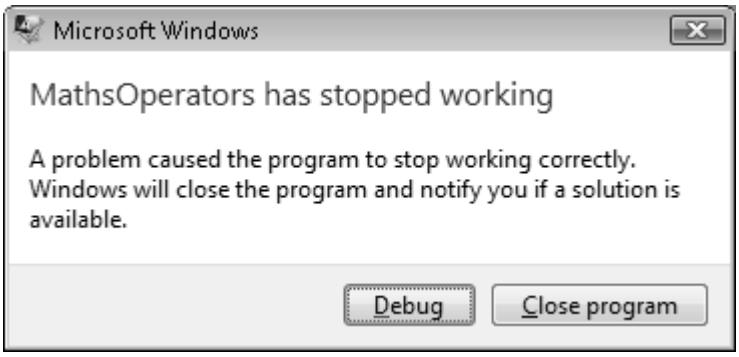
در اینجا یک پرسش پایانی وجود دارد که شما باید در این نقطه در حال پرسیدن آن باشید: در صورتی که استثنا یکسانی با گردانده‌های *catch* متعددی در انتهای یک بلوک *try* مطابقت کند، چه اتفاقی می‌افتد؟ اگر شما *FormatException* و *Exception* را به دو شیوه متفاوت دریافت نمایید، کدام یک اجرا خواهد شد (یا هر دو اجرا خواهند شد)؟

هرگاه یک استثنا رخ می‌دهد، اولین گرداننده یافت شده توسط runtime که با استثنا تطبیق می‌کند، به کار برده می‌شود و از استثناهای دیگر صرف نظر می‌شود. آن چه که این حرف معنا می‌دهد این است که اگر شما یک گرداننده را برای *Exception* قبل از یک گرداننده برای *FormatException* جای دهید، گرداننده *FormatException* هرگز اجرا نمی‌شود. از این رو، شما باید گردانده‌های *catch* خیلی خاص را بالای یک گرداننده *catch* عمومی‌بعد از یک بلوک *try* جای دهید. اگر هیچ یک از گردانده‌های *catch* ویژه با استثنا مطابقت نکنند، گرداننده *catch* عمومی‌اجر خواهد شد.

در فعالیت زیر، شما یک بلوک *try* خواهید نوشت و یک استثنا را دریافت خواهید کرد.

نوشتن یک دستور

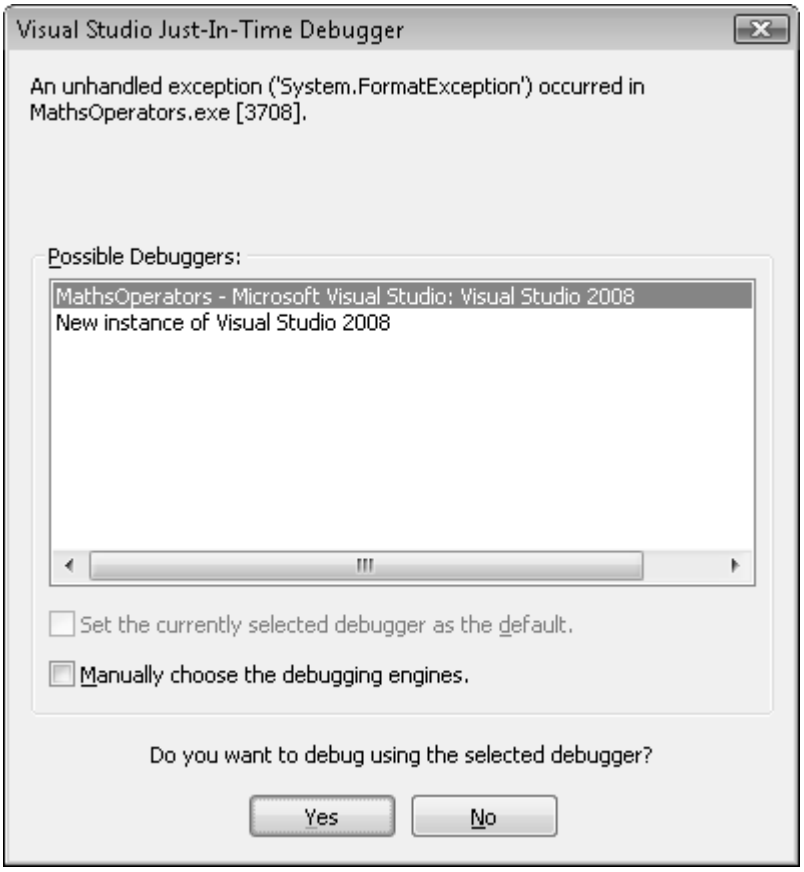
ویژوال استودیو ۲۰۰۸ را اگر از قبل در حال اجرا نباشد، شروع کنید.
محلول MathsOperators جای گرفته در پوشه \Microsoft Press\Visual CSharp Step By Step\Chapter 6\MathsOperators واقع در پوشه Documents خودتان را باز کنید.
این برنامه یک تغییر در مورد برنامه ای است که شما در ابتدا در فصل ۲، "کار با متغیرها، عملگرها و عبارات"، دیدید. این برنامه برای تشریح عملگرهای محاسباتی مختلف به کار برده شد.
در منوی *Debug*، گزینه *Start Without Debugging* را انتخاب کنید.
فرم ظاهر می‌شود. اکنون شما قصد دارید که برخی متون را وارد کنید که به طور عمدی در جعبه متن عملوند دست چپی معتبر نیستند. این عملیات فقدان نیرومندی در نسخه جاری برنامه را نشان خواهد داد.
عبارت **John** را در جعبه متن عملوند دست چپی تایپ کنید و سپس *Calculate* را کلیک کنید.
یک جعبه گفتگو یک استثنا اداره نشده را گزارش می‌شود؛ متنی که شما در جعبه متن عملوند دست چپی وارد کردید، باعث شده که برنامه نافرجام بماند.



توجه در صورتی که شما در حال استفاده کردن از Microsoft Visual C# 2008 Express Edition هستید، دکمه *Debug* ظاهر نمی‌شود.

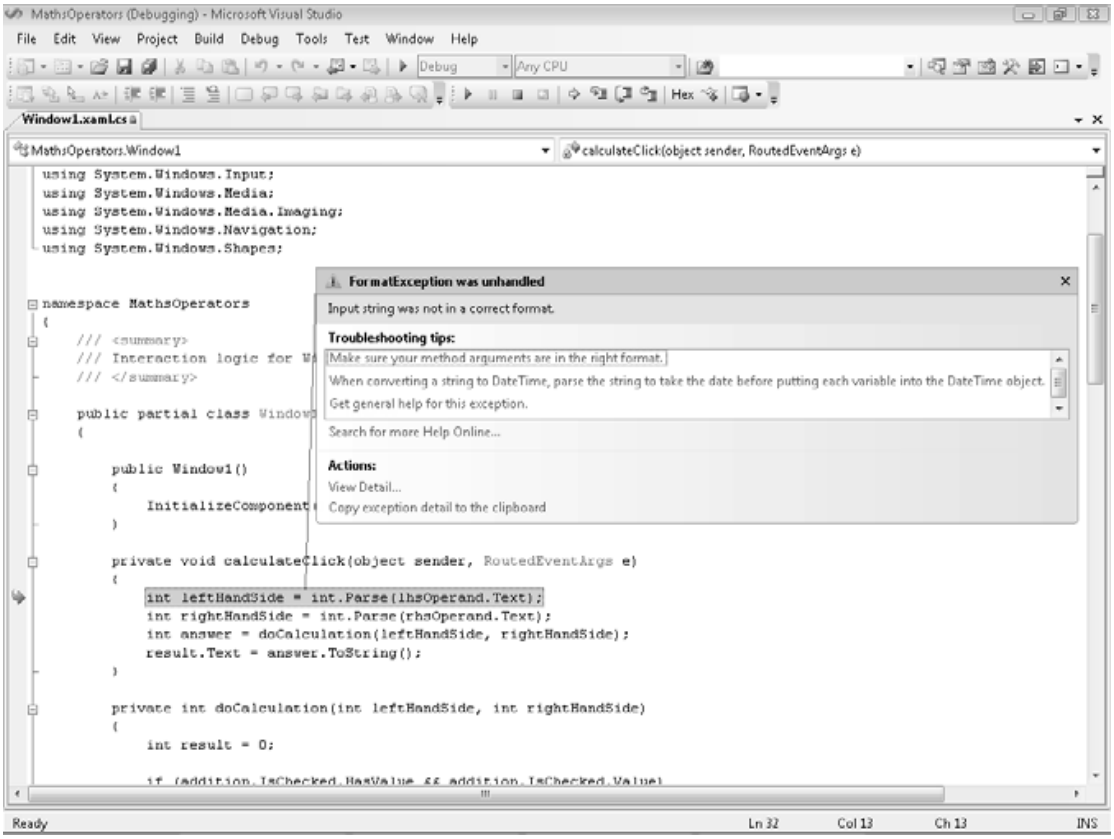
شما ممکن است نسخه متفاوتی از این جعبه گفتگو را (نشان داده شده در آینده) بسته به اینکه شما چگونه گزارش مشکل را در Control Panel پیکربندی کرده باشید، ببینید. اگر شما این جعبه گفتگو را ببینید، به سادگی هرگاه که دستورالعمل‌های واقع در گامهای زیر به دکمه *Close Program* اشاره می‌کنند پیوند *Close the program* را کلیک کنید و هرگاه که دستورالعمل‌ها به دکمه *Debug* اشاره می‌کنند پیوند *Debug the program* را کلیک کنید. (اگر در حال استفاده از ویندوز XP به جای ویندوز Vista هستید، شما یک جعبه گفتگوی متفاوت با دکمه‌های Debug، Send Error Report و Don't Send خواهید دید. دکمه Don't Send را برای بستن برنامه کلیک کنید.)

اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید، *Debug* را کلیک کنید. در جعبه گفتگوی *Visual Studio Just-In-Time Debugger*، در جعبه لیست *Possible Debuggers*، *MathsOperators- Microsoft Visual Studio: Visual Studio 2008* را انتخاب کنید و سپس *Yes* را کلیک کنید:



اگر در حال استفاده از Visual C# 2008 Express Edition هستید، *Close Program* را کلیک کنید. در منوی *Debug*، گزینه *Start Debugging* را کلیک کنید. **John** را در جعبه متن عملوند دست چپی تایپ کنید و سپس *Calculate* را کلیک کنید.

خواه در حال استفاده از ویژوال استودیو ۲۰۰۸ باشید یا از Visual C# 2008 Express Edition استفاده کنید، دیباگر ویژوال استودیو ۲۰۰۸ شروع می‌شود و سطر کدی را که باعث بروز استثنا شده پررنگ می‌کند و برخی اطلاعات اضافی را درباره استثنا نمایش می‌دهد:



شما می‌توانید ببینید که استثنا توسط فراخوان به *int.Parse* در میان متد *calculateClick* بازتابیده شده بود. مشکل این است که این متد قادر به تجزیه متن "John" به یک عدد معتبر نیست.

توجه شما تنها در صورتی که واقعاً کد منبع را در روی کامپیوتر در دسترس داشته باشید، می‌توانید کدی را که باعث بروز استثنا شده است، ببینید.

در منوی *Debug*، گزینه *Stop Debugging* را کلیک کنید.

کد مربوط به فایل Window1.xaml.cs را در پنجره ویرایشگر کد و متن نمایش دهید و متد *calculateClick* را پیدا کنید.

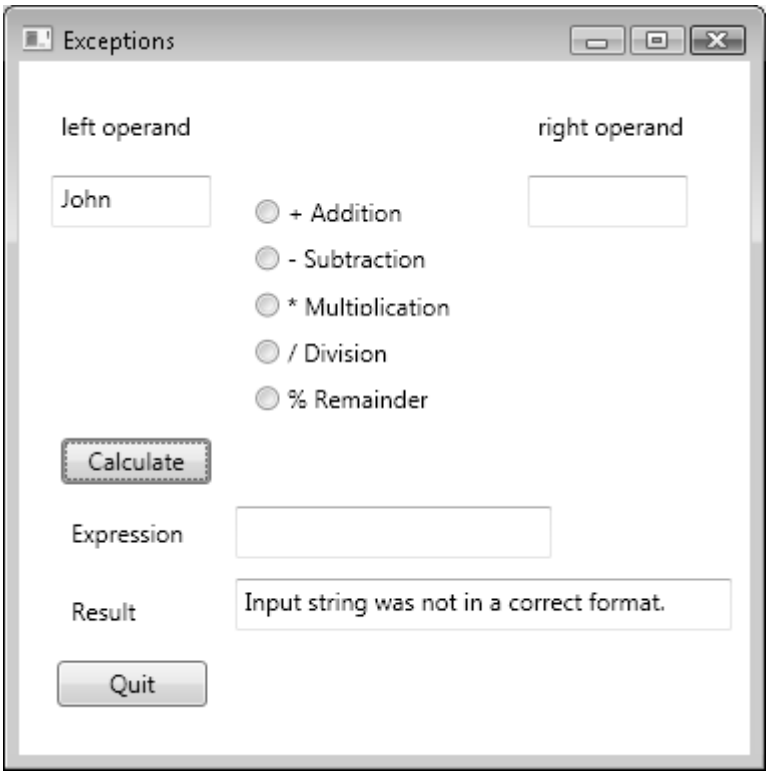
یک بلوک *try* (از جمله آکولادها) را حول چهار دستور داخل این متد اضافه کنید، همان طور که با فونت درشت در زیر نشان داده شده است:

```
try
{
int leftHandSide = int.Parse(lhsOperand.Text);
int rightHandSide = int.Parse(rhsOperand.Text);
int answer = doCalculation(leftHandSide, rightHandSide);
result.Text = answer.ToString();
}
```

یک بلوک *catch* بلافاصله بعد از آکولاد بسته مربوط به این بلوک *try* جدید اضافه کنید، همانند زیر:

```
catch (FormatException fEx)
{
}
```

این گرداننده *catch* استثنا *FormatException* بازتاب شده توسط *int.Parse* را می‌گیرد و سپس متن واقع در خاصیت در جعبه *Message* استثنا را در جعبه متن *result* در پایین فرم نمایش می‌دهد.
در منوی *Debug*، گزینه *Start Without Debugging* را کلیک کنید.
عبارت **John** را در جعبه متن عملوند چپی تایپ کنید و سپس *Calculate* را کلیک کنید.
گرداننده *catch* به طور موفقیت آمیزی *FormatException* را دریافت می‌کند و پیغام "Input string was not in a correct format" در جعبه متن *Result* نوشته می‌شود. اکنون برنامه‌اندکی نیرومند است.



John را با عدد 10 جایگزین کنید، عبارت **Sharp** را در جعبه متن عملوند دست راستی تایپ کنید و سپس *Calculate* را کلیک کنید.
توجه کنید که از آن جایی که بلوک *try* دستورات مربوط به هر دو جعبه متن را محصور می‌کند، گرداننده استثنا یکسان خطاهای وردوی کاربر را در مورد هر دو جعبه متن اداره می‌کند.
Quit را کلیک کنید تا به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ بازگردید.

استفاده از حساب اعداد صحیح بررسی شده و بررسی نشده

در فصل ۲، یادگرفتید که چگونه از عملگرهای محاسباتی باینری مانند + و * در روی انواع داده اصلی مانند *int* و *double* استفاده کنید. هم چنین شما دیدید که انواع داده اصلی اندازه ثابتی دارند. برای مثال، یک *int* متعلق به C# ۳۲ بیت دارد. از آن جایی که *int* اندازه ثابتی دارد، شما دقیقاً دامنه مقادیری را که *int* نگه می‌دارد می‌دانید: 2147483648– تا 2147483647 .

توضیح در صورتی که بخواهید به مقادیر حداکثر و حداقل *int* در میان کد اشاره کنید، شما می‌توانید از خاصیت *int.Min* یا *int.Max* استفاده کنید.

اندازه ثابت نوع *int* یک مشکل ایجاد می‌کند. برای مثال، در صورتی که شما 1 را به یک *int* که مقدارش اخیراً 2147483647 است، اضافه کنید چه اتفاقی می‌افتد؟ پاسخ این است که جواب بسته به این است که برنامه چگونه کامپایل شده است. به طور پیش فرض، کامپایلر C# کدی تولید می‌کند که اجازه می‌دهد تا محاسبه به طور آهسته سرریز کند. به عبارت دیگر، شما یک جواب غلط می‌گیرید. (در واقع، محاسبه پیرامون بزرگترین مقدار صحیح منفی می‌پیچد و نتیجه تولید شده برابر 2147483648– است.) علت این رفتار سطح کارایی است: حساب اعداد صحیح تقریباً یک عملیات متداول در هر برنامه ای است و اضافه کردن هزینه بررسی سرریز به هر عبارت صحیح می‌تواند منجر به کارایی خیلی ضعیفی شود. در بسیاری از موارد، این ریسک قابل قبول است زیرا شما می‌دانید (یا امیدوارید!) که مقادیر *int* به حدودشان نمی‌رسند. در صورتی که این رویکرد را دوست ندارید، می‌توانید بررسی سرریز را به جریان بیاندازید.

توضیح با تنظیم کردن خاصیت‌های پروژه می‌توانید بررسی سرریز را فعال یا غیرفعال کنید. در منوی *Project*، مورد *YourProject Properties* را کلیک کنید (جایی که *YourProject* اسم پروژه شماست). در جعبه گفتگوی خاصیت‌های پروژه، برگه *Build* را کلیک کنید. دکمه *Advanced* را در سمت راست پایینی صفحه کلیک کنید. در جعبه گفتگوی *Advanced Build Settings*، جعبه چک *Check for arithmetic overflow/underflow* را انتخاب یا پاک کنید.

قطع نظر از این که چگونه یک برنامه را کامپایل می‌کنید، شما می‌توانید از واژه‌های کلیدی *checked* و *unchecked* برای به جریان انداختن یا غیر فعال کردن بررسی سرریز حساب اعداد صحیح به طور انتخابی در بخش‌های یک برنامه که شما فکر می‌کنید که نیازمند آن هستید، استفاده کنید. این واژه‌های کلیدی گزینه کامپایلر تعیین شده برای پروژه را باطل می‌کنند.

نوشتن دستورات بررسی شده

یک دستور *checked* بلوکی است که قبل از آن واژه کلیدی *checked* می‌آید. همه محاسبات اعداد صحیح در یک بلوک *checked* در صورتی که یک محاسبه عدد صحیح در بلوک سرریز کند، همواره یک استثنا *OverflowException* بازتاب می‌دهد، همان طور که در این مثال نشان داده شده است:

```
int number = int.MaxValue;
checked
{
int willThrow = number++;
Console.WriteLine("this won't be reached");
}
```

مهم تنها محاسبات اعداد صحیحی که به طور مستقیم در میان بلوک *checked* جای دارند، در معرض بررسی سرریز هستند. برای مثال، در صورتی که یکی از دستورات *checked* یک فراخوانی متد باشد، بررسی سرریز به کدی که در متدی که فراخوانده شده است، اعمال نمی‌شود.

هم چنین شما می‌توانید از واژه کلیدی *unchecked* برای ایجاد یک بلوک دستور *unchecked* استفاده کنید. همه محاسبات واقع در یک بلوک *unchecked* بررسی نمی‌شود و هرگز یک *OverflowException* را بازتاب نمی‌دهد. برای مثال:

```
int number = int.MaxValue;
unchecked
{
int wontThrow = number++;
Console.WriteLine("this will be reached");
}
```

نوشتن عبارات بررسی شده

در ضمن شما می‌توانید از واژه‌های کلیدی *checked* و *unchecked* برای کنترل بررسی سرریز روی عبارات صحیح تنها با مقدم کردن واژه‌های کلیدی *checked* یا *unchecked* به عبارات محصور شده در میان پارانتزهای مجزا، استفاده کنید، همان طور که در این مثال نشان داده شده است:

```
int wontThrow = unchecked(int.MaxValue + 1);
int willThrow = checked(int.MaxValue + 1);
```

عملگرهای مرکب (مانند =+ و =-) و عملگرهای افزایش (++) و کاهش (- -) عملگرهای محاسباتی هستند و می‌توانند با استفاده از واژه‌های کلیدی *checked* و *unchecked* کنترل شوند. به یاد داشته باشید که x += y; همان x = x + y; است.

مهم شما نمی‌توانید از واژه‌های کلیدی *checked* و *unchecked* برای کنترل محاسبات ممیز شناور (غیر صحیح) استفاده کنید. واژه‌های کلیدی *checked* و *unchecked* هنگام استفاده از انواع داده ای مانند *int* و *long* تنها به محاسبات اعداد صحیح اعمال می‌شوند. محاسبات ممیز شناور هرگز استثنا *OverflowException* را بازتاب نمی‌کنند --- نه حتی زمانی که شما بر 0.0 تقسیم کنید. (یک تمثال برای بی نهایت دارد. .NET Framework)

در فعالیت زیر، شما چگونگی انجام دادن محاسبات بررسی شده را هنگام کار با ویژوال استودیو ۲۰۰۸ خواهید دید.

استفاده از عبارات بررسی شده

به ویژوال استودیو ۲۰۰۸ بازگردید.
در منوی *Debug*، مورد *Start Without Debugging* را انتخاب کنید.
اکنون شما مبادرت به ضرب کردن دو مقدار بزرگ خواهید کرد.
عدد **9876543** را در جعبه متن عملوند دست چپی و عدد **9876543** را در جعبه متن عملوند دست راستی تایپ کنید، مورد *Multiplication* را انتخاب کنید و سپس *Calculate* را کلیک کنید.

مقدار 1195595903– در جعبه متن *Result* در روی فرم ظاهر می‌شود. این عدد یک مقدار منفی است، که به هیچ وجه جواب درست نمی‌باشد. این مقدار نتیجه یک عملیات ضرب است که به طور بی‌صدایی از حد ۳۲-بیت نوع *int* سرریز شده است.
Quit را کلیک کنید و به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ بازگردید.
در پنجره ویرایشگر کد و متن که در حال نمایش Window1.xaml.cs است، متد *multiplyValues* را پیدا کنید. این متد مانند زیر به نظر می‌رسد:

```
private int multiplyValues(int leftHandSide, int rightHandSide)
{
expression.Text = leftHandSide.ToString() + " * " + rightHandSide.ToString();
return leftHandSide * rightHandSide;
}
```

دستور *return* حاوی عملیات ضرب است که به طور بی صدایی در حال سرریز است.
دستور *return* را ویرایش کنید به طوری که مقدار *return* بررسی شده شود، مانند این:

```
return checked(leftHandSide * rightHandSide);
```

عمل ضرب اکنون بررسی شده است و به جای این که به طور بی سر وصدایی جواب نادرست را برگرداند، یک استثنا *OverflowException* بازتاب خواهد کرد.
متد *calculateClick* را پیدا کنید.

گردانندهٔ *catch* زیر را بلافاصله بعد از گردانندهٔ *FormatException catch* در متد *calculateClick* اضافه کنید:

```
catch (OverflowException oEx)
{
result.Text = oEx.Message;
}
```

توضیح منطق این گردانندهٔ *catch* مشابه منطق گردانندهٔ *FormatException catch* است. اگرچه، هنوز هم ارزش این را دارد که این گردانندهها را به جای نوشتن یک گردانندهٔ *Exception catch* کلی، مجزا از هم نگه داشت زیرا شما ممکن است که بخواهید در آینده این استثناها را به طور متفاوتی اداره کنید.

در منوی *Debug*، مورد *Start Without Debugging* را کلیک کنید تا برنامه بنا شده و اجرا شود.
مقدار **9876543** را در جعبهٔ متن عملوند دست چپی و مقدار **9876543** را در جعبهٔ متن عملوند دست راستی تایپ کنید، مورد *Multiplication* را انتخاب کنید و سپس *Calculate* را کلیک کنید.
دومین گردانندهٔ *catch* با موفقیت *OverflowException* را به چنگ میآورد و پیغام "Arithmetic operation resulted in an overfl ow" را در جعبهٔ متن *Result* نمایش میدهد.
Quit را برای بازگشت به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ کلیک کنید.

بازتاباندن استثناها

تصور کنید که در حال پیاده سازی یک متد با نام *monthName* هستید که تنها یک آرگومان int را قبول میکند و اسم ماه متناظر را برمیگرداند. برای مثال، *monthName(1)* عبارت "January" را برمیگرداند، *monthName(2)* عبارت "February" را برمیگرداند و الی آخر. سؤال این است که: هنگامی که عدد صحیح کمتر از 1 یا بزرگتر از 12 باشد متد چه چیزی را باید برگرداند؟ بهترین پاسخ این است که متد به هیچ وجه نباید چیزی را برگرداند؛ متد باید باید یک استثنا را بازتاب دهد. کتابخانههای کلاس .NET Framework حاوی کلاسهای استثنا بسیار زیادی هستند که مخصوصاً برای وضعیتهایی مانند این طراحی شدهاند. اغلب اوقات، یکی از این کلاسها را پیدا خواهید کرد که شرایط استثنایی شما را توصیف میکنند. (اگر نه، شما میتوانید به راحتی کلاس استثنا مال خود را ایجاد کنید، اما قبل از این که بتوانید این کار را انجام دهید، نیازمند این هستید که اندکی بیشتر دربارهٔ زبان C# بدانید.) در این حالت، کلاس موجود *ArgumentOutOfRangeException* متعلق به .NET Framework. دقیقاً مناسب است. شما میتوانید یک استثنا را با استفاده از دستور *throw* بازتاب دهید، همان طور که در مثال زیر نشان داده شده است:

```
public static string monthName(int month)
{
switch (month)
```

```
{
:case 1
return "January";
:case 2
return "February";
...
:case 12
return "December";
:default
throw new ArgumentOutOfRangeException("Bad month");
}
}
```

دستور *throw* نیازمند یک شیء استثنا است تا بازتاب دهد. این شیء حاوی جزییات استثنا، از جمله هر پیغام خطا، است. این مثال از یک استثنا استفاده می‌کند که یک شیء *ArgumentOutOfRangeException* تازه ایجاد می‌کند. شیء با یک رشته مقداردهی می‌شود که خاصیت *Message* آن با استفاده از یک سازنده پر خواهد شد.

در فعالیت زیر، شما کدی را به پروژه MathsOperators به منظور بازتاباندن یک استثنا اضافه خواهید کرد.

بازتاباندن استثنا مال خود

به ویژوال استودیو ۲۰۰۸ بازگردید.

در منوی *Debug*، مورد *Start Without Debugging* را کلیک کنید.

مقدار **24** را در جعبه متن عملوند دست چپی و مقدار **36** را در جعبه متن عملوند دست راستی تایپ کنید و سپس *Calculate* را کلیک کنید.

مقدار 0 در جعبه متن *Result* ظاهر می‌شود. واقعیت این است که شما یک مورد عملگر را که بلافاصله مشهود باشد، انتخاب نکرده‌اید. نوشتن یک پیغام تشخیصی در جعبه متن *Result*، در این مورد، سودمند خواهد بود.

Quit را کلیک کنید تا به محیط برنامه نویسی ویژوال استودیو ۲۰۰۸ بازگردید.

در پنجره ویرایشگر کد و متن در حالی که Window1.xaml.cs را نمایش می‌دهد، متد *doCalculation* را پیدا کرده و بررسی کنید. این متد مانند زیر به نظر می‌رسد:

```
private int doCalculation(int leftHandSide, int rightHandSide) {
int result = 0;
if (addition.IsChecked.HasValue && addition.IsChecked.Value)
result = addValues(leftHandSide, rightHandSide);
else if (subtraction.IsChecked.HasValue && subtraction.IsChecked.Value)
result = subtractValues(leftHandSide, rightHandSide);
else if (multiplication.IsChecked.HasValue && multiplication.IsChecked.Value)
```

```
result = multiplyValues(leftHandSide, rightHandSide);
else if (division.IsChecked.HasValue && division.IsChecked.Value)
result = divideValues(leftHandSide, rightHandSide);
else if (remainder.IsChecked.HasValue && remainder.IsChecked.Value)
result = remainderValues(leftHandSide, rightHandSide);
return result;
}
```

فیلدهای *addition*، *subtraction*، *multiplication*، *division* و *remainder* دکمه‌های رادیویی هستند که در روی فرم ظاهر می‌شوند. هر دکمه رادیویی یک خاصیت با نام *IsChecked* دارد که نشان می‌دهد که آیا کاربر آن را انتخاب کرده است یا نه. خاصیت *IsChecked* یک مثال از یک مقدار *nullable* (قابل تهی شدن) است، که به معنای این است که این خاصیت یا می‌تواند حاوی یک مقدار معین باشد یا در یک حالت تعریف نشده باشد. (در باره مقادیر قابل تهی شدن در فصل ۸، "شناخت مقادیر و ارجاعات"، بیشتر خواهید آموخت.) خاصیت *IsChecked.HasValue* نشان می‌دهد که آیا دکمه رادیویی در یک حالت تعریف شده است، و اگر این گونه باشد، خاصیت *IsChecked.Value* نشان می‌دهد که این حالت چیست. خاصیت *IsChecked.Value* یک Boolean است که اگر دکمه رادیویی انتخاب شده باشد، مقدار *true* را دارد و در غیر این صورت مقدار *false* را دارد. دستور *if* آبشاری هر دکمه رادیویی را به نوبت بررسی می‌کند تا دریابد که کدام یک انتخاب شده است. (دکمه‌های رادیویی دو به دو ناسازگار هستند، از این رو کاربر خیلی باشد تنها می‌تواند یک دکمه رادیویی را انتخاب کند.) اگر هیچ یک از دکمه‌ها انتخاب نشده باشند، هیچ یک از دستورات *if* برابر *true* نخواهند شد و متغیر *result* با مقدار اولیه اش (0) باقی خواهد ماند. این متغیر مقداری را که توسط متد برگردانده می‌شود، نگه می‌دارد.

شما می‌توانید که مشکل را با استفاده از اضافه کردن یک دستور *else* بیشتر به آبشار *if-else* برای نوشتن یک پیغام به جعبه متن *result* واقع در روی فرم، حل کنید. گرچه، این راه حل ایده خوبی نیست زیرا در واقع منظور این متد بیرون دادن پیغام نیست. بهتر است که آشکارسازی و مخابره یک خطا را از به چنگ آوردن و اداره کردن آن خطا جدا کنیم.

دستور *else* دیگری را به فهرست دستورات *if-else* (بلافاصله قبل از دستور *return*) اضافه کنید و عیناً یک *InvalidOperationException* را بازتاب دهید، همانند زیر:

```
else
throw new InvalidOperationException("no operator selected");
```

در منوی *Debug*، مورد *Start Without Debugging* را برای بناکردن و اجرای برنامه کلیک کنید.

مقدار **24** را در جعبه متن عملوند دست چپی و مقدار **36** را در جعبه متن عملوند دست راستی تایپ کنید و *Calculate* را کلیک کنید.

یک جعبه گفتگوی استثنا ظاهر می‌شود. برنامه یک استثنا بازتابانده است، اما کد شما هنوز هم آن را به چنگ نیاورده است.

Close program را کلیک کنید.

برنامه خاتمه می‌یابد و شما به ویژوال استودیو ۲۰۰۸ بازمی‌گردید.

اکنون که یک دستور *throw* نوشته‌اید و بررسی کرده‌اید که او یک استثنا را باتابانده است، شما یک گرداننده *catch* برای اداره کردن این استثنا خواهید نوشت.

به چنگ آوردن استثنا مال خودتان

در پنجره ویرایشگر کد و متن که در حال نمایش Window1.xaml.cs است، متد *calculateClick* را پیدا کنید.

گرداننده *catch* زیر را بلافاصله پایین دو گرداننده *catch* موجود در متد *calculateClick* اضافه کنید:

```
catch (InvalidOperationException ioEx)
{
result.Text = ioEx.Message;
}
```

این کد *InvalidOperationException* را که هنگامی که هیچ دکمه رادیویی عملگری انتخاب نشده، بازتاب شده است، به چنگ می‌آورد.

در منوی *Debug*، مورد *Start Without Debugging* را کلیک کنید.

مقدار **24** را در جعبه متن عملوند دست چپی و مقدار **36** را در جعبه متن عملوند دست راستی تایپ کنید و *Calculate* را کلیک کنید.

پیغام "no operator selected" در جعبه متن *Result* ظاهر می‌شود.

Quit را کلیک کنید.

برنامه خیلی نیرومند تر از آنچه قبلاً بود، است. گرچه، استثناهای متعددی هنوز هم می‌توانند رخ دهند که که گرفته نخواهند شد و ممکن است باعث شوند که برنامه نافرجام بماند. برای مثال، اگر شما سعی کنید که تقسیم بر صفر کنید، یک *DivideByZeroException* اداره نشده بازتابیده خواهد شد. (تقسیم اعداد صحیح بر صفر برخلا تقسیم اعداد ممیز شناور بر صفر، یک استثنا را بازمی‌تاباند.) یک راه برای حل این مشکل این است که در هر صورت تعداد زیادی گرداننده *catch* در میان متد *calculateClick* نوشت. گرچه، راه حل بهتر این است که یک گرداننده *catch* کلی اضافه کرد که *Exception* را در انتهای فهرست گردانندههای *catch* می‌گیرد. این کار همه استثناهای اداره نشده را به دام می‌اندازد.

توضیح تصمیم گیری درباره این که آیا همه استثناهای اداره نشده را به طور صریح در یک متد به دام انداخت بستگی به طبیعت برنامه ای که در حال ساختنش هستید دارد. در برخی موارد، احساس می‌شود که استثناها تا جایی که ممکن است نزدیک جایی که آنها رخ داده‌اند، در اختیار گرفته شوند. در وضعیت‌های دیگر، این کار خیلی سودمند است که اجازه دهیم تا یک استثنا رو به عقب به متدی که روتینی را احظار کرده که آن روتین استثنا را بازتابانده است، منتشر شود.

به چنگ آوردن استثناهای اداره نشده

در پنجره ویرایشگر کد و متن که در حال نمایش Window1.xaml.cs است، متد *calculateClick* را پیدا کنید.
گرداننده *catch* زیر را به انتهای فهرست گرداننده‌های *catch* موجود اضافه کنید:

```
catch (Exception ex)
{
result.Text = ex.Message;
}
```

این گرداننده *catch*، همه استثناهای اداره نشده تا اینجا را، فارغ از نوع مخصوص آنها به چنگ خواهد آورد.
در منوی *Debug*، مورد *Start Without Debugging* را کلیک کنید.
اکنون سعی خواهید کرد که برخی محاسبات را انجام دهید که معلوم است که باعث بروز استثناهایی می‌شوند و تصدیق خواهید کرد که آنها همگی به درستی اداره می‌شوند.
مقدار **24** را در جعبه متن عملوند دست چپی و مقدار **36** را در جعبه متن عملوند دست راستی تایپ کنید و *Calculate* را کلیک کنید.
تصدیق کنید که پیغام تشخیصی "no operator selected" هنوز هم در جعبه متن *Result* ظاهر می‌شود. این پیغام توسط گرداننده *InvalidOperationException* تولید شده بود.
عبارت **John** را در جعبه متن عملوند دست چپی تایپ کنید و سپس *Calculate* را کلیک کنید.
تصدیق کنید که پیغام تشخیصی "Input string was not in a correct format" در جعبه متن *Result* ظاهر می‌شود. این پیغام توسط گرداننده *FormatException* تولید شده بود.
مقدار **24** را در جعبه متن عملوند دست چپی و مقدار **0** را در جعبه متن عملوند دست راستی تایپ کنید و دکمه رادیویی *Divide* را انتخاب کنید و سپس *Calculate* را کلیک کنید.
تصدیق کنید که پیغام تشخیصی "Attempted to divide by zero" در جعبه متن *Result* ظاهر می‌شود. این پیغام توسط گرداننده *Exception* تولید شده بود.
Quit را کلیک کنید.

استفاده از بلوک finally

مهم است که به یاد داشته باشیم که هنگامی که یک استثنا بازتابیده می‌شود، استثنا جریان اجرا را در میان برنامه تغییر می‌دهد. این حرف به معنای این است که شما نمی‌توانید تضمین کنید که یک دستور همواره هنگامی که دستور قبلی خاتمه می‌یابد اجرا می‌شود زیرا دستور قبلی ممکن است که یک استثنا را بازتابانده باشد. به مثال زیر نگاه کنید. خیلی راحت است که فرض کرد که فراخوان به *reader.Close* همواره اتفاق می‌افتد. بعد از همه این حرف‌ها، این درست است که در آن جا در کد:

```
TextReader reader = src.OpenText();
string line;
```

```
while ((line = reader.ReadLine()) != null)
{
source.Text += line + "\n";
}
reader.Close();
```

برخی اوقات در صورتی که یک دستور به خصوص اجرا نشود، مسئله ای نیست، اما در مورد بسیاری از وضعیت‌ها این امر می‌تواند یک مشکل بسیار بزرگی باشد. در صورتی که دستور منبعی را که در یک دستور پیشین اشغال شده شده بود، آزاد می‌کند، عدم موفقیت اجرای این دستور منجر می‌شود که این منبع نگه داشته شود. این مثال درست مانند حالتی است: در صورتی که فراخوان به *src.OpenText* موفقیت آمیز باشد، نیازمند یک منبع (یک دستگیره فایل) است و شما باید مطمئن شوید که *reader.Close* را برای آزاد کردن منبع فراخوان کرده‌اید. اگر این کار را نکرده باشید، به زودی یا بعداً دستگیره‌ها را تمام خواهید کرد و دیگر قادر نخواهید بود که فایل‌های بیشتری را باز کنید. (اگر دستگیره‌های فایل را خیلی ناچیز یافتید، در عوض به فکر اتصالات پایگاه داده باشید.)

شیوه ای که می‌توان مطمئن شد یک دستور همواره اجرا می‌شود، خواه یک استثنا بازتابیده شده باشد یا نه، این است که دستور را در میان یک بلوک *finally* نوشت. یک بلوک *finally* بلافاصله بعد از یک بلوک *try* یا بلافاصله بعد از آخرین گرداننده *catch* بعد از یک بلوک *try* رخ می‌دهد. مادامی که برنامه بلوک *try* مرتبط با یک بلوک *finally* را وارد کند، بلوک *finally* همواره اجرا خواهد شد، حتی اگر یک استثنا رخ دهد. اگر یک استثنا بازتابانده شود و به طور محلی در اختیار گرفته شود، نخست گرداننده استثنا اجرا می‌شود که با بلوک *finally* پی گرفته می‌شود. اگر استثنا به طور محلی در اختیار گرفته نشود (runtime باید در میان فهرست متدهای فراخواننده جستجو کند تا یک گرداننده را پیدا کند)، نخست بلوک *finally* اجرا می‌شود. در هر حالتی، بلوک *finally* همواره اجرا می‌شود.

راه حل مشکل *reader.Close* همانند زیر است:

```
TextReader reader = null;
try
{
reader = src.OpenText();
string line;
while ((line = reader.ReadLine()) != null)
{
source.Text += line + "\n";
}
}
finally
{
if (reader != null)
{
```

```
reader.Close();
}
}
```

حتی اگر یک استثنا صادر شده باشد، بلوک *finally* اطمینان می‌دهد که دستور *reader.Close* همواره اجرا می‌شود.

- اگر می‌خواهید که به فصل بعدی بروید
  ویژوال استودیو ۲۰۰۸ را در حال اجرا نگه دارید و به فصل ۷ مراجعه کنید.

- اگر می‌خواهید که هم اکنون از ویژوال استودیو ۲۰۰۸ خارج شوید
  در منوی *File*، گزینه *Exit* را کلیک کنید. اگر یک جعبه گفتگوی *Save* دیدید، *Yes* را کلیک کنید (اگر در حال استفاده از ویژوال استودیو ۲۰۰۸ هستید) یا *Save* را کلیک کنید (اگر درحال استفاده از Visual C# 2008 Express Edition هستید) و پروژه را ذخیره کنید.

فصل ۷

مقدمه‌ای بر کلاس‌ها و اشیاء

در فصول قبلی، یاد گرفتید که چگونه متغیرها را اعلان کنید، از عملگرها برای ایجاد مقادیر استفاده کنید، متدها را فراخوانی کنید و دستورات بسیاری را که هنگام پیاده‌سازی یک متد نیاز دارید، بنویسید. اکنون به‌اندازه کافی می‌دانید تا به مرحله بعدی پیشروی کنید--- با ترکیب کردن متدها و داده‌ها در میان کلاس‌های مال خودتان.

Microsoft .NET Framework حاوی هزاران کلاس است و شما تعدادی از آنها، از جمله *Console* و *Exception* را قبلاً به کار برده‌اید. کلاس‌ها یک مکانیزم سرراست برای مدل سازی هویت‌های دست‌کاری شده توسط برنامه‌ها عرضه می‌کنند. یک هویت (*entity*) یک آیتم به خصوص، مانند یک مشتری، یا یک چیزی مجرد تر، مانند یک معامله را بیان می‌کند. بخشی از فرایند طراحی هر سیستم با تعیین هویت‌هایی که مهم هستند سپس انجام دادن تحلیل‌هایی برای دیدن این که آنها چه اطلاعاتی را نیاز دارند که نگه دارند و چه اعمالی را باید انجام دهند، مرتبط است. شما اطلاعاتی را که یک کلاس نگه می‌دارد به صورت فیلدها ذخیره می‌کنید و از متدها برای پیاده‌سازی عملیات‌هایی که یک کلاس می‌تواند انجام دهد، استفاده کنید.

شناخت طبقه بندی (Classification)

Class کلمه ریشه‌ی عبارت *classification* (طبقه‌بندی) است. هـرگاه یک کلاس را طراحی می‌کنید، با روشی سیستماتیک اطلاعات را به میان یک هویت معنادار سازماندهی می‌کنید. این آرایش بندی یک عمل طبقه بندی است و چیزی است که هرکسی انجام می‌دهد--- نه فقط برنامه‌نویسان. برای مثال، همه ماشین‌ها رفتارهای مشترکی (آنها می‌توانند رانده شوند، متوقف شوند، شتاب‌دار شوند و ...) و خصوصیات مشترکی (آنها یک فرمان، یک موتور و ... دارند) را به اشتراک می‌گذارند. مردم از واژه ماشین برای معنا کردن اشیائی که این رفتارها و خصوصیات مشترک را به اشتراک می‌گذارند، استفاده می‌کنند. مادامی که هر کسی موافق این باشد که یک واژه چه معنایی می‌دهد، این سیستم به خوبی جواب می‌دهد؛ شما می‌توانید ایده‌های پیچیده اما صریح را در یک قالب موجز و مختصر بیان کنید. بدون طبقه بندی، خیلی سخت است که تصور کنیم که اصلاً مردم چگونه فکر می‌کنند یا ارتباط برقرار می‌کنند.

از آن جایی که طبقه بندی آن قدر در روشی که ما در آن فکر می‌کنیم و ارتباط برقرار می‌کنیم، ریشه دوانده است، خوب است که سعی کنیم با استفاده از طبقه بندی مفاهیم متفاوتی که در یک مسأله و راه حل آن ذاتی هستند، و سپس مدل سازی کردن این کلاس‌ها در یک زبان برنامه نویسی، برنامه‌ها را بنویسیم. این دقیقاً آن چیزی است که شما می‌خواهید با زبان‌های برنامه نویسی شیءگرای مدرن، مانند C# انجام دهید.

منظور از کپسوله کردن

کپسوله کردن (Encapsulation) یک مفهوم با اهمیت در زمان تعریف کردن کلاس‌هاست. ایده این است که یک برنامه که از یک کلاس استفاده می‌کند نباید نگران باشد که کلاس در واقع به طور درونی چگونه کار می‌کند. برنامه در اصل یک وهله از یک کلاس ایجاد می‌کند و متدهای آن وهله را فرا می‌خواند. مادامی که آن متدها آنچه را که آنها گفته‌اند انجام خواهند داد، انجام دهند، برنامه اهمیتی نمی‌دهد که آنها چگونه پیاده‌سازی می‌شوند. برای مثال، هنگامی که شما متد *Console.WriteLine* را فرامی خوانید، شما نمی‌خواهید که خود را با همه جزییات بغرنجی اذیت نمی‌کنید که چگونه کلاس *Console* به طور فیزیکی سازماندهی می‌شود تا داده‌ها به صفحه نوشته‌شوند. یک کلاس ممکن است نیازمند این باشد که از تمامی انواع اطلاعات وضعیت درونی برای انجام متدهای متعدد خود، حفاظت و پشتیبانی کند. این فعالیت و اطلاعات وضعیت اضافی از برنامه‌ای که در حال استفاده از کلاس است، مخفی است. از این رو، کپسوله کردن برخی اوقات به عنوان پنهان سازی اطلاعات اشاره می‌کند. کپسوله کردن در واقع دو هدف دارد:

- ترکیب متدها و داده‌ها در میان یک کلاس ؛ به عبارت دیگر، پشتیبانی طبقه بندی
- کنترل قابلیت دسترسی داده و متدها ؛ به عبارت دیگر، کنترل استفاده از کلاس

تعریف و استفاده از یک کلاس

در C#، شما واژه کلیدی *class* را برای تعریف یک کلاس جدید به کار می‌برید. داده‌ها و متدهای کلاس در بدنه کلاس در میان یک جفت آکولاد رخ می‌دهند. در اینجا یک کلاس C# با نام *Circle* آورده شده است که حاوی یک متد (برای محاسبه مساحت دایره) و یک قطعه داده (شعاع دایره) است:

```
class Circle
{
double Area()
{
return Math.PI * radius * radius;
}
int radius;
}
```

توجه کلاس *Math* دربردارنده متدهای برای انجام دادن محاسبات ریاضیاتی و فیلدهایی حاوی ثوابت ریاضیاتی است. فیلد *Math.PI* حاوی مقدار 3.14159265358979323846 است، که یک تقریب مقدار پی است.

بدنه یک کلاس حاوی متدها (مانند *Area*) و فیلدهای (مانند *radius*) معمولی است--- به یاد داشته باشید که متغیرها در یک کلاس فیلد خوانده می‌شوند. شما قبلاً در فصل ۲، دیده‌اید که چگونه متغیرها را اعلان کنید و در فصل ۳ دیده‌اید که چگونه متدها را بنویسید؛ در واقع تقریباً هیچ ترکیب نوشتاری جدیدی در اینجا وجود ندارد.

استفاده از کلاس *Circle* مشابه استفاده از انواع دیگری است که قبلاً با آنها آشنا شده‌اید؛ یک متغیر همراه با مشخص کردن *Circle* به عنوان نوع آن ایجاد می‌کنید و سپس متغیر را با داده معتبر معینی مقداردهی می‌کنید. در این جا یک مثال آورده شده است:

```
Circle c; // Create a Circle variable
c = new Circle(); // Initialize it
```

به کاربرد واژه کلیدی *new* توجه کنید. قبلاً، هرگاه شما یک متغیر مانند یک *int* یا *float* را مقداردهی کردید، شما در اصل یک مقدار به آن تخصیص می‌دهید:

```
int i;
i = 42;
```

شما نمی‌توانید همان کار را با متغیرهایی از انواع کلاس انجام دهید. یک دلیل این است که C# صرفاً ترکیب نوشتاری برای تخصیص مقادیر کلاس لیترال به متغیرها عرضه نمی‌کند. (معادل 42 از *Circle* چیست؟) دلیل دیگر با شیوه ای که در آن حافظه مربوط به متغیرهای انواع کلاس توسط runtime تخصیص داده شده و مدیریت می‌شود. برای این لحظه، صرفاً قبول کنید که واژه کنید *new* یک نمونه (وهله) از یک کلاس (که اغلب یک شیء خوانده می‌شود) ایجاد می‌کند.

گرچه، شما می‌توانید به طور مستقیم یک وهله از یک کلاس را به متغیر دیگری از همان نوع تخصیص دهید، مانند این:

```
Circle c;
c = new Circle();
Circle d;
d = c;
```

مهم میان عبارات شیء و کلاس سردرگم نشوید. یک کلاس تعریف یک نوع است. یک شیء، نمونه یا وهله‌ای از آن نوع است که هنگام اجرای برنامه ایجاد می‌شود.