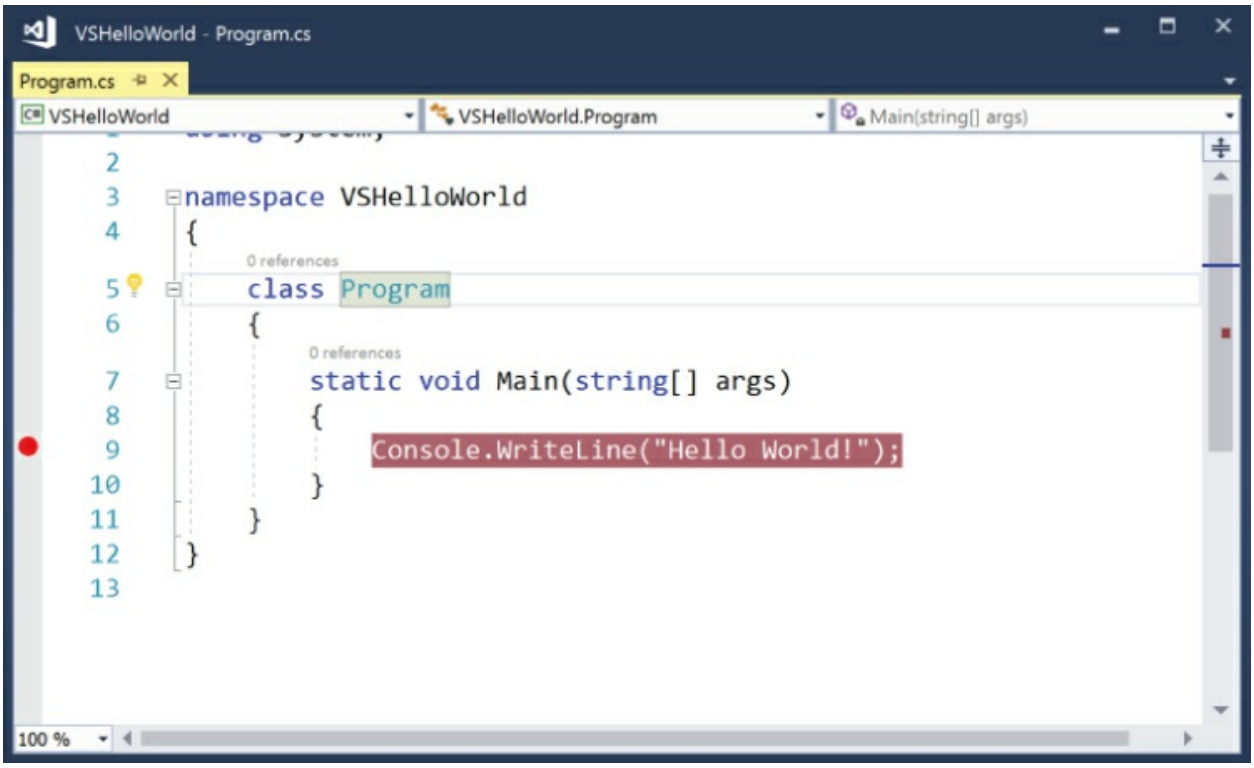

FIGURE 1-15

Now you've seen the parts of Visual Studio that are most important for helping you to survive the first chapters in this book. Chapter 18 takes a deeper look at Visual Studio 2017.

FIGURE 1-16

APPLICATION TYPES AND TECHNOLOGIES

You can use C# to create console applications; with most samples in the first chapters of this book you'll do that exact thing. For many programs, console applications are not used that often. You can use C# to create applications that use many of the technologies associated with .NET. This section gives you an overview of the different types of applications that you can write in C#.

Data Access

Before having a look at the application types, let's look at technologies that are used by all application types: access to data.

Files and directories can be accessed by using simple API calls; however, the simple API calls are not flexible enough for some scenarios. With the stream API you have a lot of flexibility, and the streams offer many more features such as encryption or compression. Readers and writers make using streams easier. All the different options available here are covered in Chapter 22, "Files and Streams." It's also possible to serialize complete objects in XML or JSON format. Bonus Chapter 2, "XML and JSON," (which you can find online) discusses these options.

To read and write to databases, you can use ADO.NET directly (see Chapter 25, "ADO.NET and Transactions"), or you can use an abstraction layer, Entity Framework Core (Chapter 26, "Entity Framework Core"). Entity Framework Core offers a mapping of object hierarchies to the relations of a database.

Entity Framework Core 1.0 is a complete redesign of Entity Framework, as is reflected with the new name. Code needs to be changed to migrate applications from older versions of Entity Framework to the new version. Older mapping variants, such as Database First and Model First, have been dropped, as Code First is a better alternative. The complete redesign was also done to support not only relational databases but also NoSQL. Entity Framework Core 2.0 has a long list of new features, which are covered in this book.

Windows Apps

For creating Windows apps, the technology of choice should be the Universal Windows Platform. Of course, there are restrictions when this option is not available—for example, if you still need to support older O/S versions like Windows 7. In this case you can use Windows Presentation Foundation (WPF). WPF is not covered in this book, but you can read the previous edition, *Professional C# 6 and .NET Core 1.0,* which has five chapters dedicated to WPF, plus some additional WPF coverage in other chapters.

This book has one focus: developing apps with the Universal Windows

Platform (UWP). Compared to WPF, UWP offers a more modern XAML to create the user interface. For example, data binding offers a compiled binding variant where you get errors at compile time instead of not showing the bound data. The application is compiled to native code before it's run on the client systems. And it offers a modern design, which is now called Fluent Design from Microsoft.

> **NOTE**
>
> *Creating UWP apps is covered in Chapter 33, "Windows Apps," along with an introduction to XAML, the different XAML controls, and the lifetime of apps. You can create apps with WPF, UWP, and Xamarin by using as much common code as possible by supporting the MVVM pattern. This pattern is covered in Chapter 34, "Patterns with XAML Apps." To create cool looks and style the app, be sure to read Chapter 35, "Styling Windows Apps." Chapter 36, "Advanced Windows Apps," dives into some advanced features of UWP.*

Xamarin

It would have been great if Windows had been a bigger player in the mobile phone market. Then Universal Windows Apps would run on the mobile phones as well. Reality turned out differently, and Windows on the phone is (currently) a thing of the past. However, with Xamarin you can use C# and XAML to create apps on the iPhone and Android. Xamarin offers APIs to create apps on Android and libraries to create apps on iPhone—using the C# code you are used to.

With Android, a mapping layer using Android Callable Wrappers (ACW) and Managed Callable Wrappers (MCW) are used to interop between .NET code and Android's Java runtime. With iOS, an Ahead of Time (AOT) compiler compiles the managed code to native code.

Xamarin.Forms offers XAML code to create the user interface and share as much of the user interface as possible between Android, iOS,

Windows, and Linux. XAML only offers UI controls that can be mapped to all platforms. For using specific controls from a platform, you can create platform-specific renderers.

> **NOTE**
>
> *Developing with Xamarin and Xamarin.Forms is covered in Chapter 37, "Xamarin.Forms."*

Web Applications

The original introduction of ASP.NET fundamentally changed the web programming model. ASP.NET Core changed it again. ASP.NET Core allows the use of .NET Core for high performance and scalability, and it not only runs on Windows but also on Linux systems.

With ASP.NET Core, ASP.NET Web Forms is no longer covered (ASP.NET Web Forms can still be used and is updated with .NET 4.7).

ASP.NET Core MVC is based on the well-known Model-View-Controller (MVC) pattern for easier unit testing. It also allows a clear separation for writing user interface code with HTML, CSS, and JavaScript, and it uses C# on the backend.

> **NOTE**
>
> *Chapter 30 covers the foundation of ASP.NET Core. Chapter 31 continues building on the foundation and adds using the ASP.NET Core MVC framework.*

Web API

SOAP and WCF fulfilled their duty in the past, and they're not needed anymore. Modern apps make use of REST (Representational State

Transfer) and the Web API. Using ASP.NET Core to create a Web API is an option that is a lot easier for communication and fulfills more than 90 percent of requirements by distributed applications. This technology is based on REST, which defines guidelines and best practices for stateless and scalable web services.

The client can receive JSON or XML data. JSON and XML can also be formatted in a way to make use of the Open Data specification (OData).

The features of this new API make it easy to consume from web clients using JavaScript, the Universal Windows Platform, and Xamarin.

Creating a Web API is a good approach for creating microservices. The approach to build microservices defines smaller services that can run and be deployed independently, having their own control of a data store.

To describe the services, a new standard was defined: the OpenAPI (`https://www.openapis.org`). This standard has its roots with Swagger (`https://swagger.io/`).

> **NOTE**
>
> *The ASP.NET Core Web API, Swagger, and more information on microservices are covered in Chapter 32.*

WebHooks and SignalR

For real-time web functionality and bidirectional communication between the client and the server, WebHooks and SignalR are ASP.NET Core technologies available with .NET Core 2.1.

SignalR allows pushing information to connected clients as soon as information is available. SignalR makes use of the WebSocket technology to push information.

WebHooks allows you to integrate with public services, and these services can call into your public ASP.NET Core created Web API

service. WebHooks is a technology to receive push notification from services such as GitHub or Dropbox and many other services.

> **NOTE**
>
> *The foundation of SignalR connection management, grouping of connections, and authorization and integration of WebHooks are discussed in Bonus Chapter 3, "WebHooks and SignalR," which you can find online.*

Microsoft Azure

Nowadays you can't ignore the cloud when considering the development picture. Although there's not a dedicated chapter on cloud technologies, Microsoft Azure is referenced in several chapters in this book.

Microsoft Azure offers Software as a Service (SaaS), Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Functions as a Service (FaaS), and sometimes offerings are in between these categories. Let's have a look at some Microsoft Azure offerings.

Software as a Service

SaaS offers complete software; you don't have to deal with management of servers, updates, and so on. Office 365 is one of the SaaS offerings for using e-mail and other services via a cloud offering. A SaaS offering that's relevant for developers is *Visual Studio Team Services*. Visual Studio Team Services is the Team Foundation Server in the cloud that can be used as a private code repository, for tracking bugs and work items, and for build and testing services. Chapter 18 explains DevOps features that can be used from Visual Studio.

Infrastructure as a Service

Another service offering is *IaaS*. Virtual machines are offered by this service offering. You are responsible for managing the operating

system and maintaining updates. When you create virtual machines, you can decide between different hardware offerings starting with shared Cores up to 128 cores (at the time of this writing, but things change quickly). 128 cores, 2 TB RAM, and 4 TB local SSD belong to the “M-Series” of machines.

With preinstalled operating systems you can decide between Windows, Windows Server, Linux, and operating systems that come preinstalled with SQL Server, BizTalk Server, SharePoint, and Oracle, and many other products.

I use virtual machines often for environments that I need only for several hours a week, as the virtual machines are paid on an hourly basis. In case you want to try compiling and running .NET Core programs on Linux but don’t have a Linux machine, installing such an environment on Microsoft Azure is an easy task.

Platform as a Service

For developers, the most relevant part of Microsoft Azure is PaaS. You can access services for storing and reading data, use computing and networking capabilities of app services, and integrate developer services within the application.

For storing data in the cloud, you can use a relational data store SQL Database. SQL Database is nearly the same as the on-premise version of SQL Server. There are also some NoSQL solutions such as Cosmos DB with different store options like JSON data, relationships, or table storage, and Azure Storage that stores blobs (for example, for images or videos).

App Services can be used to host your web apps and API apps that you are creating with ASP.NET Core.

Microsoft also offers Developer Services in Microsoft Azure. Part of the Developer Services is Visual Studio Team Services. Visual Studio Team Services allows you to manage the source code, automatic builds, tests, and deployments—continuous integration (CI).

Part of the Developer Services is Application Insights. With faster release cycles, it’s becoming more and more important to get

information about how the user uses the app. What menus are never used because the users probably don't find them? What paths in the app is the user taking to fulfill his or her tasks? With Application Insights, you can get good anonymous user information to find out the issues users have with the application, and with DevOps in place you can do quick fixes.

You also can use *Cognitive Services* that offer functionality to process images, use Bing Search APIs, understand what users say with language services, and more.

Functions as a Service

FaaS is a new concept for cloud service, also known as a *serverless computing* technology. Of course, behind the scenes there's always a server. You just don't pay for reserved CPU and memory as you do with App Services that are used from web apps. Instead the amount you pay is based on consumption—on the number of calls done with some limitations on the memory and time needed for the activity. Azure Functions is one technology that can be deployed using FaaS.

> **NOTE**
>
> *In Chapter 29, "Tracing, Logging, and Analytics," you can read about tracing features and learn how to use the Application Insights offering of Microsoft Azure. Chapter 32, "Web API," not only covers creating Web APIs with ASP.NET Core MVC but also shows how the same service functionality can be used from an Azure Function. The Microsoft Bot service as well as Cognitive Services are explained in Bonus Chapter 4, "Bot Framework and Cognitive Services," which you can find online.*

DEVELOPER TOOLS

This final part of the chapter, before we switch to a lot of C# code in the next chapter, covers developer tools and editions of Visual Studio

2017.

Visual Studio Community

This edition of Visual Studio is a free edition with features that the Professional edition previously had. There's a license restriction for when it can be used. It's free for open-source projects and training and to academic and small professional teams. Unlike the Express editions of Visual Studio that previously have been the free editions, this product allows using extensions with Visual Studio.

Visual Studio Professional

This edition includes more features than the Community edition, such as the CodeLens and Team Foundation Server for source code management and team collaboration. With this edition, you also get an MSDN subscription that includes several server products from Microsoft for development and testing.

Visual Studio Enterprise

Unlike the Professional edition, this edition contains a lot of tools for testing, such as Web Load & Performance Testing, Unit Test Isolation with Microsoft Fakes, and Coded UI Testing. (Unit testing is part of all Visual Studio editions.) With Code Clone you can find code clones in your solution. Visual Studio Enterprise also contains architecture and modeling tools to analyze and validate the solution architecture.

> **NOTE**
>
> *Be aware that with a Visual Studio subscription you're entitled to free use of Microsoft Azure up to a specific monthly amount that is contingent on the type of Visual Studio subscription you have.*

> **NOTE**
>
> *Chapter 18 includes details on using several features of Visual Studio 2017. Chapter 28, "Testing," gets into details of unit testing, web testing, and creating Coded UI tests.*

> **NOTE**
>
> *For some of the features in the book—for example, the Coded UI Tests —you need Visual Studio Enterprise. You can work through most parts of the book with the Visual Studio Community edition.*

Visual Studio for Mac

Visual Studio for Mac originates in the Xamarin Studio, but now it offers a lot more than the earlier product. For example, the editor shares code with Visual Studio, so you're soon familiar with it. With Visual Studio for Mac you can not only create Xamarin apps, but you also can create ASP.NET Core apps that run on Windows, Linux, and the Mac. With many chapters of this book, you can use Visual Studio for Mac. Exceptions are the chapters covering the Universal Windows Platform, which requires Windows to run the app and also to develop the app.

Visual Studio Code

Visual Studio Code is a completely different development tool compared to the other Visual Studio editions. While Visual Studio 2017 offers project-based features with a rich set of templates and tools, Visual Studio is a code editor with little project management support. However, Visual Studio Code runs not only on Windows, but also on Linux and OS X.

With many chapters of this book, you can use Visual Studio Code as your development editor. What you can't do is create UWP and Xamarin applications, and you also don't have access to the features covered in Chapter 18, "Visual Studio 2017." You can use Visual Studio Code for .NET Core console applications, and ASP.NET Core 1.0 web applications using .NET Core.

You can download Visual Studio Code from `http://code.visualstudio.com`.

SUMMARY

This chapter covered a lot of ground to review important technologies and changes with technologies. Knowing about the history of some technologies helps you decide which technology should be used with new applications and what you should do with existing applications.

You read about the differences between .NET Framework and .NET Core, and you saw how to create and run a Hello World application with all these environments with and without using Visual Studio.

You've seen the functions of the Common Language Runtime (CLR) and looked at technologies for accessing the database and creating Windows apps. You also reviewed the advantages of ASP.NET Core.

Chapter 2 dives fast into the syntax of C#. You learn variables, implement program flows, organize your code into namespaces, and more.

2
Core C#

WHAT'S IN THIS CHAPTER?

- Declaring variables
- Initialization and scope of variables
- Working with redefined C# data types
- Dictating execution flow within a C# program
- Organizing classes and types with namespaces
- Getting to know the `Main` method
- Using internal comments and documentation features
- Using preprocessor directives
- Understanding guidelines and conventions for good programming in C#

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The Wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `CoreCSharp`.

The code for this chapter is divided into the following major examples:

- HelloWorldApp
- VariablesSample
- VariableScopeSample
- IfStatement
- ForLoop
- NamespacesSample
- ArgumentsSample
- StringSample

FUNDAMENTALS OF C#

Now that you understand more about what C# can do, you need to know how to use it. This chapter gives you a good start in that direction by providing a basic understanding of the fundamentals of C# programming, which is built on in subsequent chapters. By the end of this chapter, you will know enough C# to write simple programs (though without using inheritance or other object-oriented features, which are covered in later chapters).

Hello, World!

Chapter 1, ".NET Application Architectures and Tools," shows how to create a Hello, World! application using the .NET Core CLI tools, Visual Studio, Visual Studio for Mac, and Visual Studio Code. Now let's concentrate on the C# source code. First, I have a few general comments about C# syntax. In C#, as in other C-style languages, statements end in a semicolon (`;`) and can continue over multiple lines without needing a continuation character. Statements can be joined into blocks using curly braces (`{}`). Single-line comments begin with two forward slash characters (`//`), and multiline comments begin with a slash and an asterisk (`/*`) and end with the same combination reversed (`*/`). In these aspects, C# is identical to C++ and Java but different from Visual Basic. It is the semicolons and curly braces that give C# code such a different visual appearance from Visual Basic

code. If your background is predominantly Visual Basic, take extra care to remember the semicolon at the end of every statement. Omitting this is usually the biggest single cause of compilation errors among developers who are new to C-style languages. Another thing to remember is that C# is case sensitive. That means the variables named `myVar` and `MyVar` are two different variables.

The first few lines in the previous code example are related to *namespaces* (mentioned later in this chapter), which is a way to group associated classes. The `namespace` keyword declares the namespace with which your class should be associated. All code within the braces that follow it is regarded as being within that namespace. The `using` declaration specifies a namespace that the compiler should look at to find any classes that are referenced in your code but aren't defined in the current namespace. This serves the same purpose as the `import` statement in Java and the `using namespace` statement in C++ (code file `HelloWorldApp/Program.cs`):

```
using System;
namespace Wrox.HelloWorldApp
{
```

The reason for the presence of the `using System;` declaration in the `Program.cs` file is that you are going to use the class `Console` from the namespace `System`: `System.Console`. The `using System;` declaration enables you to refer to this class without adding the namespace. You can invoke the `WriteLine` method using the following class:

```
using System;
// ...
Console.WriteLine("Hello World!");
```

NOTE

Namespaces are explained in detail later in this chapter in the section "Getting Organized with Namespaces."

With the `using static` declaration you can open not only a namespace,

but all static members of a class. Declaring `using static System.Console`, you can invoke the `WriteLine` method of the `Console` class without the class name:

```
using static System.Console;
// ...
WriteLine("Hello World!");
```

Omitting the complete `using` declaration, you need to add the namespace name invoking the `WriteLine` method:

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

The standard `System` namespace is where the most commonly used .NET types reside. It is important to realize that everything you do in C# depends on .NET base classes. In this case, you are using the `Console` class within the `System` namespace to write to the console window. C# has no built-in keywords of its own for input or output; it is completely reliant on the .NET classes.

Within the source code, a class called `Program` is declared. However, because it has been placed in a namespace called `Wrox.HelloWorldApp`, the fully qualified name of this class is `Wrox.HelloWorldApp.Program` (code file `HelloWorldApp/Program.cs`):

```
namespace Wrox.HelloWorldApp
{
 class Program
 {
```

All C# code must be contained within a class. The class declaration consists of the `class` keyword, followed by the class name and a pair of curly braces. All code associated with the class should be placed between these braces.

The class `Program` contains a method called `Main`. Every C# executable (such as console applications, Windows applications, Windows services, and web applications) must have an entry point—the `Main` method (note the capital `M`).

```
static void Main()
{
```

The method is called when the program is started. This method must return either nothing (`void`) or an integer (`int`). Note the format of method definitions in C#:

```
[modifiers] return_type MethodName([parameters])
{
 // Method body. NB. This code block is pseudo-code.
}
```

Here, the first square brackets represent certain optional keywords. Modifiers are used to specify certain features of the method you are defining, such as from where the method can be called. In this case the `Main` method doesn't have a public access modifier applied. You can do this in case you need a unit test for the `Main` method. The runtime doesn't need the public access modifier applied, and it still can invoke the method. The static modifier is required as the runtime invokes the method without creating an instance of the class. The return type is set to `void`, and in the example parameters are not included.

Finally, we come to the code statement themselves:

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

In this case, you simply call the `WriteLine` method of the `System.Console` class to write a line of text to the console window. `WriteLine` is a `static` method, so you don't need to instantiate a `Console` object before calling it.

Now that you have had a taste of basic C# syntax, you are ready for more detail. Because it is virtually impossible to write any nontrivial program without *variables*, we start by looking at variables in C#.

WORKING WITH VARIABLES

You declare variables in C# using the following syntax:

```
datatype identifier;
```

For example:

```
int i;
```

This statement declares an `int` named `i`. The compiler won't actually let you use this variable in an expression until you have initialized it with a value.

After it has been declared, you can assign a value to the variable using the assignment operator, =:

```
i = 10;
```

You can also declare the variable and initialize its value at the same time:

```
int i = 10;
```

If you declare and initialize more than one variable in a single statement, all the variables will be of the same data type:

```
int x = 10, y =20; // x and y are both ints
```

To declare variables of different types, you need to use separate statements. You cannot assign different data types within a multiple-variable declaration:

```
int x = 10;
bool y = true; // Creates a variable that stores true or
false
int x = 10, bool y = true; // This won't compile!
```

Notice the `//` and the text after it in the preceding examples. These are comments. The `//` character sequence tells the compiler to ignore the text that follows on this line because it is included for a human to better understand the program; it's not part of the program itself. Comments are explained further later in this chapter in the "Using Comments" section.

Initializing Variables

Variable initialization demonstrates an example of C#'s emphasis on safety. Briefly, the C# compiler requires that any variable be initialized with some starting value before you refer to that variable in an operation. Most modern compilers will flag violations of this as a warning, but the ever-vigilant C# compiler treats such violations as

errors.

C# has two methods for ensuring that variables are initialized before use:

- Variables that are fields in a class or struct, if not initialized explicitly, are by default zeroed out when they are created (classes and structs are discussed later).
- Variables that are local to a method must be explicitly initialized in your code prior to any statements in which their values are used. In this case, the initialization doesn't have to happen when the variable is declared, but the compiler checks all possible paths through the method and flags an error if it detects any possibility of the value of a local variable being used before it is initialized.

For example, you can't do the following in C#:

```
static int Main()
{
 int d;
 Console.WriteLine(d); // Can't do this! Need to initialize d
before use
 return 0;
}
```

Notice that this code snippet demonstrates defining `Main` so that it returns an `int` instead of `void`.

If you attempt to compile the preceding lines, you receive this error message:

```
Use of unassigned local variable 'd'
```

Consider the following statement:

```
Something objSomething;
```

In C#, this line of code would create only a *reference* for a `Something` object, but this reference would not yet actually refer to any object. Any attempt to call a method or property against this variable would result in an error.

To instantiate a reference object in C#, you must use the `new` keyword.

You create a reference as shown in the previous example and then point the reference at an object allocated on the heap using the `new` keyword:

```
objSomething = new Something(); // This creates a Something
object on the heap
```

Using Type Inference

Type inference makes use of the `var` keyword. The syntax for declaring the variable changes by using the `var` keyword instead of the real type. The compiler "infers" what the type of the variable is by what the variable is initialized to. For example:

```
var someNumber = 0;
```

becomes:

```
int someNumber = 0;
```

Even though `someNumber` is never declared as being an `int`, the compiler figures this out and `someNumber` is an `int` for as long as it is in scope. Once compiled, the two preceding statements are equal.

Here is a short program to demonstrate (code file `VariablesSample/Program.cs`):

```
using System;
namespace Wrox
{
  class Program
  {
   static void Main()
   {
    var name = "Bugs Bunny";
    var age = 25;
    var isRabbit = true;
    Type nameType = name.GetType();
    Type ageType = age.GetType();
    Type isRabbitType = isRabbit.GetType();
    Console.WriteLine($"name is of type {nameType}");
    Console.WriteLine($"age is of type {ageType}");
    Console.WriteLine($"isRabbit is of type {isRabbitType}");
   }
  }
```

```
}
```

The output from this program is as follows:

```
name is of type System.String
age is of type System.Int32
isRabbit is of type System.Boolean
```

There are a few rules that you need to follow:

- The variable must be initialized. Otherwise, the compiler doesn't have anything from which to infer the type.
- The initializer cannot be null.
- The initializer must be an expression.
- You can't set the initializer to an object unless you create a new object in the initializer.

Chapter 3, "Objects and Types," examines these rules more closely in the discussion of anonymous types.

After the variable has been declared and the type inferred, the variable's type cannot be changed. When established, the variable's type strong typing rules that any assignment to this variable must follow the inferred type.

Understanding Variable Scope

The *scope* of a variable is the region of code from which the variable can be accessed. In general, the scope is determined by the following rules:

- A *field* (also known as a member variable) of a class is in scope for as long as a local variable of this type is in scope.
- A *local variable* is in scope until a closing brace indicates the end of the block statement or method in which it was declared.
- A local variable that is declared in a `for`, `while`, or similar statement is in scope in the body of that loop.

Scope Clashes for Local Variables

It's common in a large program to use the same variable name for different variables in different parts of the program. This is fine as long as the variables are scoped to completely different parts of the program so that there is no possibility for ambiguity. However, bear in mind that local variables with the same name can't be declared twice in the same scope. For example, you can't do this:

```
int x = 20;
// some more code
int x = 30;
```

Consider the following code sample (code file `VariableScopeSample/Program.cs`):

```
using System;
namespace VariableScopeSample
{
  class Program
  {
    static int Main()
    {
      for (int i = 0; i < 10; i++)
      {
        Console.WriteLine(i);
      } // i goes out of scope here

      // We can declare a variable named i again, because
      // there's no other variable with that name in scope
      for (int i = 9; i >= 0; i –)
      {
        Console.WriteLine(i);
      } // i goes out of scope here.

      return 0;
    }
  }
}
```

This code simply prints out the numbers from 0 to 9, and then back again from 9 to 0, using two `for` loops. The important thing to note is that you declare the variable `i` twice in this code, within the same method. You can do this because `i` is declared in two separate loops, so each `i` variable is local to its own loop.

Here's another example (code file `VariableScopeSample2/Program.cs`):

```
static int Main()
{
  int j = 20;
  for (int i = 0; i < 10; i++)
  {
    int j = 30; // Can't do this – j is still in scope
    Console.WriteLine(j + i);
  }
  return 0;
}
```

If you try to compile this, you'll get an error like the following:

```
error CS0136: A local variable named 'j' cannot be declared
in
this scope because that name is used in an enclosing local
scope
to define a local or parameter
```

This occurs because the variable `j`, which is defined before the start of the `for` loop, is still in scope within the `for` loop and won't go out of scope until the `Main` method has finished executing. Although the second `j` (the illegal one) is in the loop's scope, that scope is nested within the `Main` method's scope. The compiler has no way to distinguish between these two variables, so it won't allow the second one to be declared.

Scope Clashes for Fields and Local Variables

In certain circumstances, however, you can distinguish between two identifiers with the same name (although not the same fully qualified name) and the same scope, and in this case the compiler allows you to declare the second variable. That's because C# makes a fundamental distinction between variables that are declared at the type level (fields) and variables that are declared within methods (local variables).

Consider the following code snippet (code file `VariableScopeSample3/Program.cs`):

```
using System;
namespace Wrox
{
  class Program
```

```
  {
    static int j = 20;
    static void Main()
    {
      int j = 30;
      Console.WriteLine(j);
      return;
    }
  }
}
```

This code will compile even though you have two variables named `j` in scope within the `Main` method: the `j` that was defined at the class level and doesn't go out of scope until the class `Program` is destroyed (when the `Main` method terminates and the program ends), and the `j` defined within `Main`. In this case, the new variable named `j` that you declare in the `Main` method *hides* the class-level variable with the same name, so when you run this code, the number `30` is displayed.

What if you want to refer to the class-level variable? You can actually refer to fields of a class or struct from outside the object, using the syntax `object.fieldname`. In the previous example, you are accessing a static field (you find out what this means in the next section) from a static method, so you can't use an instance of the class; you just use the name of the class itself:

```
// ...
static void Main()
{
 int j = 30;
 Console.WriteLine(j);
 Console.WriteLine(Program.j);
}
// ...
```

If you are accessing an instance field (a field that belongs to a specific instance of the class), you need to use the `this` keyword instead.

Working with Constants

As the name implies, a constant is a variable whose value cannot be changed throughout its lifetime. Prefixing a variable with the `const` keyword when it is declared and initialized designates that variable as

a constant:

```
const int a = 100; // This value cannot be changed.
```

Constants have the following characteristics:

- They must be initialized when they are declared. After a value has been assigned, it can never be overwritten.
- The value of a constant must be computable at compile time. Therefore, you can't initialize a constant with a value taken from a variable. If you need to do this, you must use a read-only field (this is explained in Chapter 3).
- Constants are always implicitly static. However, notice that you don't have to (and, in fact, are not permitted to) include the `static` modifier in the constant declaration.

At least three advantages exist for using constants in your programs:

- Constants make your programs easier to read by replacing magic numbers and strings with readable names whose values are easy to understand.
- Constants make your programs easier to modify. For example, assume that you have a `SalesTax` constant in one of your C# programs, and that constant is assigned a value of 6 percent. If the sales tax rate changes later, you can modify the behavior of all tax calculations simply by assigning a new value to the constant; you don't have to hunt through your code for the value `.06` and change each one, hoping you will find all of them.
- Constants help prevent mistakes in your programs. If you attempt to assign another value to a constant somewhere in your program other than at the point where the constant is declared, the compiler flags the error.

USING PREDEFINED DATA TYPES

Now that you have seen how to declare variables and constants, let's take a closer look at the data types available in C#. As you will see, C# is much stricter about the types available and their definitions than

some other languages.

Value Types and Reference Types

Before examining the data types in C#, it is important to understand that C# distinguishes between two categories of data type:

- Value types
- Reference types

The next few sections look in detail at the syntax for value and reference types. Conceptually, the difference is that a *value type* stores its value directly, whereas a *reference type* stores a reference to the value.

These types are stored in different places in memory; value types are stored in an area known as the *stack*, and reference types are stored in an area known as the *managed heap*. It is important to be aware of whether a type is a value type or a reference type because of the different effect each assignment has. For example, `int` is a value type, which means that the following statement results in two locations in memory storing the value `20`:

```
// i and j are both of type int
i = 20;
j = i;
```

However, consider the following example. For this code, assume you have defined a class called `Vector` and that `Vector` is a reference type and has an `int` member variable called `Value`:

```
Vector x, y;
x = new Vector();
x.Value = 30; // Value is a field defined in Vector class
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

The crucial point to understand is that after executing this code, there is only one `Vector` object: `x` and `y` both point to the memory location that contains this object. Because `x` and `y` are variables of a reference

type, declaring each variable simply reserves a reference—it doesn't instantiate an object of the given type. In neither case is an object actually created. To create an object, you have to use the `new` keyword, as shown. Because `x` and `y` refer to the same object, changes made to `x` will affect `y` and vice versa. Hence, the code will display `30` and then `50`.

If a variable is a reference, it is possible to indicate that it does not refer to any object by setting its value to `null`:

```
y = null;
```

If a reference is set to `null`, then it is not possible to call any nonstatic member functions or fields against it; doing so would cause an exception to be thrown at runtime.

> **NOTE**
>
> *Non-nullable reference types are planned for C# 8. Variables of these types require initialization with non-null. Reference types that allow* `null` *explicitly require declaration as a nullable reference type.*

In C#, basic data types such as `bool` and `long` are value types. This means that if you declare a `bool` variable and assign it the value of another `bool` variable, you will have two separate `bool` values in memory. Later, if you change the value of the original `bool` variable, the value of the second `bool` variable does not change. These types are copied by value.

In contrast, most of the more complex C# data types, including classes that you yourself declare, are reference types. They are allocated upon the heap, have lifetimes that can span multiple function calls, and can be accessed through one or several aliases. The CLR implements an elaborate algorithm to track which reference variables are still reachable and which have been orphaned. Periodically, the CLR destroys orphaned objects and returns the memory that they once occupied back to the operating system. This is done by the garbage

collector.

C# has been designed this way because high performance is best served by keeping primitive types (such as `int` and `bool`) as value types, and larger types that contain many fields (as is usually the case with classes) as reference types. If you want to define your own type as a value type, you should declare it as a struct.

> **NOTE**
>
> *The layout of primitive data types typically aligns with native layouts. This makes it possible to share the same memory between managed and native code.*

.NET Types

The C# keywords for data types—such as `int`, `short`, and `string`—are mapped from the compiler to .NET data types. For example, when you declare an `int` in C#, you are actually declaring an instance of a .NET struct: `System.Int32`. This might sound like a small point, but it has a profound significance: It means that you can treat all the primitive data types syntactically, as if they are classes that support certain methods. For example, to convert an `int i` to a `string`, you can write the following:

```
string s = i.ToString();
```

It should be emphasized that behind this syntactical convenience, the types really are stored as primitive types, so absolutely no performance cost is associated with the idea that the primitive types are notionally represented by C# structs.

The following sections review the types that are recognized as built-in types in C#. Each type is listed, along with its definition and the name of the corresponding .NET type. C# has 15 predefined types, 13 value types, and 2 (`string` and `object`) reference types.

Predefined Value Types

The built-in .NET value types represent primitives, such as integer and floating-point numbers, character, and Boolean types.

Integer Types

C# supports eight predefined integer types, shown in the following table.

NAME	.NET TYPE	DESCRIPTION	RANGE (MIN:MAX)
`sbyte`	`System.SByte`	8-bit signed integer	–128:127 (-2^7:2^7-1)
`short`	`System.Int16`	16-bit signed integer	–32,768:32,767 (-2^{15}:$2^{15}-1$)
`int`	`System.Int32`	32-bit signed integer	–2,147,483,648:2,147,483,647 (-2^{31}:$2^{31}-1$)
`long`	`System.Int64`	64-bit signed integer	–9,223,372,036,854,775,808 9,223,372,036,854,775,807 (-2^{63}:$2^{63}-1$)
`byte`	`System.Byte`	8-bit unsigned integer	0:255 (0:2^8-1)
`ushort`	`System.UInt16`	16-bit unsigned integer	0:65,535 (0:$2^{16}-1$)
`uint`	`System.UInt32`	32-bit unsigned integer	0:4,294,967,295 (0:$2^{32}-1$)
`ulong`	`System.UInt64`	64-bit unsigned integer	0:18,446,744,073,709,551,61 (0:$2^{64}-1$)

Some C# types have the same names as C++ and Java types but have different definitions. For example, in C# an `int` is always a 32-bit signed integer. In C++ an `int` is a signed integer, but the number of bits is platform-dependent (32 bits on Windows). In C#, all data types have been defined in a platform-independent manner to allow for the

possible future porting of C# and .NET to other platforms.

A `byte` is the standard 8-bit type for values in the range 0 to 255 inclusive. Be aware that, in keeping with its emphasis on type safety, C# regards the `byte` type and the `char` type as completely distinct types, and any programmatic conversions between the two must be explicitly requested. Also, be aware that unlike the other types in the integer family, a `byte` type is by default unsigned. Its signed version bears the special name `sbyte`.

With .NET, a `short` is no longer quite so short; it is 16 bits long. The `int` type is 32 bits long. The `long` type reserves 64 bits for values. All integer-type variables can be assigned values in decimal, hex, or binary notation. Binary notation requires the `0b` prefix; hex notation requires the `0x` prefix:

```
long x = 0x12ab;
```

Binary notation is discussed later in the section "Working with Binary Values."

If there is any ambiguity about whether an integer is `int`, `uint`, `long`, or `ulong`, it defaults to an `int`. To specify which of the other integer types the value should take, you can append one of the following characters to the number:

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

You can also use lowercase `u` and `l`, although the latter could be confused with the integer `1` (one).

Digit Separators

C# 7 offers digit separators. These separators help with readability and don't add any functionality. For example, you can add underscores to numbers, as shown in the following code snippet (code file `UsingNumbers/Program.cs`):

```
long l1 = 0x123_4567_89ab_cedf;
```

The underscores used as separators are ignored by the compiler. With the preceding sample, reading from the right every 16 bits (or four hexadecimal characters) a digit separator is added. The result is a lot more readable than the alternative:

```
long l2 = 0x123456789abcedf;
```

Because the compiler just ignores the underscores, you are responsible for ensuring readability. You can put the underscores at any position, you need to make sure it helps readability, not as shown in this example:

```
long l3 = 0x12345_6789_abc_ed_f;
```

It's useful to have it allowed on any position as this allows for different use cases—for example, to work with hexadecimal or octal values, or to separate different bits needed for a protocol (as shown in the next section).

> **NOTE**
>
> *Digit separators are new with C# 7. C# 7.0 doesn't allow leading digit separators, having the separator before the value (and after the prefix). Leading digit separators can be used with C# 7.2.*

Working with Binary Values

Besides offering digit separators, C# 7 also makes it easier to assign binary values to integer types. If you prefix the variable value with the 0b literal, it's only allowed to use 0 and 1. Only binary values are allowed to assign to the variable, as you can see in the following code snippet (code file `UsingNumbers/Program.cs`):

```
uint binary1 = 0b1111_1110_1101_1100_1011_1010_1001_1000;
```

This preceding code snippet uses an unsigned `int` with 32 bits available. Digit separators help a lot with readability in binary values. This snippet makes a separation every four bits. Remember, you can

write this in the hex notation as well:

```
uint hex1 = 0xfedcba98;
```

Using the separator every three bits helps when you're working with the octal notation, where characters are used between 0 (000 binary) and 7 (111 binary).

```
uint binary2 = 0b111_110_101_100_011_010_001_000;
```

The following example shows how to define values that could be used in a binary protocol where two bits define the rightmost part, six bits are in the next section, and the last two sections have four bits to complete 16 bits:

```
ushort binary3 = 0b1111_0000_101010_11;
```

Remember to use the correct integer type for the number of bits needed: `ushort` for 16, `uint` for 32, and `ulong` for 64 bits.

> **NOTE**
>
> *Read Chapter 6, "Operators and Casts," and Chapter 11, "Special Collections," for additional information on working with binary data.*

> **NOTE**
>
> *Binary literals are new with C# 7.*

Floating-Point Types

Although C# provides a plethora of integer data types, it supports floating-point types as well.

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPROX

`float`	`System.Single`	32-bit, single-precision floating point	7	$\pm 1.5 \times 10^{24}$ $\times 10^{38}$
`double`	`System.Double`	64-bit, double-precision floating point	15/16	$\pm 5.0 \times 10^{23}$ $\pm 1.7 \times 10^{30}$

The `float` data type is for smaller floating-point values, for which less precision is required. The `double` data type is bulkier than the `float` data type but offers twice the precision (15 digits).

If you hard-code a non-integer number (such as 12.3), the compiler will normally assume that you want the number interpreted as a `double`. To specify that the value is a `float`, append the character `F` (or `f`) to it:

```
float f = 12.3F;
```

The Decimal Type

The `decimal` type represents higher-precision floating-point numbers, as shown in the following table.

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE (APPRO
`decimal`	`System.Decimal`	128-bit, high-precision decimal notation	28	$\pm 1.0 \times 10^{2}$ $\times 10^{28}$

One of the great things about the .NET and C# data types is the provision of a dedicated `decimal` type for financial calculations. How you use the 28 digits that the decimal type provides is up to you. In other words, you can track smaller dollar amounts with greater accuracy for cents or larger dollar amounts with more rounding in the fractional portion. Bear in mind, however, that `decimal` is not implemented under the hood as a primitive type, so using `decimal` has a performance effect on your calculations.

To specify that your number is a `decimal` type rather than a `double`, a

`float`, or an integer, you can append the `M` (or `m`) character to the value, as shown here:

```
decimal d = 12.30M;
```

The Boolean Type

The C# `bool` type is used to contain Boolean values of either `true` or `false`.

NAME	.NET TYPE	DESCRIPTION	SIGNIFICANT FIGURES	RANGE
`bool`	`System.Boolean`	Represents true or false	NA	true or false

You cannot implicitly convert `bool` values to and from integer values. If a variable (or a function return type) is declared as a `bool`, you can only use values of `true` and `false`. You get an error if you try to use zero for `false` and a nonzero value for `true`.

The Character Type

For storing the value of a single character, C# supports the `char` data type.

NAME	.NET TYPE	VALUES
`char`	`System.Char`	Represents a single 16-bit (Unicode) character

Literals of type `char` are signified by being enclosed in single quotation marks—for example, `'A'`. If you try to enclose a character in double quotation marks, the compiler treats the character as a string and throws an error.

As well as representing chars as character literals, you can represent them with four-digit hex Unicode values (for example, `'\u0041'`), as integer values with a cast (for example, `(char)65`), or as hexadecimal values (for example,`'\x0041'`). You can also represent them with an escape sequence, as shown in the following table.

ESCAPE SEQUENCE	CHARACTER
`\'`	Single quotation mark

`\"`	Double quotation mark
`\\`	Backslash
`\0`	Null
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab character
`\v`	Vertical tab

Literals for Numbers

The following table summarizes the literals that can be used for numbers. The table repeats the literals from the preceding sections so they're all collected in one place.

LITERAL	POSITION	DESCRIPTION
`U`	Postfix	unsigned int
`L`	Postfix	long
`UL`	Postfix	unsigned long
`F`	Postfix	float
`M`	Postfix	decimal (money)
`0x`	Prefix	Hexadecimal number; values from 0 to F are allowed
`0b`	Prefix	Binary number; only 0 and 1 are allowed
`true`	NA	Boolean value
`False`	NA	Boolean value

Predefined Reference Types

C# supports two predefined reference types, `object` and `string`,

described in the following table.

NAME	.NET TYPE	DESCRIPTION
`object`	`System.Object`	The root type. All other types (including value types) are derived from `object`.
`string`	`System.String`	Unicode character string

The object Type

Many programming languages and class hierarchies provide a root type, from which all other objects in the hierarchy are derived. C# and .NET are no exception. In C#, the `object` type is the ultimate parent type from which all other intrinsic and user-defined types are derived. This means that you can use the `object` type for two purposes:

- You can use an `object` reference to bind to an object of any particular subtype. For example, in Chapter 6, "Operators and Casts," you see how you can use the `object` type to box a value object on the stack to move it to the heap; `object` references are also useful in reflection, when code must manipulate objects whose specific types are unknown.
- The `object` type implements a number of basic, general-purpose methods, which include `Equals`, `GetHashCode`, `GetType`, and `ToString`. Responsible user-defined classes might need to provide replacement implementations of some of these methods using an object-oriented technique known as *overriding*, which is discussed in Chapter 4, "Object Oriented Programming with C#." When you override `ToString`, for example, you equip your class with a method for intelligently providing a string representation of itself. If you don't provide your own implementations for these methods in your classes, the compiler picks up the implementations in `object`, which might or might not be correct or sensible in the context of your classes.

You examine the `object` type in more detail in subsequent chapters.

The string Type

C# recognizes the `string` keyword, which under the hood is translated

to the .NET class, `System.String`. With it, operations like string concatenation and string copying are a snap:

```
string str1 = "Hello ";
string str2 = "World";
 string str3 = str1 + str2; // string concatenation
```

Despite this style of assignment, `string` is a reference type. Behind the scenes, a `string` object is allocated on the heap, not the stack; and when you assign one string variable to another string, you get two references to the same string in memory. However, `string` differs from the usual behavior for reference types. For example, strings are immutable. Making changes to one of these strings creates an entirely new `string` object, leaving the other string unchanged. Consider the following code (code file `StringSample/Program.cs`):

```
using System;

class Program
{
  static void Main()
  {
    string s1 = "a string";
    string s2 = s1;
    Console.WriteLine("s1 is " + s1);
    Console.WriteLine("s2 is " + s2);
    s1 = "another string";
    Console.WriteLine("s1 is now " + s1);
    Console.WriteLine("s2 is now " + s2);
  }
}
```

The output from this is as follows:

```
s1 is a string
s2 is a string
s1 is now another string
s2 is now a string
```

Changing the value of `s1` has no effect on `s2`, contrary to what you'd expect with a reference type! What's happening here is that when `s1` is initialized with the value `a string`, a new string object is allocated on the heap. When `s2` is initialized, the reference points to this same object, so `s2` also has the value `a string`. However, when you now

change the value of `s1`, instead of replacing the original value, a new object is allocated on the heap for the new value. The `s2` variable still points to the original object, so its value is unchanged. Under the hood, this happens as a result of operator overloading, a topic that is explored in Chapter 6. In general, the `string` class has been implemented so that its semantics follow what you would normally intuitively expect for a string.

String literals are enclosed in double quotation marks (`"."`); if you attempt to enclose a string in single quotation marks, the compiler takes the value as a `char` and throws an error. C# strings can contain the same Unicode and hexadecimal escape sequences as `chars`. Because these escape sequences start with a backslash, you can't use this character unescaped in a string. Instead, you need to escape it with two backslashes (`\\`):

```
string filepath = "C:\\ProCSharp\\First.cs";
```

WARNING

Be aware that using backslash (`\`) for directories and using `C:` restricts the application to the Windows operating system. Both Windows and Linux can use the forward slash (`/`) to separate directories. Chapter 22, "Files and Streams," gives you details about how to work with files and directories both on Windows and Linux.

Even if you are confident that you can remember to do this all the time, typing all those double backslashes can prove annoying. Fortunately, C# gives you an alternative. You can prefix a string literal with the at character (`@`) and all the characters after it are treated at face value; they aren't interpreted as escape sequences:

```
string filepath = @"C:\ProCSharp\First.cs";
```

This even enables you to include line breaks in your string literals:

```
string jabberwocky = @"'Twas brillig and the slithy toves
```

```
Did gyre and gimble in the wabe.";
```

In this case, the value of `jabberwocky` would be this:

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

C# defines a string interpolation format that is marked by using the `$` prefix. You've previously seen this prefix in the section "Working with Variables." You can change the earlier code snippet that demonstrated string concatenation to use the string interpolation format. Prefixing a string with `$` enables you to put curly braces into the string that contains a variable—or even a code expression. The result of the variable or code expression is put into the string at the position of the curly braces:

```
public static void Main()
{
  string s1 = "a string";
  string s2 = s1;
  Console.WriteLine($"s1 is {s1}");
  Console.WriteLine($"s2 is {s2}");
  s1 = "another string";
  Console.WriteLine($"s1 is now {s1}");
  Console.WriteLine($"s2 is now {s2}");
}
```

> **NOTE**
>
> *Note Strings and the features of string interpolation are covered in detail in Chapter 9, "Strings and Regular Expressions."*

CONTROLLING PROGRAM FLOW

This section looks at the real nuts and bolts of the language: the statements that allow you to control the *flow* of your program rather than execute every line of code in the order it appears in the program.

Conditional Statements

Conditional statements enable you to branch your code depending on whether certain conditions are met or what the value of an expression is. C# has two constructs for branching code: the `if` statement, which tests whether a specific condition is met, and the `switch` statement, which compares an expression with several different values.

The if Statement

For conditional branching, C# inherits the C and C++ `if.else` construct. The syntax should be fairly intuitive for anyone who has done any programming with a procedural language:

```
if (condition)
 statement(s)
else
 statement(s)
```

If more than one statement is to be executed as part of either condition, these statements need to be joined into a block using curly braces (`{.}`). (This also applies to other C# constructs where statements can be joined into a block, such as the `for` and `while` loops):

```
bool isZero;
if (i == 0)
{
  isZero = true;
  Console.WriteLine("i is Zero");
}
else
{
  isZero = false;
  Console.WriteLine("i is Non-zero");
}
```

If you want to, you can use an `if` statement without a final `else` statement. You can also combine `else if` clauses to test for multiple conditions (code file `IfStatement/Program.cs`):

```
using System;
namespace Wrox
{
  class Program
```

```
  {
    static void Main()
    {
      Console.WriteLine("Type in a string");
      string input;
      input = Console.ReadLine();
      if (input == "")
      {
        Console.WriteLine("You typed in an empty string.");
      }
      else if (input.Length < 5)
      {
        Console.WriteLine("The string had less than 5
characters.");
      }
      else if (input.Length < 10)
      {
        Console.WriteLine(
          "The string had at least 5 but less than 10
Characters.");
      }
      Console.WriteLine("The string was " + input);
    }
  }
}
```

There is no limit to how many `else if`s you can add to an `if` clause.

Note that the previous example declares a string variable called `input`, gets the user to enter text at the command line, feeds this into `input`, and then tests the length of this string variable. The code also shows how easy string manipulation can be in C#. To find the length of `input`, for example, use `input.Length`.

Another point to note about the `if` statement is that you don't need to use the braces when there's only one statement in the conditional branch:

```
if (i == 0)
  Console.WriteLine("i is Zero"); // This will only execute
if i == 0
  Console.WriteLine("i can be anything"); // Will execute
whatever the
// value of i
```

However, for consistency, many programmers prefer to use curly

braces whenever they use an `if` statement.

> **TIP**
>
> *Not using curly braces with* `if` *statements can lead to errors in maintaining the code. It happens too often that a second statement is added to the* `if` *statement that runs no matter whether the* `if` *returns true or false. Using curly braces every time avoids this coding error.*
>
> *A good guideline in regard to the* `if` *statement is to allow programmers to not use curly braces only when the statement is written in the same line as the* `if` *statement. With this guideline, programmers are less likely to add a second statement without adding curly braces.*

The `if` statements presented also illustrate some of the C# operators that compare values. Note in particular that C# uses == to compare variables for equality. Do not use = for this purpose. A single = is used to assign values.

In C#, the expression in the `if` clause must evaluate to a Boolean. It is not possible to test an integer directly (returned from a function, for example). You have to convert the integer that is returned to a Boolean `true` or `false`, for example, by comparing the value with zero or `null`:

```
if (DoSomething() != 0)
{
  // Non-zero value returned
}
else
{
  // Returned zero
}
```

The switch Statement

The `switch / case` statement is good for selecting one branch of execution from a set of mutually exclusive ones. It takes the form of a

`switch` argument followed by a series of `case` clauses. When the expression in the `switch` argument evaluates to one of the values beside a `case` clause, the code immediately following the `case` clause executes. This is one example for which you don't need to use curly braces to join statements into blocks; instead, you mark the end of the code for each case using the `break` statement. You can also include a `default` case in the `switch` statement, which executes if the expression doesn't evaluate to any of the other cases. The following `switch` statement tests the value of the `integerA` variable:

```
switch (integerA)
{
  case 1:
    Console.WriteLine("integerA = 1");
    break;
  case 2:
    Console.WriteLine("integerA = 2");
    break;
  case 3:
    Console.WriteLine("integerA = 3");
    break;
  default:
    Console.WriteLine("integerA is not 1, 2, or 3");
    break;
}
```

Note that the case values must be constant expressions; variables are not permitted.

Though the `switch.case` statement should be familiar to C and C++ programmers, C#'s `switch.case` is a bit safer than its C++ equivalent. Specifically, it prohibits fall-through conditions in almost all cases. This means that if a `case` clause is fired early on in the block, later clauses cannot be fired unless you use a `goto` statement to indicate that you want them fired, too. The compiler enforces this restriction by flagging every `case` clause that is not equipped with a `break` statement as an error:

```
Control cannot fall through from one case label ('case 2:')
to another
```

Although it is true that fall-through behavior is desirable in a limited number of situations, in the vast majority of cases it is unintended and

results in a logical error that's hard to spot. Isn't it better to code for the norm rather than for the exception?

By getting creative with `goto` statements, you can duplicate fall-through functionality in your `switch.cases`. However, if you find yourself really wanting to, you probably should reconsider your approach. The following code illustrates both how to use `goto` to simulate fall-through, and how messy the resultant code can be:

```
// assume country and language are of type string
switch(country)
{
  case "America":
    CallAmericanOnlyMethod();
    goto case "Britain";
  case "France":
    language = "French";
    break;
  case "Britain":
    language = "English";
    break;
}
```

There is one exception to the no-fall-through rule, however, in that you can fall through from one case to the next if that case is empty. This allows you to treat two or more cases in an identical way (without the need for `goto` statements):

```
switch(country)
{
  case "au":
  case "uk":
  case "us":
    language = "English";
    break;
  case "at":
  case "de":
    language = "German";
    break;
}
```

One intriguing point about the `switch` statement in C# is that the order of the cases doesn't matter—you can even put the `default` case first! As a result, no two cases can be the same. This includes different

constants that have the same value, so you can't, for example, do this:

```
// assume country is of type string
const string england = "uk";
const string britain = "uk";
switch(country)
{
  case england:
  case britain: // This will cause a compilation error.
    language = "English";
    break;
}
```

The previous code also shows another way in which the `switch` statement is different in C# compared to C++: In C#, you are allowed to use a string as the variable being tested.

> **NOTE**
>
> *With C# 7, the* `switch` *statement has been enhanced with pattern matching. Using pattern matching, the ordering of the cases becomes important. Read Chapter 13, "Functional Programming with C#," for more information about the* `switch` *statement using pattern matching.*

Loops

C# provides four different loops (`for`, `while`, `do…while`, and `foreach`) that enable you to execute a block of code repeatedly until a certain condition is met.

The for Loop

C# `for` loops provide a mechanism for iterating through a loop whereby you test whether a particular condition holds true before you perform another iteration. The syntax is

where:

- The `initializer` is the expression evaluated before the first loop is

executed (usually initializing a local variable as a loop counter).

- The `condition` is the expression checked before each new iteration of the loop (this must evaluate to `true` for another iteration to be performed).
- The `iterator` is an expression evaluated after each iteration (usually incrementing the loop counter).

The iterations end when the condition evaluates to `false`.

The `for` loop is a so-called pretest loop because the loop condition is evaluated before the loop statements are executed; therefore, the contents of the loop won't be executed at all if the loop condition is `false`.

The `for` loop is excellent for repeating a statement or a block of statements for a predetermined number of times. The following example demonstrates typical usage of a `for` loop. It writes out all the integers from 0 to 99:

```
for (int i = 0; i < 100; i = i + 1)
{
  Console.WriteLine(i);
}
```

Here, you declare an `int` called `i` and initialize it to zero. This is used as the loop counter. You then immediately test whether it is less than 100. Because this condition evaluates to `true`, you execute the code in the loop, displaying the value `0`. You then increment the counter by one, and walk through the process again. Looping ends when `i` reaches 100.

Actually, the way the preceding loop is written isn't quite how you would normally write it. C# has a shorthand for adding 1 to a variable, so instead of `i = i + 1`, you can simply write `i++`:

```
for (int i = 0; i < 100; i++)
{
  // ...
}
```

You can also make use of type inference for the iteration variable `i` in

the preceding example. Using type inference, the loop construct would be as follows:

```
for (var i = 0; i < 100; i++)
{
  // ...
}
```

It's not unusual to nest `for` loops so that an inner loop executes once completely for each iteration of an outer loop. This approach is typically employed to loop through every element in a rectangular multidimensional array. The outermost loop loops through every row, and the inner loop loops through every column in a particular row. The following code displays rows of numbers. It also uses another `Console` method, `Console.Write`, which does the same thing as `Console.WriteLine` but doesn't send a carriage return to the output (code file `ForLoop/Program.cs`):

```
using System;

namespace Wrox
{
  class Program
  {
    static void Main()
    {
      // This loop iterates through rows
      for (int i = 0; i < 100; i+=10)
      {
        // This loop iterates through columns
        for (int j = i; j < i + 10; j++)
        {
          Console.Write($" {j}");
        }
        Console.WriteLine();
      }
    }
  }
}
```

Although `j` is an integer, it is automatically converted to a string so that the concatenation can take place.

The preceding sample results in this output:

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

It is technically possible to evaluate something other than a counter variable in a `for` loop's test condition, but it is certainly not typical. It is also possible to omit one (or even all) of the expressions in the `for` loop. In such situations, however, you should consider using the `while` loop.

The while Loop

Like the `for` loop, `while` is a pretest loop. The syntax is similar, but `while` loops take only one expression:

```
while(condition)
  statement(s);
```

Unlike the `for` loop, the `while` loop is most often used to repeat a statement or a block of statements for a number of times that is not known before the loop begins. Usually, a statement inside the `while` loop's body will set a Boolean flag to `false` on a certain iteration, triggering the end of the loop, as in the following example:

```
bool condition = false;
while (!condition)
{
  // This loop spins until the condition is true.
  DoSomeWork();
  condition = CheckCondition(); // assume CheckCondition()
returns a bool
}
```

The do…while Loop

The `do…while` loop is the post-test version of the `while` loop. This means that the loop's test condition is evaluated after the body of the

loop has been executed. Consequently, `do…while` loops are useful for situations in which a block of statements must be executed at least one time, as in this example:

```
bool condition;
do
{
  // This loop will at least execute once, even if Condition
is false.
  MustBeCalledAtLeastOnce();
  condition = CheckCondition();
} while (condition);
```

The foreach Loop

The `foreach` loop enables you to iterate through each item in a collection. For now, don't worry about exactly what a collection is (it is explained fully in Chapter 10, "Collections"); just understand that it is an object that represents a list of objects. Technically, for an object to count as a collection, it must support an interface called `IEnumerable`. Examples of collections include C# arrays, the collection classes in the `System.Collections` namespaces, and user-defined collection classes. You can get an idea of the syntax of `foreach` from the following code, if you assume that `arrayOfInts` is (unsurprisingly) an array of `int`s:

```
foreach (int temp in arrayOfInts)
{
  Console.WriteLine(temp);
}
```

Here, `foreach` steps through the array one element at a time. With each element, it places the value of the element in the `int` variable called `temp` and then performs an iteration of the loop.

Here is another situation where you can use type inference. The `foreach` loop would become the following:

```
foreach (var temp in arrayOfInts)
{
  // ...
}
```

`temp` would be inferred to `int` because that is what the collection item

type is.

An important point to note with `foreach` is that you can't change the value of the item in the collection (`temp` in the preceding code), so code such as the following will not compile:

```
foreach (int temp in arrayOfInts)
{
  temp++;
  Console.WriteLine(temp);
}
```

If you need to iterate through the items in a collection and change their values, you must use a `for` loop instead.

Jump Statements

C# provides a number of statements that enable you to jump immediately to another line in the program. The first of these is, of course, the notorious `goto` statement.

The goto Statement

The `goto` statement enables you to jump directly to another specified line in the program, indicated by a *label* (this is just an identifier followed by a colon):

```
goto Label1;
Console.WriteLine("This won't be executed");
Label1:
  Console.WriteLine("Continuing execution from here");
```

A couple of restrictions are involved with `goto`. You can't jump into a block of code such as a `for` loop, you can't jump out of a class, and you can't exit a `finally` block after `try…catch` blocks (Chapter 14, "Errors and Exceptions," looks at exception handling with `try.catch.finally`).

The reputation of the `goto` statement probably precedes it, and in most circumstances, its use is sternly frowned upon. In general, it certainly doesn't conform to good object-oriented programming practices.

The break Statement

You have already met the `break` statement briefly—when you used it to exit from a case in a `switch` statement. In fact, `break` can also be used to exit from `for`, `foreach`, `while`, or `do…while` loops. Control switches to the statement immediately after the end of the loop.

If the statement occurs in a nested loop, control switches to the end of the innermost loop. If the break occurs outside a `switch` statement or a loop, a compile-time error occurs.

The continue Statement

The `continue` statement is similar to `break`, and you must use it within a `for`, `foreach`, `while`, or `do…while` loop. However, it exits only from the current iteration of the loop, meaning that execution restarts at the beginning of the next iteration of the loop rather than restarting outside the loop altogether.

The return Statement

The `return` statement is used to exit a method of a class, returning control to the caller of the method. If the method has a return type, `return` must return a value of this type; otherwise, if the method returns `void`, you should use `return` without an expression.

GETTING ORGANIZED WITH NAMESPACES

As discussed earlier in this chapter, namespaces provide a way to organize related classes and other types. Unlike a file or a component, a namespace is a logical, rather than a physical, grouping. When you define a class in a C# file, you can include it within a namespace definition. Later, when you define another class that performs related work in another file, you can include it within the same namespace, creating a logical grouping that indicates to other developers using the classes how they are related and used:

```
using System;
namespace CustomerPhoneBookApp
{
  public struct Subscriber
  {
    // Code for struct here..
```

```
  }
}
```

Placing a type in a namespace effectively gives that type a long name, consisting of the type's namespace as a series of names separated with periods (.), terminating with the name of the class. In the preceding example, the full name of the `Subscriber` struct is `CustomerPhoneBookApp.Subscriber`. This enables distinct classes with the same short name to be used within the same program without ambiguity. This full name is often called the *fully qualified name*.

You can also nest namespaces within other namespaces, creating a hierarchical structure for your types:

```
namespace Wrox
{
  namespace ProCSharp
  {
    namespace Basics
    {
      class NamespaceExample
      {
        // Code for the class here..
      }
    }
  }
}
```

Each namespace name is composed of the names of the namespaces it resides within, separated with periods, starting with the outermost namespace and ending with its own short name. Therefore, the full name for the `ProCSharp` namespace is `Wrox.ProCSharp`, and the full name of the `NamespaceExample` class is `Wrox.ProCSharp.Basics.NamespaceExample`.

You can use this syntax to organize the namespaces in your namespace definitions too, so the previous code could also be written as follows:

```
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    // Code for the class here..
  }
```

```
}
```

Note that you are not permitted to declare a multipart namespace nested within another namespace.

Namespaces are not related to assemblies. It is perfectly acceptable to have different namespaces in the same assembly or to define types in the same namespace in different assemblies.

You should define the namespace hierarchy prior to starting a project. Generally the accepted format is `CompanyName.ProjectName.SystemSection`. In the previous example, Wrox is the company name, ProCSharp is the project, and in the case of this chapter, Basics is the section.

The using Directive

Obviously, namespaces can grow rather long and tiresome to type, and the capability to indicate a particular class with such specificity may not always be necessary. Fortunately, as noted earlier in this chapter, C# allows you to abbreviate a class's full name. To do this, list the class's namespace at the top of the file, prefixed with the `using` keyword. Throughout the rest of the file, you can refer to the types in the namespace simply by their type names:

```
using System;
using Wrox.ProCSharp;
```

As mentioned earlier, many C# files have the statement `using System;` simply because so many useful classes supplied by Microsoft are contained in the `System` namespace.

If two namespaces referenced by `using` statements contain a type of the same name, you need to use the full (or at least a longer) form of the name to ensure that the compiler knows which type to access. For example, suppose classes called `NamespaceExample` exist in both the `Wrox.ProCSharp.Basics` and `Wrox.ProCSharp.OOP` namespaces. If you then create a class called `Test` in the `Wrox.ProCSharp` namespace, and instantiate one of the `NamespaceExample` classes in this class, you need to specify which of these two classes you're talking about:

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;
namespace Wrox.ProCSharp
{
  class Test
  {
    static void Main()
    {
      Basics.NamespaceExample nSEx = new
Basics.NamespaceExample();
      // do something with the nSEx variable.
    }
  }
}
```

Your organization will probably want to spend some time developing a namespace convention so that its developers can quickly locate functionality that they need and so that the names of the organization's homegrown classes won't conflict with those in off-the-shelf class libraries. Guidelines on establishing your own namespace convention, along with other naming recommendations, are discussed later in this chapter.

Namespace Aliases

Another use of the `using` keyword is to assign aliases to classes and namespaces. If you need to refer to a very long namespace name several times in your code but don't want to include it in a simple `using` statement (for example, to avoid type name conflicts), you can assign an alias to the namespace. The syntax for this is as follows:

```
using alias = NamespaceName;
```

The following example (a modified version of the previous example) assigns the alias `Introduction` to the `Wrox.ProCSharp.Basics` namespace and uses this to instantiate a `NamespaceExample` object, which is defined in this namespace. Notice the use of the namespace alias qualifier (`::`). This forces the search to start with the `Introduction` namespace alias. If a class called `Introduction` had been introduced in the same scope, a conflict would occur. The `::` operator enables the alias to be referenced even if the conflict exists. The `NamespaceExample` class has one method, `GetNamespace`, which uses the

`GetType` method exposed by every class to access a `Type` object representing the class's type. You use this object to return a name of the class's namespace (code file `NamespaceSample/Program.cs`):

```
using System;
using Introduction = Wrox.ProCSharp.Basics;

class Program
{
  static void Main()
  {
    Introduction::NamespaceExample NSEx = new
Introduction::NamespaceExample();
    Console.WriteLine(NSEx.GetNamespace());
  }
}

namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    public string GetNamespace()
    {
      return this.GetType().Namespace;
    }
  }
}
```

UNDERSTANDING THE MAIN METHOD

As described at the beginning of this chapter, C# programs start execution at a method named `Main`. Depending on the execution environment there are different requirements.

- Have a `static` modifier applied
- Be in a class with any name
- Return a type of `int` or `void`

Although it is common to specify the `public` modifier explicitly—because by definition the method must be called from outside the program—it doesn't actually matter what accessibility level you assign to the entry-point method; it will run even if you mark the method as `private`.

The examples so far have shown only the `Main` method without any parameters. However, when the program is invoked, you can get the CLR to pass any command-line arguments to the program by including a parameter. This parameter is a string array, traditionally called `args` (although C# accepts any name). The program can use this array to access any options passed through the command line when the program is started.

The following example loops through the string array passed in to the `Main` method and writes the value of each option to the console window (code file `ArgumentsSample/Program.cs`):

```
using System;
namespace Wrox
{
  class Program
  {
    static void Main(string[] args)
    {
      for (int i = 0; i < args.Length; i++)
      {
        Console.WriteLine(args[i]);
      }
    }
  }
}
```

For passing arguments to the program when running the application from Visual Studio 2017, you can define the arguments in the Debug section of the project properties as shown in Figure 2-1. Running the application reveals the result to show all argument values to the console.

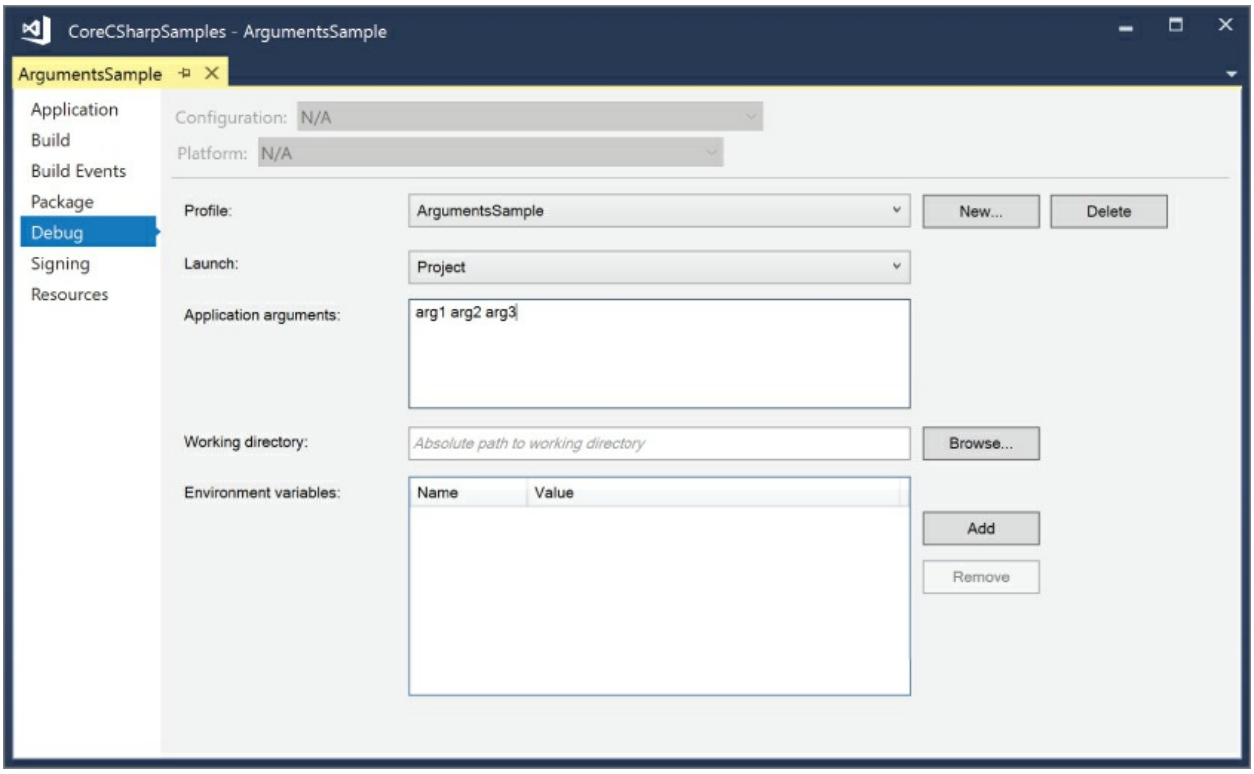

FIGURE 2-1

When you run the application from the command line using the .NET Core CLI tools, you just need to supply the arguments following the `dotnet run` command:

```
dotnet run arg1 arg2 arg3
```

In case you want to supply arguments that are in conflict with the arguments of the `dotnet run` command, you can add two dashes (`--`) before supplying the arguments of the program:

```
dotnet run -- arg1 arg2 arg3
```

USING COMMENTS

The next topic—adding comments to your code—looks very simple on the surface, but it can be complex. Comments can be beneficial to other developers who may look at your code. Also, as you will see, you can use comments to generate documentation of your code for other developers to use.

Internal Comments Within the Source Files

As noted earlier in this chapter, C# uses the traditional C-type single-line (`//..`) and multiline (`/* .. */`) comments:

```
// This is a single-line comment
/* This comment
spans multiple lines. */
```

Everything in a single-line comment, from the `//` to the end of the line, is ignored by the compiler, and everything from an opening `/*` to the next `*/` in a multiline comment combination is ignored. Obviously, you can't include the combination `*/` in any multiline comments, because this will be treated as the end of the comment.

It is possible to put multiline comments within a line of code:

```
Console.WriteLine(/* Here's a comment! */ "This will
compile.");
```

Use inline comments with care because they can make code hard to read. However, they can be useful when debugging if, for example, you temporarily want to try running the code with a different value somewhere:

```
DoSomething(Width, /*Height*/ 100);
```

Comment characters included in string literals are, of course, treated like normal characters:

```
string s = "/* This is just a normal string .*/";
```

XML Documentation

In addition to the C-type comments, illustrated in the preceding section, C# has a very neat feature: the capability to produce documentation in XML format automatically from special comments. These comments are single-line comments, but they begin with three slashes (`///`) instead of the usual two. Within these comments, you can place XML tags containing documentation of the types and type members in your code.

The tags in the following table are recognized by the compiler.

TAG	DESCRIPTION
`<c>`	Marks up text within a line as code—for example, `<c>int i = 10;</c>`.
`<code>`	Marks multiple lines as code.
`<example>`	Marks up a code example.
`<exception>`	Documents an exception class. (Syntax is verified by the compiler.)
`<include>`	Includes comments from another documentation file. (Syntax is verified by the compiler.)
`<list>`	Inserts a list into the documentation.
`<para>`	Gives structure to text.
`<param>`	Marks up a method parameter. (Syntax is verified by the compiler.)
`<paramref>`	Indicates that a word is a method parameter. (Syntax is verified by the compiler.)
`<permission>`	Documents access to a member. (Syntax is verified by the compiler.)
`<remarks>`	Adds a description for a member.
`<returns>`	Documents the return value for a method.
`<see>`	Provides a cross-reference to another parameter. (Syntax is verified by the compiler.)
`<seealso>`	Provides a “see also” section in a description. (Syntax is verified by the compiler.)
`<summary>`	Provides a short summary of a type or member.
`<typeparam>`	Describes a type parameter in the comment of a generic type.
`<typeparamref>`	Provides the name of the type parameter.
`<value>`	Describes a property.

Add some XML comments to the `Calculator.cs` file from the previous

section. You add a `<summary>` element for the class and for its `Add` method, and a `<returns>` element and two `<param>` elements for the `Add` method:

```
// MathLib.cs
namespace Wrox.MathLib
{
  ///<summary>
  /// Wrox.MathLib.Calculator class.
  /// Provides a method to add two doublies.
  ///</summary>
  public class Calculator
  {
    ///<summary>
    /// The Add method allows us to add two doubles.
    ///</summary>
    ///<returns>Result of the addition (double)</returns>
    ///<param name="x">First number to add</param>
    ///<param name="y">Second number to add</param>
    public static double Add(double x, double y) => x + y;
  }
}
```

UNDERSTANDING C# PREPROCESSOR DIRECTIVES

Besides the usual keywords, most of which you have now encountered, C# also includes a number of commands that are known as *preprocessor directives*. These commands are never actually translated to any commands in your executable code, but they affect aspects of the compilation process. For example, you can use preprocessor directives to prevent the compiler from compiling certain portions of your code. You might do this if you are planning to release two versions of it—a basic version and an enterprise version that will have more features. You could use preprocessor directives to prevent the compiler from compiling code related to the additional features when you are compiling the basic version of the software. In another scenario, you might have written bits of code that are intended to provide you with debugging information. You probably don't want those portions of code compiled when you actually ship the software.

The preprocessor directives are all distinguished by beginning with the

symbol.

> **NOTE**
>
> *C++ developers will recognize the preprocessor directives as something that plays an important part in C and C++. However, there aren't as many preprocessor directives in C#, and they are not used as often. C# provides other mechanisms, such as custom attributes, that achieve some of the same effects as C++ directives. Also, note that C# doesn't actually have a separate preprocessor in the way that C++ does. The so-called preprocessor directives are actually handled by the compiler. Nevertheless, C# retains the name preprocessor directive because these commands give the impression of a preprocessor.*

The following sections briefly cover the purposes of the preprocessor directives.

#define and #undef

`#define` is used like this:

```
#define DEBUG
```

This tells the compiler that a symbol with the given name (in this case `DEBUG`) exists. It is a little bit like declaring a variable, except that this variable doesn't really have a value—it just exists. Also, this symbol isn't part of your actual code; it exists only for the benefit of the compiler, while the compiler is compiling the code, and has no meaning within the C# code itself.

`#undef` does the opposite, and removes the definition of a symbol:

```
#undef DEBUG
```

If the symbol doesn't exist in the first place, then `#undef` has no effect. Similarly, `#define` has no effect if a symbol already exists.

You need to place any `#define` and `#undef` directives at the beginning of the C# source file, before any code that declares any objects to be compiled.

`#define` isn't much use on its own, but when combined with other preprocessor directives, especially `#if`, it becomes very powerful.

> **NOTE**
>
> *Incidentally, you might notice some changes from the usual C# syntax. Preprocessor directives are not terminated by semicolons and they normally constitute the only command on a line. That's because for the preprocessor directives, C# abandons its usual practice of requiring commands to be separated by semicolons. If the compiler sees a preprocessor directive, it assumes that the next command is on the next line.*

#if, #elif, #else, and #endif

These directives inform the compiler whether to compile a block of code. Consider this method:

```
int DoSomeWork(double x)
{
  // do something
  #if DEBUG
  Console.WriteLine($"x is {x}");
  #endif
}
```

This code compiles as normal except for the `Console.WriteLine` method call contained inside the `#if` clause. This line is executed only if the symbol `DEBUG` has been defined by a previous `#define` directive. When the compiler finds the `#if` directive, it checks to see whether the symbol concerned exists, and compiles the code inside the `#if` clause only if the symbol does exist. Otherwise, the compiler simply ignores all the code until it reaches the matching `#endif` directive. Typical practice is to define the symbol `DEBUG` while you are debugging and

have various bits of debugging-related code inside `#if` clauses. Then, when you are close to shipping, you simply comment out the `#define` directive, and all the debugging code miraculously disappears, the size of the executable file gets smaller, and your end users don't get confused by seeing debugging information. (Obviously, you would do more testing to ensure that your code still works without `DEBUG` defined.) This technique is very common in C and C++ programming and is known as *conditional compilation*.

The `#elif` (`=else if`) and `#else` directives can be used in `#if` blocks and have intuitively obvious meanings. It is also possible to nest `#if` blocks:

```
#define ENTERPRISE
#define W10
// further on in the file
#if ENTERPRISE
// do something
#if W10
// some code that is only relevant to enterprise
// edition running on W10
#endif
#elif PROFESSIONAL
// do something else
#else
// code for the leaner version
#endif
```

`#if` and `#elif` support a limited range of logical operators too, using the operators `!`, `==`, `!=`, and `||`. A symbol is considered to be `true` if it exists and `false` if it doesn't. For example:

```
#if W10 && (ENTERPRISE==false) // if W10 is defined but
ENTERPRISE isn't
```

#warning and #error

Two other very useful preprocessor directives are `#warning` and `#error`. These will respectively cause a warning or an error to be raised when the compiler encounters them. If the compiler sees a `#warning` directive, it displays whatever text appears after the `#warning` to the user, after which compilation continues. If it encounters an `#error`

directive, it displays the subsequent text to the user as if it is a compilation error message and then immediately abandons the compilation, so no IL code is generated.

You can use these directives as checks that you haven't done anything silly with your `#define` statements; you can also use the `#warning` statements to remind yourself to do something:

```
#if DEBUG && RELEASE
#error "You've defined DEBUG and RELEASE simultaneously!"
#endif
#warning "Don't forget to remove this line before the boss
tests the code!"
Console.WriteLine("*I love this job.*");
```

#region and #endregion

The `#region` and `#endregion` directives are used to indicate that a certain block of code is to be treated as a single block with a given name, like this:

```
#region Member Field Declarations
int x;
double d;
Currency balance;
#endregion
```

This doesn't look that useful by itself; it doesn't affect the compilation process in any way. However, the real advantage is that these directives are recognized by some editors, including the Visual Studio editor. These editors can use the directives to lay out your code better on the screen. You find out how this works in Chapter 18, "Visual Studio 2017."

#line

The `#line` directive can be used to alter the filename and line number information that is output by the compiler in warnings and error messages. You probably won't want to use this directive very often. It's most useful when you are coding in conjunction with another package that alters the code you are typing before sending it to the compiler. In

this situation, line numbers, or perhaps the filenames reported by the compiler, don't match up to the line numbers in the files or the filenames you are editing. The `#line` directive can be used to restore the match. You can also use the syntax `#line default` to restore the line to the default line numbering:

```
#line 164 "Core.cs" // We happen to know this is line 164 in
the file
// Core.cs, before the intermediate
// package mangles it.
// later on
#line default // restores default line numbering
```

#pragma

The `#pragma` directive can either suppress or restore specific compiler warnings. Unlike command-line options, the `#pragma` directive can be implemented on the class or method level, enabling fine-grained control over what warnings are suppressed and when. The following example disables the "field not used" warning and then restores it after the `MyClass` class compiles:

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

C# PROGRAMMING GUIDELINES

This final section of the chapter supplies the guidelines you need to bear in mind when writing C# programs. These are guidelines that most C# developers use. When you use these guidelines, other developers will feel comfortable working with your code.

Rules for Identifiers

This section examines the rules governing what names you can use for variables, classes, methods, and so on. Note that the rules presented in this section are not merely guidelines: they are enforced by the C#

compiler.

Identifiers are the names you give to variables, to user-defined types such as classes and structs, and to members of these types. Identifiers are case sensitive, so, for example, variables named `interestRate` and `InterestRate` would be recognized as different variables. Following are a few rules determining what identifiers you can use in C#:

- They must begin with a letter or underscore, although they can contain numeric characters.
- You can't use C# keywords as identifiers.

The following table lists reserved C# keywords.

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

If you need to use one of these words as an identifier (for example, if you are accessing a class written in a different language), you can

prefix the identifier with the `@` symbol to indicate to the compiler that what follows should be treated as an identifier, not as a C# keyword (so `abstract` is not a valid identifier, but `@abstract` is).

Finally, identifiers can also contain Unicode characters, specified using the syntax `\uXXXX`, where `XXXX` is the four-digit hex code for the Unicode character. The following are some examples of valid identifiers:

- `Name`
- `Überfluß`
- `_Identifier`
- `\u005fIdentifier`

The last two items in this list are identical and interchangeable (because `005f` is the Unicode code for the underscore character), so obviously these identifiers couldn't both be declared in the same scope. Note that although syntactically you are allowed to use the underscore character in identifiers, this isn't recommended in most situations. That's because it doesn't follow the guidelines for naming variables that Microsoft has written to ensure that developers use the same conventions, making it easier to read one another's code.

NOTE

You might wonder why some newer keywords added with the recent versions of C# are not in the list of reserved keywords. The reason is that if they had been added to the list of reserved keywords, it would have broken existing code that already made use of the new C# keywords. The solution was to enhance the syntax by defining these keywords as contextual keywords; they can be used only in some specific code places. For example, the `async` *keyword can be used only with a method declaration, and it is okay to use it as a variable name. The compiler doesn't have a conflict with that.*

Usage Conventions

In any development language, certain traditional programming styles usually arise. The styles are not part of the language itself but rather are conventions—for example, how variables are named or how certain classes, methods, or functions are used. If most developers using that language follow the same conventions, it makes it easier for different developers to understand each other's code—which in turn generally helps program maintainability. Conventions do, however, depend on the language and the environment. For example, C++ developers programming on the Windows platform have traditionally used the prefixes `psz` or `lpsz` to indicate strings—`char *pszResult; char *lpszMessage;`—but on Unix machines it's more common not to use any such prefixes: `char *Result; char *Message;`.

Notice from the sample code in this book that the convention in C# is to name local variables without prefixes: `string result; string message;`.

> **NOTE**
>
> *The convention by which variable names are prefixed with letters that represent the data type is known as Hungarian notation. It means that other developers reading the code can immediately tell from the variable name what data type the variable represents. Hungarian notation is widely regarded as redundant in these days of smart editors and IntelliSense.*

Whereas many languages' usage conventions simply evolved as the language was used, for C# and the whole of the .NET Framework, Microsoft has written very comprehensive usage guidelines, which are detailed in the .NET/C# documentation. This means that, right from the start, .NET programs have a high degree of interoperability in terms of developers being able to understand code. The guidelines have also been developed with the benefit of some 20 years' hindsight in object-oriented programming. Judging by the relevant newsgroups,

the guidelines have been carefully thought out and are well received in the developer community. Hence, the guidelines are well worth following.

Note, however, that the guidelines are not the same as language specifications. You should try to follow the guidelines when you can. Nevertheless, you won't run into problems if you have a good reason for not doing so—for example, you won't get a compilation error because you don't follow these guidelines. The general rule is that if you don't follow the usage guidelines, you must have a convincing reason. When you depart from the guidelines you should be making a conscious decision rather than simply not bothering. Also, if you compare the guidelines with the samples in the remainder of this book, you'll notice that in numerous examples I have chosen not to follow the conventions. That's usually because the conventions are designed for much larger programs than the samples; although the guidelines are great if you are writing a complete software package, they are not really suitable for small 20-line standalone programs. In many cases, following the conventions would have made the samples harder, rather than easier, to follow.

The full guidelines for good programming style are quite extensive. This section is confined to describing some of the more important guidelines, as well as those most likely to surprise you. To be absolutely certain that your code follows the usage guidelines completely, you need to refer to the Microsoft documentation.

Naming Conventions

One important aspect of making your programs understandable is how you choose to name your items—and that includes naming variables, methods, classes, enumerations, and namespaces.

It is intuitively obvious that your names should reflect the purpose of the item and should not clash with other names. The general philosophy in the .NET Framework is also that the name of a variable should reflect the purpose of that variable instance and not the data type. For example, `height` is a good name for a variable, whereas `integerValue` isn't. However, you are likely to find that principle is an

ideal that is hard to achieve. Particularly when you are dealing with controls, in most cases you'll probably be happier sticking with variable names such as `confirmationDialog` and `chooseEmployeeListBox`, which do indicate the data type in the name.

The following sections look at some of the things you need to think about when choosing names.

Casing of Names

In many cases you should use *Pascal casing* for names. With Pascal casing, the first letter of each word in a name is capitalized: `EmployeeSalary`, `ConfirmationDialog`, `PlainTextEncoding`. Notice that nearly all the names of namespaces, classes, and members in the base classes follow Pascal casing. In particular, the convention of joining words using the underscore character is discouraged. Therefore, try not to use names such as `employee_salary`. It has also been common in other languages to use all capitals for names of constants. This is not advised in C# because such names are harder to read—the convention is to use Pascal casing throughout:

```
const int MaximumLength;
```

The only other casing convention that you are advised to use is *camel casing*. Camel casing is similar to Pascal casing, except that the first letter of the first word in the name is not capitalized: `employeeSalary`, `confirmationDialog`, `plainTextEncoding`. Following are three situations in which you are advised to use camel casing:

- For names of all private member fields in types:

Note, however, that often it is conventional to prefix names of member fields with an underscore:

- For names of all parameters passed to methods
- To distinguish items that would otherwise have the same name. A common example is when a property wraps around a field:

```
private string employeeName;
public string EmployeeName
{
  get
```

```
    {
      return employeeName;
    }
  }
```

If you are wrapping a property around a field, you should always use camel casing for the private member and Pascal casing for the public or protected member, so that other classes that use your code see only names in Pascal case (except for parameter names).

You should also be wary about case sensitivity. C# is case sensitive, so it is syntactically correct for names in C# to differ only by the case, as in the previous examples. However, bear in mind that your assemblies might at some point be called from Visual Basic applications—and *Visual Basic is not case sensitive*. Hence, if you do use names that differ only by case, it is important to do so only in situations in which both names will never be seen outside your assembly. (The previous example qualifies as okay because camel case is used with the name that is attached to a `private` variable.) Otherwise, you may prevent other code written in Visual Basic from being able to use your assembly correctly.

Name Styles

Be consistent about your style of names. For example, if one of the methods in a class is called `ShowConfirmationDialog`, then you should not give another method a name such as `ShowDialogWarning` or `WarningDialogShow`. The other method should be called `ShowWarningDialog`.

Namespace Names

It is particularly important to choose Namespace names carefully to avoid the risk of ending up with the same name for one of your namespaces as someone else uses. Remember, namespace names are the *only* way that .NET distinguishes names of objects in shared assemblies. Therefore, if you use the same namespace name for your software package as another package, and both packages are used by the same program, problems will occur. Because of this, it's almost always a good idea to create a top-level namespace with the name of

your company and then nest successive namespaces that narrow down the technology, group, or department you are working in or the name of the package for which your classes are intended. Microsoft recommends namespace names that begin with `<CompanyName>.<TechnologyName>`, as in these two examples:

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

Names and Keywords

It is important that the names do not clash with any keywords. In fact, if you attempt to name an item in your code with a word that happens to be a C# keyword, you'll almost certainly get a syntax error because the compiler will assume that the name refers to a statement. However, because of the possibility that your classes will be accessed by code written in other languages, it is also important that you don't use names that are keywords in other .NET languages. Generally speaking, C++ keywords are similar to C# keywords, so confusion with C++ is unlikely, and those commonly encountered keywords that are unique to Visual C++ tend to start with two underscore characters. As with C#, C++ keywords are spelled in lowercase, so if you hold to the convention of naming your public classes and members with Pascal-style names, they will always have at least one uppercase letter in their names, and there will be no risk of clashes with C++ keywords. However, you are more likely to have problems with Visual Basic, which has many more keywords than C# does, and being non-case-sensitive means that you cannot rely on Pascal-style names for your classes and methods.

Check the Microsoft documentation at https://docs.microsoft.com/dotnet/csharp/language-reference/keywords. Here, you find a long list of C# keywords that you shouldn't use with classes and members.

Use of Properties and Methods

One area that can cause confusion regarding a class is whether a particular quantity should be represented by a property or a method. The rules are not hard and strict, but in general you should use a

property if something should look and behave like a variable. (If you're not sure what a property is, see Chapter 3.) This means, among other things, that

- Client code should be able to read its value. Write-only properties are not recommended, so, for example, use a `SetPassword` method, not a write-only `Password` property.
- Reading the value should not take too long. The fact that something is a property usually suggests that reading it will be relatively quick.
- Reading the value should not have any observable and unexpected side effect. Furthermore, setting the value of a property should not have any side effect that is not directly related to the property. Setting the width of a dialog has the obvious effect of changing the appearance of the dialog on the screen. That's fine, because that's obviously related to the property in question.
- It should be possible to set properties in any order. In particular, it is not good practice when setting a property to throw an exception because another related property has not yet been set. For example, to use a class that accesses a database, you need to set `ConnectionString`, `UserName`, and `Password`, and then the author of the class should ensure that the class is implemented such that users can set them in any order.
- Successive reads of a property should give the same result. If the value of a property is likely to change unpredictably, you should code it as a method instead. `Speed`, in a class that monitors the motion of an automobile, is not a good candidate for a property. Use a `GetSpeed` method here; but `Weight` and `EngineSize` are good candidates for properties because they will not change for a given object.

If the item you are coding satisfies all the preceding criteria, it is probably a good candidate for a property. Otherwise, you should use a method.

Use of Fields

The guidelines are pretty simple here. Fields should almost always be private, although in some cases it may be acceptable for constant or read-only fields to be public. Making a field public may hinder your ability to extend or modify the class in the future.

The previous guidelines should give you a foundation of good practices, and you should use them in conjunction with a good object-oriented programming style.

A final helpful note to keep in mind is that Microsoft has been relatively careful about being consistent and has followed its own guidelines when writing the .NET base classes, so a very good way to get an intuitive feel for the conventions to follow when writing .NET code is to simply look at the base classes—see how classes, members, and namespaces are named, and how the class hierarchy works. Consistency between the base classes and your classes will facilitate readability and maintainability.

> **NOTE**
>
> *The new* `ValueTuple` *type contains public fields, whereas the old* `Tuple` *type instead used properties. Microsoft broke one of its own guidelines that's been defined for fields. Because variables of a tuple can be as simple as a variable of an* `int`*, and performance is paramount, it was decided to have public fields for value tuples. It just goes to show that there are no rules without exceptions. Read Chapter 13 for more information on tuples.*

SUMMARY

This chapter examined some of the basic syntax of C#, covering the areas needed to write simple C# programs. We covered a lot of ground, but much of it will be instantly recognizable to developers who are familiar with any C-style language (or even JavaScript).

You have seen that although C# syntax is similar to C++ and Java syntax, there are many minor differences. You have also seen that in

many areas this syntax is combined with facilities to write code very quickly—for example, high-quality string handling facilities. C# also has a strongly defined type system, based on a distinction between value and reference types. Chapters 3 and 4 cover the C# object-oriented programming features.

3
Objects and Types

WHAT'S IN THIS CHAPTER?

- The differences between classes and structs
- Class members
- Expression-bodied members
- Passing values by value and by reference
- Method overloading
- Constructors and static constructors
- Read-only fields
- Enumerations
- Partial classes
- Static classes
- The Object class, from which all other types are derived

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `ObjectsAndTypes`.

The code for this chapter is divided into the following major examples:

- MathSample
- MethodSample
- StaticConstructorSample
- StructsSample
- PassingByValueAndByReference
- OutKeywordSample
- EnumSample
- ExtensionMethods

CREATING AND USING CLASSES

So far, you've been introduced to some of the building blocks of the C# language, including variables, data types, and program flow statements, and you have seen a few very short complete programs containing little more than the `Main` method. What you haven't seen yet is how to put all these elements together to form a longer, complete program. The key to this lies in working with classes—the subject of this chapter. Chapter 4, "Object-Oriented Programming with C#," covers inheritance and features related to inheritance.

NOTE

This chapter introduces the basic syntax associated with classes. However, we assume that you are already familiar with the underlying principles of using classes—for example, that you know what a constructor or a property is. This chapter is largely confined to applying those principles in C# code.

CLASSES AND STRUCTS

Classes and structs are essentially templates from which you can create objects. Each object contains data and has methods to manipulate and access that data. The class defines what data and behavior each particular object (called an *instance*) of that class can contain. For example, if you have a class that represents a customer, it might define fields such as `CustomerID`, `FirstName`, `LastName`, and `Address`, which are used to hold information about a particular customer. It might also define functionality that acts upon the data stored in these fields. You can then instantiate an object of this class to represent one specific customer, set the field values for that instance, and use its functionality:

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

Structs differ from classes because they do not need to be allocated on the heap (classes are reference types and are always allocated on the heap). Structs are value types and are usually stored on the stack. Also, structs cannot derive from a base struct.

You typically use structs for smaller data types for performance reasons. Storing value types on the stack avoids garbage collection. Another use case of structs are interop with native code; the layout of the struct can look the same as native data types.

In terms of syntax, however, structs look very similar to classes; the main difference is that you use the keyword `struct` instead of `class` to declare them. For example, if you wanted all `PhoneCustomer` instances to be allocated on the stack instead of the managed heap, you could write the following:

```
struct PhoneCustomerStruct
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
```

```
}
```

For both classes and structs, you use the keyword `new` to declare an instance. This keyword creates the object and initializes it; in the following example, the default behavior is to zero out its fields:

```
var myCustomer = new PhoneCustomer(); // works for a class
var myCustomer2 = new PhoneCustomerStruct();// works for a
struct
```

In most cases, you use classes much more often than structs. Therefore, this chapter covers classes first and then the differences between classes and structs and the specific reasons why you might choose to use a struct instead of a class. Unless otherwise stated, however, you can assume that code presented for a class works equally well for a struct.

> **NOTE**
>
> *An important difference between classes and structs is that objects of type of class are passed by reference, and objects of type of a struct are passed by value. This is explained later in this chapter in the section "Passing Parameters by Value and by Reference."*

CLASSES

A class contains members, which can be static or instance members. A static member belongs to the class; an instance member belongs to the object. With static fields, the value of the field is the same for every object. With instance fields, every object can have a different value. Static members have the `static` modifier attached.

The kind of members are explained in the following table.

MEMBER	DESCRIPTION
Fields	A field is a data member of a class. It is a variable of a

	type that is a member of a class.
Constants	Constants are associated with the class (although they do not have the `static` modifier). The compiler replaces constants everywhere they are used with the real value.
Methods	Methods are functions associated with a particular class.
Properties	Properties are sets of functions that can be accessed from the client in a similar way to the public fields of the class. C# provides a specific syntax for implementing read and write properties on your classes, so you don't have to use method names that are prefixed with the words `Get` or `Set`. Because there's a dedicated syntax for properties that is distinct from that for normal functions, the illusion of objects as actual things is strengthened for client code.
Constructors	Constructors are special functions that are called automatically when an object is instantiated. They must have the same name as the class to which they belong and cannot have a return type. Constructors are useful for initialization.
Indexers	Indexers allow your object to be accessed the same way as arrays. Indexers are explained in 6, "Operators and Casts."
Operators	Operators, at their simplest, are actions such as + or –. When you add two integers, you are, strictly speaking, using the + operator for integers. C# also allows you to specify how existing operators will work with your own classes (operator overloading). Chapter 6 looks at operators in detail.
Events	Events are class members that allow an object to notify a subscriber whenever something noteworthy happens, such as a field or property of the class changing, or some form of user interaction occurring. The client can have code, known as an event handler, that reacts to the event. Chapter 8, "Delegates, Lambdas, and Events,"

	looks at events in detail.
Destructors	The syntax of destructors or finalizers is similar to the syntax for constructors, but they are called when the CLR detects that an object is no longer needed. They have the same name as the class, preceded by a tilde (~). It is impossible to predict precisely when a finalizer will be called. Finalizers are discussed in Chapter 17, "Managed and Unmanaged Memory."
Types	Classes can contain inner classes. This is interesting if the inner type is only used in conjunction with the outer type.

Let's get into the details of class members.

Fields

Fields are any variables associated with the class. You have already seen fields in use in the `PhoneCustomer` class in the previous example.

After you have instantiated a `PhoneCustomer` object, you can then access these fields using the `object.FieldName` syntax, as shown in this example:

```
var customer1 = new PhoneCustomer();
customer1.FirstName = "Simon";
```

Constants can be associated with classes in the same way as variables. You declare a constant using the `const` keyword. If it is declared as `public`, then it is accessible from outside the class:

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

Readonly Fields

To guarantee that fields of an object cannot be changed, you can

declare fields with the `readonly` modifier. Fields with the `readonly` modifier can be assigned only values from constructors, which is different from the `const` modifier. With the `const` modifier, the compiler replaces the variable with its value everywhere it is used. The compiler already knows the value of the constant. Read-only fields are assigned during runtime from a constructor. Unlike const fields, read-only fields can be instance members. For using a read-only field as a class member, the `static` modifier needs to be assigned to the field.

Suppose that you have a program that edits documents, and for licensing reasons you want to restrict the number of documents that can be opened simultaneously. Assume also that you are selling different versions of the software, and it's possible for customers to upgrade their licenses to open more documents simultaneously. Clearly, this means you can't hard-code the maximum number in the source code. You would probably need a field to represent this maximum number. This field has to be read in—perhaps from some file storage—each time the program is launched. Therefore, your code might look something like this:

```
public class DocumentEditor
{
  private static readonly uint s_maxDocuments;
  static DocumentEditor()
  {
    s_maxDocuments = DoSomethingToFindOutMaxNumber();
  }
}
```

In this case, the field is `static` because the maximum number of documents needs to be stored only once per running instance of the program. This is why the field is initialized in the static constructor. If you had an instance `readonly` field, you would initialize it in the instance constructor(s). For example, presumably each document you edit has a creation date, which you wouldn't want to allow the user to change (because that would be rewriting the past!).

As noted earlier, the date is represented by the class `System.DateTime`. The following code initializes the `_creationTime` field in the constructor using the `DateTime` struct. After initialization of the `Document` class, the creation time cannot be changed anymore:

```
public class Document
{
  private readonly DateTime _creationTime;
  public Document()
  {
    _creationTime = DateTime.Now;
  }
}
```

`_creationDate` and `s_maxDocuments` in the previous code snippets are treated like any other fields, except that they are read-only, which means they can't be assigned outside the constructors:

```
void SomeMethod()
{
  s_maxDocuments = 10; // compilation error here.
MaxDocuments is readonly
}
```

It's also worth noting that you don't have to assign a value to a `readonly` field in a constructor. If you don't assign a value, the field is left with the default value for its particular data type or whatever value you initialized it to at its declaration. That applies to both static and instance `readonly` fields.

It's a good idea *not* to declare fields `public`. If you change a public member of a class, every caller that's using this public member needs to be changed as well. For example, in case you want to introduce a check for the maximum string length with the next version, the public field needs to be changed to a property. Existing code that makes use of the public field must be recompiled for using this property (although the syntax from the caller side looks the same with properties). If you instead change the check within an existing property, the caller doesn't need to be recompiled for using the new version.

It's good practice to declare fields `private` and use properties to access the field, as described in the next section.

Properties

The idea of a *property* is that it is a method or a pair of methods

dressed to look like a field. Let's change the field for the first name from the previous example to a `private` field with the variable name `_firstName`. The property named `FirstName` contains a `get` and `set` accessor to retrieve and set the value of the backing field:

```
class PhoneCustomer
{
  private string _firstName;
  public string FirstName
  {
    get
    {
      return _firstName;
    }
    set
    {
      _firstName = value;
    }
  }
  //...
}
```

The `get` accessor takes no parameters and must return the same type as the declared property. You should not specify any explicit parameters for the `set` accessor either, but the compiler assumes it takes one parameter, which is of the same type again, and which is referred to as `value`.

Let's get into another example with a different naming convention. The following code contains a property called `Age`, which sets a field called `age`. In this example, `age` is referred to as the backing variable for the property `Age`:

```
private int age;
public int Age
{
  get
  {
    return age;
  }
  set
  {
    age = value;
  }
}
```

Note the naming convention used here. You take advantage of C#'s case sensitivity by using the same name—Pascal-case for the public property, and camel-case for the equivalent private field if there is one. In earlier .NET versions, this naming convention was preferred by Microsoft's C# team. Recently they switched the naming convention to prefix field names by an underscore. This provides an extremely convenient way to identify fields in contrast to local variables.

> **NOTE**
>
> *Microsoft teams use either one or the other naming convention. For using private members of types, .NET doesn't have strict naming conventions. However, within a team the same convention should be used. The .NET Core team switched to using an underscore to prefix fields, which is the convention used in this book in most places (see* https://github.com/dotnet/corefx/blob/master/Documentation/coding-guidelines/coding-style.md*).*

Expression-Bodied Property Accessors

With C# 7, you can also write property accessors as expression-bodied members. For example, the previously shown property `FirstName` can be written using `=>`. This new feature reduces the need to write curly brackets, and the `return` keyword is omitted with the `get` accessor.

```
private string _firstName;
public string FirstName
{
  get => _firstName;
  set => _firstName = value;
}
```

When you use expression-bodied members, the implementation of the property accessor can be made up of only a single statement.

Auto-Implemented Properties

If there isn't going to be any logic in the properties `set` and `get`, then auto-implemented properties can be used. Auto-implemented properties implement the backing member variable automatically. The code for the earlier `Age` example would look like this:

```
public int Age { get; set; }
```

The declaration of a private field is not needed. The compiler creates this automatically. With auto-implemented properties, you cannot access the field directly as you don't know the name the compiler generates. If all you need to do with a property is read and write a field, the syntax for the property when you use auto-implemented properties is shorter than when you use expression-bodied property accessors.

By using auto-implemented properties, validation of the property cannot be done at the property set. Therefore, with the `Age` property you could not have checked to see if an invalid age is set.

Auto-implemented properties can be initialized using a *property initializer*:

```
public int Age { get; set; } = 42;
```

Access Modifiers for Properties

C# allows the `set` and `get` accessors to have differing access modifiers. This would allow a property to have a public `get` and a private or protected `set`. This can help control how or when a property can be set. In the following code example, notice that the `set` has a private access modifier but the `get` does not. In this case, the `get` takes the access level of the property. One of the accessors must follow the access level of the property. A compile error is generated if the `get` accessor has the `protected` access level associated with it because that would make both accessors have a different access level from the property.

```
public string Name
{
  get => _name;
  private set => _name = value;
}
```

Different access levels can also be set with auto-implemented properties:

```
public int Age { get; private set; }
```

> **NOTE**
>
> *Some developers may be concerned that the previous sections have presented a number of situations in which standard C# coding practices have led to very small functions—for example, accessing a field via a property instead of directly. Will this hurt performance because of the overhead of the extra function call? The answer is no. There's no need to worry about performance loss from these kinds of programming methodologies in C#. Recall that C# code is compiled to IL, then JIT compiled at runtime to native executable code. The JIT compiler is designed to generate highly optimized code and will ruthlessly inline code as appropriate (in other words, it replaces function calls with inline code). A method or property whose implementation simply calls another method or returns a field will almost certainly be inlined.*
>
> *Usually you do not need to change the inlining behavior, but you have some control to inform the compiler about inlining. Using the attribute* `MethodImpl`*, you can define that a method should not be inlined (*`MethodImplOptions.NoInlining`*), or inlining should be done aggressively by the compiler (*`MethodImplOptions.AggressiveInlining`*). With properties, you need to apply this attribute directly to the* `get` *and* `set` *accessors. Attributes are explained in detail in Chapter 16, "Reflection, Metadata, and Dynamic Programming."*

Read-Only Properties

It is possible to create a read-only property by simply omitting the `set` accessor from the property definition. Thus, to make `Name` a read-only property, you would do the following:

```
private readonly string _name;
public string Name
{
  get => _name;
}
```

Declaring the field with the `readonly` modifier only allows initializing the value of the property in the constructor.

> **WARNING**
>
> *Similar to creating read-only properties it is also possible to create a write-only property. Write-only properties can be created by omitting the* `get` *accessor. However, this is regarded as poor programming practice because it could be confusing to authors of client code. In general, it is recommended that if you are tempted to do this, you should use a method instead.*

Auto-Implemented Read-Only Properties

C# offers a simple syntax with auto-implemented properties to create read-only properties that access read-only fields. These properties can be initialized using property initializers.

```
public string Id { get; } = Guid.NewGuid().ToString();
```

Behind the scenes, the compiler creates a read-only field and also a property with a `get` accessor to this field. The code from the initializer moves to the implementation of the constructor and is invoked before the constructor body is called.

Read-only properties can also explicitly be initialized from the constructor as shown with this code snippet:

```
public class Person
{
  public Person(string name) => Name = name;

  public string Name { get; }
}
```

Expression-Bodied Properties

Since C# 6, properties with just a `get` accessor also can be implemented using expression-bodied properties. Similar to expression-bodied methods, expression-bodied properties don't need curly brackets and return statements. Expression-bodied properties are properties with the `get` accessor, but you don't need to write the `get` keyword. Instead of writing the `get` keyword, the code you previously implemented in the get accessor can now follow the lambda operator. With the `Person` class, the `FullName` property is implemented using an expression-bodied property and returns with this property the values of the `FirstName` and `LastName` properties combined (code file `ClassesSample/Program.cs`):

```
public class Person
{
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
  public string FirstName { get; }
  public string LastName { get; }
  public string FullName => $"{FirstName} {LastName}";
}
```

Immutable Types

If a type contains members that can be changed, it is a *mutable* type. With the `readonly` modifier, the compiler complains if the state is changed. The state can be initialized only in the constructor. If an object doesn't have any members that can be changed—it has only `readonly` members—it is an *immutable* type. The content can be set only at initialization time. This is extremely useful with multithreading, as multiple threads can access the same object with the information and it can never change. Because the content can't change, synchronization is not necessary.

An example of an immutable type is the `String` class. This class does not define any member that is allowed to change its content. Methods such as `ToUpper` (which changes the string to uppercase) always return a new string, but the original string passed to the constructor remains

unchanged.

> **NOTE**
>
> *.NET also offers immutable collections. These collection classes are covered in Chapter 11, "Special Collections."*

Anonymous Types

Chapter 2, "Core C#," discusses the `var` keyword in reference to implicitly typed variables. When you use `var` with the `new` keyword, you can create anonymous types. An anonymous type is simply a nameless class that inherits from `object`. The definition of the class is inferred from the initializer, just as with implicitly typed variables.

For example, if you need an object containing a person's first, middle, and last name, the declaration would look like this:

```
var captain = new
{
  FirstName = "James",
  MiddleName = "T",
  LastName = "Kirk"
};
```

This would produce an object with `FirstName`, `MiddleName`, and `LastName` properties. If you were to create another object that looked like this:

```
var doctor = new
{
  FirstName = "Leonard",
  MiddleName = string.Empty,
  LastName = "McCoy"
};
```

then the types of `captain` and `doctor` are the same. You could set `captain = doctor`, for example. This is only possible if all the properties match.

The names for the members of anonymous types can be inferred—if the values that are being set come from another object. This way, the initializer can be abbreviated. If you already have a class that contains the properties `FirstName`, `MiddleName`, and `LastName` and you have an instance of that class with the instance name `person`, then the `captain` object could be initialized like this:

```
var captain = new
{
  person.FirstName,
  person.MiddleName,
  person.LastName
};
```

The property names from the `person` object are inferred in the new object named `captain`, so the object named `captain` has `FirstName`, `MiddleName`, and `LastName` properties.

The actual type name of anonymous types is unknown; that's where the name comes from. The compiler "makes up" a name for the type, but only the compiler is ever able to make use of it. Therefore, you can't and shouldn't plan on using any type reflection on the new objects because you won't get consistent results.

Methods

Note that official C# terminology makes a distinction between functions and methods. In C# terminology, the term "function member" includes not only methods, but also other nondata members of a class or struct. This includes indexers, operators, constructors, destructors, and—perhaps somewhat surprisingly—properties. These are contrasted with data members: fields, constants, and events.

Declaring Methods

In C#, the definition of a method consists of any method modifiers (such as the method's accessibility), followed by the type of the return value, followed by the name of the method, followed by a list of input arguments enclosed in parentheses, followed by the body of the method enclosed in curly braces:

```
[modifiers] return_type MethodName([parameters])
{
  // Method body
}
```

Each parameter consists of the name of the type of the parameter, and the name by which it can be referenced in the body of the method. Also, if the method returns a value, a return statement must be used with the return value to indicate each exit point, as shown in this example:

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

If the method doesn't return anything, specify a return type of `void` because you can't omit the return type altogether; and if it takes no arguments, you still need to include an empty set of parentheses after the method name. In this case, including a return statement is optional—the method returns automatically when the closing curly brace is reached.

Expression-Bodied Methods

If the implementation of a method consists just of one statement, C# gives a simplified syntax to method definitions: *expression-bodied methods*. You don't need to write curly brackets and the `return` keyword with the new syntax. The operator `=>` is used to distinguish the declaration of the left side of this operator to the implementation that is on the right side.

The following example is the same method as before, `IsSquare`, implemented using the expression-bodied method syntax. The right side of the lambda operator defines the implementation of the method. Curly brackets and a return statement are not needed. What's returned is the result of the statement, and the result needs to be of the same type as the method declared on the left side, which is a `bool` in this code snippet:

```
public bool IsSquare(Rectangle rect) => rect.Height ==
rect.Width;
```

Invoking Methods

The following example illustrates the syntax for definition and instantiation of classes, and definition and invocation of methods. The class `Math` defines instance and static members (code file `MathSample/Math.cs`):

```
public class Math
{
  public int Value { get; set; }
  public int GetSquare() => Value * Value;
  public static int GetSquareOf(int x) => x * x;
  public static double GetPi() => 3.14159;
}
```

The `Program` class makes use of the `Math` class, calls static methods, and instantiates an object to invoke instance members (code file `MathSample/Program.cs`);

```
using System;
namespace MathSample
{
  class Program
  {
    static void Main()
    {
      // Try calling some static functions.
      Console.WriteLine($"Pi is {Math.GetPi()}");
      int x = Math.GetSquareOf(5);
      Console.WriteLine($"Square of 5 is {x}");
      // Instantiate a Math object
      var math = new Math(); // instantiate a reference type
      // Call instance members
      math.Value = 30;
      Console.WriteLine($"Value field of math variable
contains {math.Value}");
      Console.WriteLine($"Square of 30 is
{math.GetSquare()}");
    }
  }
}
```

Running the `MathSample` example produces the following results:

```
Pi is 3.14159
Square of 5 is 25
```

```
Value field of math variable contains 30
Square of 30 is 900
```

As you can see from the code, the `Math` class contains a property that contains a number, as well as a method to find the square of this number. It also contains two static methods: one to return the value of pi and one to find the square of the number passed in as a parameter.

Some features of this class are not really good examples of C# program design. For example, `GetPi` would usually be implemented as a `const` field, but following good design would mean using some concepts that have not yet been introduced.

Method Overloading

C# supports method overloading—several versions of the method that have different signatures (that is, the same name but a different number of parameters and/or different parameter data types). To overload methods, simply declare the methods with the same name but different numbers of parameter types:

```
class ResultDisplayer
{
  public void DisplayResult(string result)
  {
    // implementation
  }

  public void DisplayResult(int result)
  {
    // implementation
  }
}
```

It's not just the parameter types that can differ; the number of parameters can differ too, as shown in the next example. One overloaded method can invoke another:

```
class MyClass
{
  public int DoSomething(int x)
  {
    return DoSomething(x, 10); // invoke DoSomething with two
parameters
```

```
  }

  public int DoSomething(int x, int y)
  {
    // implementation
  }
}
```

NOTE

With method overloading, it is not sufficient to only differ overloads by the return type. It's also not sufficient to differ by parameter names. The number of parameters and/or types needs to differ.

Named Arguments

Invoking methods, the variable name need not be added to the invocation. However, if you have a method signature like the following to move a rectangle

```
public void MoveAndResize(int x, int y, int width, int
height)
```

and you invoke it with the following code snippet, it's not clear from the invocation what numbers are used for what:

```
r.MoveAndResize(30, 40, 20, 40);
```

You can change the invocation to make it immediately clear what the numbers mean:

```
r.MoveAndResize(x: 30, y: 40, width: 20, height: 40);
```

Any method can be invoked using named arguments. You just need to write the name of the variable followed by a colon and the value passed. The compiler gets rid of the name and creates an invocation of the method just like the variable name would not be there—so there's no difference within the compiled code. C# 7.2 allows for non-trailing named arguments. When you use earlier C# versions, you need to

supply names for all arguments after using the first named argument.

You can also change the order of variables this way, and the compiler rearranges it to the correct order. A big advantage you get with named arguments is shown in the next section with optional arguments.

Optional Arguments

Parameters can also be optional. You must supply a default value for optional parameters, which must be the last ones defined:

```
public void TestMethod(int notOptionalNumber, int
optionalNumber = 42)
{
  Console.WriteLine(optionalNumber + notOptionalNumber);
}
```

This method can now be invoked using one or two parameters. Passing one parameter, the compiler changes the method call to pass `42` with the second parameter.

```
TestMethod(11);
TestMethod(11, 22);
```

> **NOTE**
>
> *Because the compiler changes methods with optional parameters to pass the default value, the default value should never change with newer versions of the assembly. With a change of the default value in a newer version, if the caller is in a different assembly that is not recompiled, it would have the older default value. That's why you should have optional parameters only with values that never change. In case the calling method is always recompiled when the default value changes, this is not an issue.*

You can define multiple optional parameters, as shown here:

```
public void TestMethod(int n, int opt1 = 11, int opt2 = 22,
int opt3 = 33)
{
```

```
    Console.WriteLine(n + opt1 + opt2 + opt3);
}
```

This way, the method can be called using 1, 2, 3, or 4 parameters. The first line of the following code leaves the optional parameters with the values `11`, `22`, and `33`. The second line passes the first three parameters, and the last one has a value of `33`:

```
TestMethod(1);
TestMethod(1, 2, 3);
```

With multiple optional parameters, the feature of named arguments shines. Using named arguments, you can pass any of the optional parameters—for example, this example passes just the last one:

```
TestMethod(1, opt3: 4);
opt3: 4
```

> **WARNING**
>
> *Pay attention to versioning issues when using optional arguments. One issue is to change default values in newer versions; another issue is to change the number of arguments. It might look tempting to add another optional parameter as it is optional anyway. However, the compiler changes the calling code to fill in all the parameters, and that's the reason earlier compiled callers fail if another parameter is added later on.*

Variable Number of Arguments

Using optional arguments, you can define a variable number of arguments. However, there's also a different syntax that allows passing a variable number of arguments—and this syntax doesn't have versioning issues.

Declaring the parameter of type array—the sample code uses an `int` array—and adding the `params` keyword, the method can be invoked using any number of `int` parameters.

```
public void AnyNumberOfArguments(params int[] data)
{
  foreach (var x in data)
  {
    Console.WriteLine(x);
  }
}
```

> **NOTE**
>
> *Arrays are explained in detail in Chapter 7, "Arrays."*

As the parameter of the method `AnyNumberOfArguments` is of type `int[]`, you can pass an `int` array, or because of the `params` keyword, you can pass one or any number of `int` values:

```
AnyNumberOfArguments(1);
AnyNumberOfArguments(1, 3, 5, 7, 11, 13);
```

If arguments of different types should be passed to methods, you can use an `object` array:

```
public void AnyNumberOfArguments(params object[] data)
{
  // ...
```

Now it is possible to use any type calling this method:

```
AnyNumberOfArguments("text", 42);
```

If the `params` keyword is used with multiple parameters that are defined with the method signature, `params` can be used only once, and it must be the last parameter:

```
Console.WriteLine(string format, params object[] arg);
```

Now that you've looked at the many aspects of methods, let's get into constructors, which are a special kind of methods.

Constructors

The syntax for declaring basic constructors is a method that has the same name as the containing class and that does not have any return type:

```
public class MyClass
{
  public MyClass()
  {
  }

  // rest of class definition
}
```

It's not necessary to provide a constructor for your class. We haven't supplied one for any of the examples so far in this book. In general, if you don't supply any constructor, the compiler generates a default one behind the scenes. It will be a very basic constructor that initializes all the member fields by zeroing them out (`null` reference for reference types, zero for numeric data types, and false for `bool`s). Often, that is adequate; if not, you need to write your own constructor.

Constructors follow the same rules for overloading as other methods—that is, you can provide as many overloads to the constructor as you want, provided they are clearly different in signature:

```
public MyClass() // zeroparameter constructor
{
  // construction code
}

public MyClass(int number) // another overload
{
  // construction code
}
```

However, if you supply any constructors that take parameters, the compiler does not automatically supply a default one. This is done only if you have not defined any constructors at all. In the following example, because a one-parameter constructor is defined, the compiler assumes that this is the only constructor you want to be available, so it does not implicitly supply any others:

```
public class MyNumber
```

```
{
  private int _number;
  public MyNumber(int number)
  {
    _number = number;
  }
}
```

If you now try instantiating a `MyNumber` object using a no-parameter constructor, you get a compilation error:

```
var numb = new MyNumber(); // causes compilation error
```

Note that it is possible to define constructors as private or protected, so that they are invisible to code in unrelated classes too:

```
public class MyNumber
{
  private int _number;
  private MyNumber(int number) // another overload
  {
    _number = number;
  }
}
```

This example hasn't actually defined any public, or even any protected, constructors for `MyNumber`. This would actually make it impossible for `MyNumber` to be instantiated by outside code using the `new` operator (though you might write a public static property or method in `MyNumber` that can instantiate the class). This is useful in two situations:

- If your class serves only as a container for some static members or properties, and therefore should never be instantiated. With this scenario, you can declare the class with the modifier `static`. With this modifier the class can contain only static members and cannot be instantiated.
- If you want the class to only ever be instantiated by calling a static member function (this is the so-called factory pattern approach to object instantiation). An implementation of the Singleton pattern is shown in the following code snippet.

```
public class Singleton
{
```

```
  private static Singleton s_instance;
  private int _state;
  private Singleton(int state)
  {
    _state = state;
  }
  public static Singleton Instance
  {
    get => s_instance ?? (s_instance = new Singleton(42);
  }
}
```

The `Singleton` class contains a private constructor, so you can instantiate it only within the class itself. To instantiate it, the static property `Instance` returns the field `s_instance`. If this field is not yet initialized (`null`), a new instance is created by calling the instance constructor. For the null check, the coalescing operator is used. If the left side of this operator is null, the right side of this operator is processed and the instance constructor invoked.

> **NOTE**
>
> *The coalescing operator is explained in detail in Chapter 6.*

Expression Bodies with Constructors

If the implementation of a constructor consists of a single expression, the constructor can be implemented with an expression-bodied implementation:

```
public class Singleton
{
  private static Singleton s_instance;
  private int _state;
  private Singleton(int state) => _state = state;

  public static Singleton Instance =>
    s_instance ?? (s_instance = new Singleton(42);
}
```

Calling Constructors from Other Constructors

You might sometimes find yourself in the situation where you have several constructors in a class, perhaps to accommodate some optional parameters for which the constructors have some code in common. For example, consider the following:

```
class Car
{
  private string _description;
  private uint _nWheels;

  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }

  public Car(string description)
  {
    _description = description;
    _nWheels = 4;
  }
  // ...
}
```

Both constructors initialize the same fields. It would clearly be neater to place all the code in one location. C# has a special syntax known as a *constructor initializer* to enable this:

```
class Car
{
  private string _description;
  private uint _nWheels;
  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }
  public Car(string description): this(description, 4)
  {
  }
  // ...
```

In this context, the `this` keyword simply causes the constructor with the nearest matching parameters to be called. Note that any constructor initializer is executed before the body of the constructor.