# C# Succinctly

by Joe Mayo

Syncfusion | Technology Resource Portal

# C# Succinctly

**By**
**Joe Mayo**

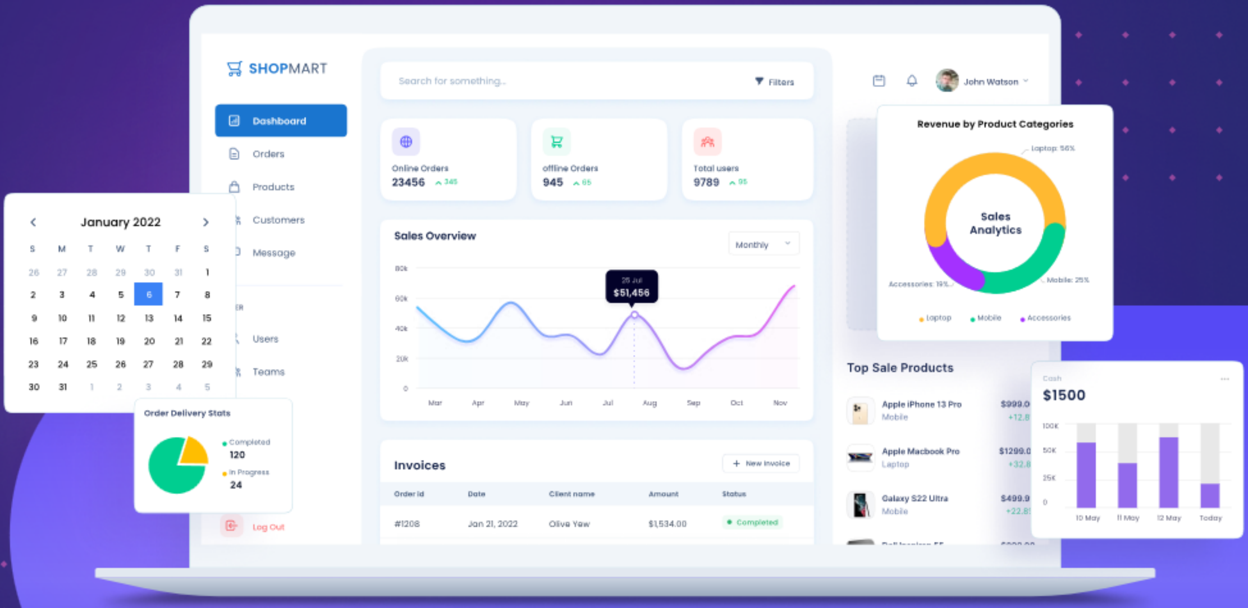
Foreword by Daniel Jebaraj

Syncfusion
Deliver innovation with ease

**Technical Reviewer:** Stephen Haunts
**Copy Editor:** Ben Ball
**Acquisitions Coordinator:** Hillary Bowling, marketing coordinator, Syncfusion, Inc.
**Proofreader:** Graham High, content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

# POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

G2 ★★★★½ 4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

IBM SIEMENS intel VISA

Syncfusion

# Table of Contents

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

## Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

# About the Author

Joe Mayo is an author, a consultant at Mayo Software, LLC, and an instructor who specializes in Microsoft .NET technology. Joe has written several books, including *C# Unleashed* (Sams) and *LINQ Programming* (McGraw-Hill), and coauthored *ASP.NET 2.0 MVP Hacks and Tips* (Wrox). His articles have been published in CODE Magazine and the online publications Inform IT and C# Station.

Joe is a regular presenter on .NET topics and has received multiple Microsoft Visual C# MVP awards. His open source project, LINQ to Twitter, is hosted on GitHub, and you can read his blog at Geeks with Blogs. You can find Joe on Twitter as @JoeMayo.

# Chapter 1  Introducing C# and .NET

Welcome to *C# Succinctly*. True to the *Succinctly* series concept, this book is very focused on a single topic: the C# programming language. I might briefly mention some technologies that you can write with C# or explain how a feature fits into those technologies, but the whole of this book is about helping you become familiar with C# syntax.

In this chapter, I'll start with some introductory information and then jump straight into a simple C# program.

## What can I do with C#?

C# is a general purpose, object-oriented, component-based programming language. As a general purpose language, you have a number of ways to apply C# to accomplish many different tasks. You can build web applications with ASP.NET, desktop applications with Windows Presentation Foundation (WPF), or build mobile applications for Windows Phone. Other applications include code that runs in the cloud via Windows Azure, and iOS, Android, and Windows Phone support with the Xamarin platform. There might be times when you need a different language, like C or C++, to communicate with hardware or real-time systems. However, from a general programming perspective, you can do a lot with C#.

## What is .NET?

.NET is a platform that includes languages, a runtime, and framework libraries, allowing developers to create many types of applications. C# is one of the .NET languages, which also includes Visual Basic, F#, C++, and more.

The runtime is more formally named the Common Language Runtime (CLR). Programming languages that target the CLR compile to an Intermediate Language (IL). The CLR itself is a virtual machine that runs IL and provides many services such as memory management, garbage collection, exception management, security, and more.

The Framework Class Library (FCL) is a set of reusable code that provides both general services and technology-specific platforms. The general services include essential types such as collections, cryptography, networking, and more. In addition to general classes, the FCL includes technology-specific platforms like ASP.NET, WPF, web services, and more. The value the FCL offers is to have common components available for reuse, saving time and money without needing to write that code yourself.

There's a huge ecosystem of open-source and commercial software that relies on and supports .NET. If you visit CodePlex, GitHub, or any other open-source code repository site, you'll see a multitude of projects written in C#. Commercial offerings include tools and services that help you build code, manage systems, and offer applications. Syncfusion is part of this ecosystem, offering reusable components for many of the .NET technologies I have mentioned.

## Writing, Running, and Deploying a C# Program

The previous section described plenty of great things you can do with C#, but most of them are so detailed that they require their own book. To stay focused on the C# programming language, the code in this book will be for the console application. A console application runs on the command line, which you'll learn about in this section. You can write your code with any editor, but this book uses Visual Studio.

***Note: The code samples in this book can be downloaded at https://bitbucket.org/syncfusiontech/c-succinctly.***

### Starting a New Program

You'll need an editor or Integrated Development Environment (IDE) to write code. Microsoft offers Visual Studio (VS), which is available via Community Edition as a free download for training and individual purposes (https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx). There are other development tools, but you can also use any editor, including Notepad. Notepad++ is another editor that does syntax highlighting, but there are many more available. Essentially, you just need the ability to type a text document. Pick your editor or IDE of choice and it will work for all programs in this book.

***Note: You need to use Visual Studio 2015 to compile the samples in this book.***

To get started, we need a program to run. In VS, select **File** > **New** > **Project**, then select **Installed** > **Templates** > **Visual C#** in the tree on the left, and finally select the **Console Application** project type. Name the solution **Chapter01**, name the project **Greetings**, set the location to your preference, and click **OK**. This will create a new solution for you. Delete the **Program.cs** file and add a **Greetings.cs** file. In any text editor, just create a file named **Greetings.cs**. The following is a C# program that prints a greeting to the command line.

```
using System;

class Greetings
{
    static void Main()
    {
        Console.WriteLine("Greetings!");
    }
}
```

*Code Listing 1*

The `class` is a container for code, defining a type, named `Greetings`. A `class` has members and this example shows a method member named `Main`. A method is similar to functions and procedures in other programming languages. For desktop application types, like console or WPF, naming a method `Main` tells the C# compiler where the program begins executing. Both the **`Greetings`** class and **`Main`** method have curly braces, referred to as a block, indicating beginning and ending scope.

The **`void`** keyword isn't a type; it indicates that a method does not return a value. For **`Main`**, you can replace **`void`** with **`int`**, meaning that the program has a return code. This number can be used by command-line shell tools to evaluate the conditions under which the program ended. It is unique to each program and specified by you. Later, you'll learn more about methods and return values.

The **`static`** modifier indicates that there is only ever one instance of a **`Greetings`** class that has that **`Main`** method—it is the **`static`** instance. **`Main`** must be **`static`**, but other methods can omit **`static`**, which makes them instance members. This means that you can have many copies of a class or instance with their own method.

Since a program only needs a single **`Main`** method, **`static`** makes sense. A program that manages customers might have a **`Customer`** class and you would need multiple instances to represent each **`Customer`**. You'll see examples of instantiating classes in later chapters of this book.

Inside the **`Main`** method is a statement that prints words to the command line. The words, enclosed in double quotes, are a string. That string is passed to the **`WriteLine`** method, which writes that string to the command line and causes it to move to the next line. **`WriteLine`** is a method that belongs to a class named **`Console`**. You see in this example, just like the **`Greetings`** class, the **`Console`** is a class too. This **`Console`** class belongs to the **`System`** namespace, which is why the **`using`** clause appears at the top of the file, allowing us to use that **`Console`** class.

The code begins with a **`using`** clause for the **`System`** namespace. The FCL is grouped into namespaces to keep code organized and avoid clashes between identically named types. This **`using`** clause allows us to use the code in the **`System`** namespace, which we're doing with the **`Console`** class. Without that, the compiler doesn't know what **`Console`** means or how to find it, but now C# knows that we're using the **`System.Console`** class.

## Namespaces and Code Organization

There are various ways to organize code and the choice should be based on the standards of your team and the nature of the project you're building. One of the common ways to organize code is with the C# namespace feature. Here's a hierarchical description of where namespaces fit into the overall structure of a program:

- Namespace
  - Type
    - Type Members

Out of this hierarchy, the namespace is optional, as demonstrated in the previous program where the **`Greetings`** class was not contained in a namespace. This means **`Greetings`** is a member of the **`global`** namespace. You should avoid this practice as it opens the possibility for other developers working with your code to write their own **`Greetings`** class in the same namespace, which will cause errors because the C# compiler can't figure out which **`Greetings`** class to use. While **Greetings** might seem unique and unlikely, think of common names, such as **File**, **Math**, or **Window**, that would cause problems. The following program uses namespaces appropriately.

```
using static System.Math;

namespace Syncfusion
{
    public class Calc
    {
        public static double Pythagorean(double a, double b)
        {
            double cSquared = Pow(a, 2) + Pow(b, 2);
            return Sqrt(cSquared);
        }
    }
}
```

*Code Listing 2*

The **Calc** class is a member of the **Syncfusion** namespace. The **Pythagorean** method is a member of the **Calc** class. A method is a block of code with a name, parameters, and return value that you can call from other code. This follows the namespace, class, member organization.

**System** is a namespace in the FCL and **Math** is a class in the **System** namespace. The **using static** clause allows the code to use static members of the **Math** class without full qualification. Instead of writing **Math.Pow(a, 2)**, which squares the value of **a**, you can use the shorthand syntax in the **Pythagorean** method. The **Pythagorean** method uses **Math.Sqrt**, which provides square root, similarly. The following sample shows how you can use this code.

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
        }
    }
}
```

*Code Listing 3*

The `Main` method calls the `Pythagorean` method of the `Calc` class, passing arguments `2` and `3` and receiving a result in `hypotenuse`. Since `Calc` is in the `Syncfusion` namespace, the code adds a `using` clause for `Syncfusion` to the top of the file. Had the code not included that `using` clause, `Main` would have been required to use the fully qualified name, `Syncfusion.Calc.Pythagorean`.

Another feature of the previous program is the namespace alias, `Crypto`. This syntax allows you to use a shorthand syntax when you need to fully qualify a namespace, but want to reduce syntax in your code. If there had been another `Cryptography` namespace used in the same code, though not in this listing, full qualification would have been necessary. `Crypto` is the alias for `System.Security.Cryptography` and `Main` uses that alias in `Crypto.AesManaged` to make the code more readable.

## Running the Program

The rest of this chapter returns to the previous Greetings program in this chapter.

Now the program is written and you want to continue by compiling the program and running it. You'll want to save this file with the name **Greetings.cs**. The name isn't necessarily important, but by convention should be meaningful and is often the same name as a class it contains. You're allowed to put multiple classes in the same file, but it's easier to find a class later if it is alone in its own file of the same name. C# files have a .cs extension.

In VS, click the green **Start** arrow on the toolbar and it will build and run the program. The program runs and stops so quickly that you won't see the command-line output, so you can press **Ctrl + F5** to make the command line stay open. This book uses Visual Studio 2015, but Syncfusion has published *Visual Studio 2013 Succinctly*, which explains many features that are still valid in Visual Studio 2015. In the meantime, I'm going to show you how to use the C# compiler directly—the benefit being that you see what the IDE is doing for you.

***Tip: Adding `Console.ReadKey();` as the last line in `Main` makes the command line stop and wait for a key press.***

Minimally, you need the .NET Framework installed on your machine, which is free for commercial as well as non-commercial use. If you installed VS, you already have the .NET Framework. Otherwise, download it from http://www.microsoft.com/en-us/download/details.aspx?id=30653 and install it. This link is for .NET Framework 4.5, but any future version should work fine.

Once .NET is installed, open Windows Explorer and do a search for the C# compiler, **csc.exe**. Since I'm using Visual Studio 2015 for the examples in this book, the C# 6 compiler on my machine is located at **C:\Program Files (x86)\MSBuild\14.0\Bin**, but yours may differ.

Next, make sure the C# compiler is in your path. Open your **System Properties** window. As of this writing, I'm on Windows 8.1 and found it via selecting **Control Panel** > **System and Security** > **System**, and then clicking **Advanced System Settings**. Select the **Advanced** tab and click the **Environment Variables** button. In the **System variables** list, select **Path**, and click **Edit**. You should see several paths separated by semicolons. At the end of that path, add your C# compiler's path that you found with the Windows Explorer search and make sure it's separated from the previous paths with a semicolon. Close out of all these windows when you're done setting the path.

Now that you have the .NET Framework installed and have the path to the C# compiler set, you can build the program that you typed in the previous example. First, open a command prompt window. On my system, I can do this by pressing the **Windows logo key + R**, typing **cmd.exe** in the **Run** dialog, and then clicking **OK**. If you've never used a command line, it's a good idea to open your favorite search engine and look for a tutorial. Alternatively, it might be good to learn PowerShell; Syncfusion has a book on it titled *PowerShell Succinctly*. In the meantime, navigate to the directory where you saved **Greetings.cs**. You can type **cd\your\path\there** and press **Enter** to get there. You can verify you're in the right location by typing **dir** to see what files reside in the current directory.

To compile the program, type **csc Greetings.cs**. If you see compiler errors, go back to Code Listing 1 and make sure you've typed the code exactly as it is there and then recompile.

***Tip: Use a space separated list to compile multiple files; e.g., csc.exe file1.cs file2.cs. For C# compiler help, type csc.exe /help.***

Now type **dir** and you'll see a new file named **Greetings.exe**. This is an executable assembly. In .NET, an assembly is a unit of identity, execution, and deployment, which is why it's not just called a file. For the purposes of this book, you won't be involved with all the nuances of assemblies, but it's an encompassing term that includes both executable (.exe) and library (.dll) files.

Now type **Greetings.exe** and press **Enter**. The program will print **`Greetings!`** on the command line. Then you'll see a new command-line prompt, meaning that the program ended. This came from the **`Console.WriteLine`** statement in the **`Main`** method. When the **`Main`** method finishes executing, the program finishes too.

## Deploying the Program

.NET uses XCopy deployment, which means that you only need to copy the assembly to anywhere you want it to go. The one caveat is that whatever machine you run the program on must also have the .NET CLR installed. Installing VS or the .NET Framework automatically installs the CLR. Also, you can only install the .NET Framework Runtime, which doesn't include development tools, to a machine where you only want to run a C# program but not perform any development tasks. In practical terms, most Windows systems already have .NET installed from the original installation and it is kept up-to-date via Windows Update.

Whenever you run the program, Windows looks at the executable, determines that it's a .NET assembly, loads the CLR, and then gives that assembly to the CLR to run. From the users' perspective, the CLR behavior is behind the scenes; the program appears like any other program when they run the executable.

## Summary

This chapter included a couple broader takeaways regarding how C# fits into the .NET Framework ecosystem and how to create a C# program. Remember that C# is a programming language, but it builds programs that use the FCL to run applications managed by the CLR. What this gives you is the ability to compile programs into assemblies that can be deployed and run on any machine that supports the CLR. The program entry point is the **`Main`** method. You can use any editor or an IDE like Visual Studio to write your code. To run a program, press **F5** in VS or compile with **`csc.exe`** on the command line. To deploy, copy the program to a machine with the CLR installed. In the next chapter, you’ll learn more about how to code logic in C# using expressions and statements.

# Chapter 2 Coding Expressions and Statements

In Chapter 1, you saw how to write, compile, and execute a C# program. The example program had a single statement in the **`Main`** method. In this chapter, you'll learn how to write more statements and add logic to your program. For efficiency, many of the examples in the rest of the book are snippets, but you can still add these statements inside of a **`Main`** method to compile and get a better feel for C# syntax. There will be plenty of complete programs too.

## Writing Simple Statements

By combining language operators and syntax, you can build expressions and statements in C#. Here are a few examples of simple C# statements.

```
int count = 7;
char keyPressed = 'Q';
string title = "Weekly Report";
```

*Code Listing 4*

Each of the examples in the previous code listing have common syntactical elements: type, variable identifier, assignment operator, value, and statement completion. The types are **`int`**, **`char`**, and **`string`**, which represent a number, a character, and a sequence of characters respectively. These are a few of the several built-in types that C# offers. Variables are a name that can be used in later code. The **=** operator assigns the right-hand side of the expression to the left-hand side. Each statement ends with a semicolon.

The previous example showed how to declare a variable and perform assignment at the same time, but that isn't necessarily required. As long as you declare a variable before trying to use it, you'll be okay. Here's a separate declaration.

```
string title;
```

*Code Listing 5*

And the variable's later assignment.

```
title = "Weekly Report";
```

*Code Listing 6*

***Note: C# is case sensitive, so "title" and "Title" are two separate variable names.***

## Overview of C# Types and Operators

C# is a strongly typed language, meaning that the compiler won't implicitly convert between incompatible types. For example, you can't assign a **`string`** to an **`int`** or an **`int`** to a **`string`**—at least, not implicitly. The following code will not compile.

```
int total = "359";
string message = 7;
```

*Code Listing 7*

The "**`359`**" with double quotes is a string, and the **`7`** without quotes is an **`int`**. While you can't perform conversions implicitly, there are ways to do this explicitly. For example, you'll often receive text input from a user that should be an **`int`** or another type. The following code listing shows a couple examples of how to perform such tasks explicitly.

```
int total = int.Parse("359");
string message = 7.ToString();
```

*Code Listing 8*

In the previous listing, **`Parse`** will convert the string to an **`int`** if the string represents a valid **`int`**. Calling **`ToString`** on any value will always produce a **`string`** that will compile.

In addition to the previous conversion examples, C# has a cast operator that lets you convert between types that allow explicit conversions. Let's say you have a **`double`**, which is a 64-bit floating point type, and want to assign that to an **`int`**, which is a 32-bit whole number. You could cast it like this:

```
double preciseLength = 5.61;
int roundedLength = (int)preciseLength;
```

*Code Listing 9*

Without the cast operator, you would receive a compiler error because a **`double`** is not an **`int`**. Essentially, the C# compiler is protecting you from shooting yourself in the foot because assigning a **`double`** to an **`int`** means that you lose precision. In the previous example, **`roundedLength`** becomes **`5`**. Using the cast operator allows you to tell the C# compiler that you know this operation could be dangerous in the wrong circumstances, but makes sense for your particular situation.

The following table lists the built-in types so you can see what is available:

*Table 1: Built-In Types*

| Type (Literal Suffix) | Description | Values/Range |
|---|---|---|
| byte | 8-bit unsigned integer | 0 to 255 |
| sbyte | 8-bit signed integer | -128 to 127 |
| short | 16-bit signed integer | -32,768 to 32,767 |
| ushort | 16-bit unsigned integer | 0 to 65,535 |
| int | 32-bit signed integer | -2,147,483,648 to 2,147,483,647 |
| uint | 32-bit unsigned integer | 0 to 4,294,967,295 |
| long (l) | 64-bit signed integer | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong (ul) | 64-bit unsigned integer | 0 to 18,446,744,073,709,551,615 |
| float (f) | 32-bit floating point | $-3.4 \times 10^{38}$ to $+3.4 \times 10^{38}$ |
| double (d) | 64-bit floating point | $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ |
| decimal (m) | 128-bit, 28 or 29 digits of precision (ideal for financial) | $(-7.9 \times 10^{28}$ to $7.9 \text{ x } 10^{28}) / (10^{0 \text{ to } 28})$ |
| bool | Boolean | true or false |
| char | 16-bit Unicode character (use single quotes) | U+0000 to U+FFFF |
| string | Sequence of Unicode characters (use double quotes) | E.g., “abc” |

You should add a suffix to a number when the meaning would be ambiguous. In the following example, the **`m`** suffix ensures the **`9.95`** literal is treated as a **`decimal`** number:

```
decimal price = 9.95m;
```

*Code Listing 10*

You can assign Unicode values directly to a char. The following example shows how to assign a carriage return.

```
char cr = '\u0013';
```

*Code Listing 11*

You can also obtain the Unicode value of a character with a cast operator as shown here.

```
int crUnicode = (int)cr;
```

*Code Listing 12*

So far, you've only seen statements with the assignment operator, but C# has many other operators that allow you to perform all of the logical operations you would expect of any general purpose programming language. The following table lists some of the available operators.

*Table 2: C# Operators*

| Category | Description |
|---|---|
| Primary | `x.y   x?.y   f(x)   a[x]   x++   x--   new   typeof   default   checked   unchecked   nameof` |
| Unary | `+   -   !   ~   ++x   --x   (T)x   await x` |
| Multiplicative | `*   /   %` |
| Additive | `+   -` |
| Shift | `<<   >>` |
| Relational and Type Testing | `<   >   <=   >=   is   as` |
| Equality | `==   !=` |
| Logical AND | `&` |
| Logical XOR | `^` |
| Logical OR | `\|` |
| Conditional AND | `&&` |
| Conditional OR | `\|\|` |
| Null Coalescing | `??` |
| Conditional | `?:` |
| Assignment | `=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   \|=   =>` |

Prefix operators change the value of the variable before assignment, and postfix operators change a variable after assignment, as demonstrated in the following sample.

```
int val1 = 5;
int val2 = ++val1;
int val3 = 2;
int val4 = val3--;
```

*Code Listing 13*

In the previous code listing, both **`val1`** and **`val2`** are **`6`**. The **`val3`** variable is **`1`**, but **`val4`** is **`2`** because the postfix operator evaluates after assignment.

The ternary operator offers simple syntax for if-then-else logic. Here's an example:

```
decimal priceGain = 2.5m;
string action = priceGain > 2m ? "Buy" : "Sell";
```

*Code Listing 14*

On the left side of **`?`** is a Boolean expression, **`priceGain > 2m`**. If that is true, which it is in this example, the ternary operator returns the first value between **`?`** and **`:`**, which is **`"Buy"`**. Otherwise, the ternary operator would return the value after the **`:`**, which is **`"Sell"`**. This statement assigns the result of the ternary operator, **`"Buy"`**, to the string variable, **`action`**.

In addition to the built-in types, the FCL has many types you will use on a daily basis. One of these is **`DateTime`**, which represents a date and time. Here's a quick demo showing a couple things you can do with a **`DateTime`**.

```
DateTime currentTime = DateTime.Now;
string shortDateString = currentTime.ToShortDateString();
string longDateString = currentTime.ToLongDateString();
string defaultDateString = currentTime.ToString();
DateTime tomorrow = currentTime.AddDays(1);
```

*Code Listing 15*

The previous code shows how to get the current **`DateTime`**, a short representation of a date (e.g., 12/8/2014), a long representation of the date and time (everything spelled out), the default numeric representation, and how to use **`DateTime`** methods for calculations.

***Tip: Search the FCL before creating your own library of types. Many of the common types you use every day, like `DateTime`, will already exist.***

## Operator Precedence and Associativity

The C# operators listed in Table 2 outlines operators in their general order of precedence. The precedence defines which operators evaluate first. Operators of higher precedence evaluate before operators of lower precedence.

Assignment and conditional operators are right-associative and all other operators are left-associative. You can change the normal order of operations by using parentheses as shown in the following code listing.

```
int result1 = 2 + 3 * 5;
int result2 = (2 + 3) * 5;
```

*Code Listing 16*

In the previous code, **`result1`** is **`17`**, but **`result2`** is **`25`**.

## Formatting Strings

There are different ways to build and format strings in C#: concatenation, numeric format strings, or string interpolation. The following code listing demonstrates string concatenation.

```
string name = "Joe";
string helloViaConcatenation = "Hello, " + name + "!";
Console.WriteLine(helloViaConcatenation);
```

*Code Listing 17*

This prints **"Hello, Joe!"** to the console. The following example does the same thing, but uses **`string.Format`**.

```
string helloViaStringFormat = string.Format("Hello, {0}!", name);
Console.WriteLine(helloViaStringFormat);
```

*Code Listing 18*

The **`string.Format`** takes a format string that has numeric placeholders in curly braces. It's 0-based, so the first placeholder is **`{0}`**. The parameters following the string are placed into the format string in the order they appear. Since **`name`** is the first (and only) parameter, **`string.Format`** replaces **`{0}`** with **`Joe`** to create **`"Hello, Joe!"`** as a string. As a convenience in console applications, **`WriteLine`** uses the same formatting. The following code accomplishes the same task as the two lines in the previous code listing.

```
Console.WriteLine("Hello, {0}!", name);
```

*Code Listing 19*

Going a little further, string formatting is more powerful, allowing you to specify column lengths, alignment, and value formatting as shown in the following code.

```
string item = "bread";
decimal amount = 2.25m;
Console.WriteLine("{0,-10}{1:C}", item, amount);
```

*Code Listing 20*

In this example, the first placeholder consumes 10 characters in length. The default alignment is right, but the minus sign changes that to align on the left. On the second placeholder, the **`C`** is a currency format string.

***Note: There are many string formatting options. You can visit https://msdn.microsoft.com/en-us/library/dwhawy9k(v=vs.110).aspx for standard formats, https://msdn.microsoft.com/en-us/library/0c899ak8(v=vs.110).aspx for custom formats, and https://msdn.microsoft.com/en-us/library/az4se3k1(v=vs.110).aspx for DateTime formats.***

C# 6 introduced a new way to format strings, called string interpolation. It's a shorthand syntax that lets you replace numeric placeholders with expressions as follows:

```
Console.WriteLine($"{item}      {amount}");
```

*Code Listing 21*

The **`$`** prefix is required. Here, the value from the **`item`** variable replaces **`{item}`** and the value from the **`amount`** variable replaces **`{amount}`**. Similar to numeric placeholders, you can include additional formatting.

```
Console.WriteLine($"{nameof(item)}: {item,-10} {nameof(amount)}: {amount:C}");
```

*Code Listing 22*

The **`nameof`** operator prints out the name ***"item"***, demonstrating how you can use expressions in placeholders. You can also see the space and currency formatting on **`item`** and **`amount`**.

## Branching Statements

You can use either an **`if–else`** or **`switch`** statement in your code for branching logic. When you only need to execute code for a true condition, use an **`if`** statement as in the following sample.

```
string action2 = "Sell";
if (priceGain > 2m)
```

```
{
    action2 = "Buy";
}
```

*Code Listing 23*

The curly braces are optional in this example because there is only one statement to execute if **`priceGain > 2m`**. However, they would be required for multiple statements. This is true for all branching and logic statements. You can also have an **`else`** case, as shown in the following listing.

```
string action3 = "Do Nothing";
if (priceGain <= 2m)
{
    action3 = "Sell";
}
else
{
    action3 = "Buy";
}
```

*Code Listing 24*

Whenever the Boolean condition of the **`if`** statement is false, as it is in the previous code sample where **`priceGain <= 2m`**, the **`else`** clause executes. In this case, **`action3`** becomes **`"Buy"`**. Of course, you can have multiple conditions by adding more **`else if`** clauses.

```
string action4 = null;
if (priceGain <= 2m)
{
    action4 = "Sell";
}
else if (priceGain > 2m && priceGain <= 3m)
{
    action4 = "Do Nothing";
}
else
{
    action4 = "Sell";
}
```

*Code Listing 25*

In the previous example, you can see a more complex Boolean expression in the **`else if`** clause. When **`priceGain`** is **`2.5`**, the value of **`action4`** becomes **`"Do Nothing"`**. The **`&&`** is a logical operator that succeeds if both the expression on the left and right are true. The logical **`||`** operator succeeds if either the expression on the left or right is true. These operators also perform short-circuit operations where the expression on the right doesn't execute if the expression on the left causes the whole expression to not be true. In the case of the **`else if`** in Code Listing 25, if **`priceGain`** were **`2m`** or less, the **`&&`** operator would not evaluate the

**`priceGain <= 3`** expression because the entire operation is already false. Once a branch of the **`if`** statement executes, no other branches are evaluated or executed.

Notice that I set **`action4`** to **`null`**. The **`null`** keyword means no value. I'll talk about **`null`** in the next chapter and explain where you can use it.

An **`if`** statement is good for either simple branching or complex conditions, such as the previous **`else if`** clause. However, when you have multiple cases and all expressions are constant values, such as an **`int`** or **`string`**, you might prefer a **`switch`** statement. The following example uses a **`switch`** statement to select appropriate equipment based on a weather forecast.

```
string currentWeather = "rain";
string equipment = null;
switch (currentWeather)
{
    case "sunny":
        equipment = "sunglasses";
        break;
    case "rain":
        equipment = "umbrella";
        break;
    case "cold":
    default:
        equipment = "jacket";
        break;
}
```

*Code Listing 26*

The **`switch`** statement tries to match a value, **`currentWeather`** in this example, with one of its **`case`** statements. It uses the **`default`** case for no match. All **`case`** statements must be terminated with a **`break`** statement. The only time fall-through is allowed is when a **`case`** has no body, as demonstrated with the **`"cold" case`** and **`default`**, which both set **`equipment`** to **`"jacket"`**. Since **`currentWeather`** is **`"rain"`**, **`equipment`** becomes **`"umbrella"`** and no other cases execute.

Beyond branching statements, you also need the ability to perform a set of operations multiple times, which is where C# loops come in. Before discussing loops, let's look at arrays and collections, which hold data that loops can use.

## Arrays and Collections

Sometimes you need to group a number of items together in a collection to manage them in memory. For this, you can either use arrays or one of the many collection types in the .NET Framework. The following sample demonstrates how to create an array.

```
int[] oddNumbers = { 1, 3, 5 };
int firstOdd = oddNumbers[0];
int lastOdd = oddNumbers[2];
```

*Code Listing 27*

Here, I've declared and initialized the array with three values. Arrays and collections are 0-based, so **`firstOdd`** is **`1`** and **`lastOdd`** is **`5`**. The **`[x]`** syntax, where **`x`** is a number, is referred to as an indexer because it allows you to access the array at the location specified by the index. Here's another example that uses **`string`** instead of **`int`**.

```
string[] names = new string[3];
names[1] = "Joe";
```

*Code Listing 28*

In this example, I instantiated an array to hold three strings. All of the strings equal **`null`** by default. This code sets the second string to **`"Joe"`**.

In addition to arrays, you can use all types of data structures, such as **`List`**, **`Stack`**, **`Queue`**, and more, which are part of the FCL. The following example shows how to use a **`List`**. Remember to add a **`using`** clause for **`System.Collections.Generic`** to use the **`List<T>`** type.

```
List<decimal> stockPrices = new List<decimal>();
stockPrices.Add(56.23m);
stockPrices.Add(72.80m);
decimal secondStockPrice = stockPrices[1];
```

*Code Listing 29*

In this sample, I instantiated a new **`List`** collection. The **`<decimal>`** is a generic type indicating that this is a strongly typed list that can only hold values of type **`decimal`**; it's a **`List`** of **`decimal`**. That list has two items. Notice how I used the array-like indexer syntax to read the second item in the (0-based) **`stockPrices`** list.

## Looping Statements

C# supports several loops, including **`for`**, **`foreach`**, **`while`**, and **`do`**. The code listings that follow perform similar logic.

```
double[] temperatures = { 72.3, 73.8, 75.1, 74.9 };
for (int i = 0; i < temperatures.Length; i++)
{
    Console.WriteLine(i);
}
```

*Code Listing 30*

The **`for`** loop initializes **`i`** to **`0`**, makes sure **`i`** is less than the number of items in the **`temperature`** array, executes the **`Console.WriteLine`**, and then increments **`i`**. It continues executing until the condition (**`i < temperatures.Length`**) is false, and then moves on to the next statement in the program.

```
foreach (int temperature in temperatures)
{
    Console.WriteLine(temperature);
}
```

*Code Listing 31*

The **`foreach`** loop used in Code Listing 31 is simpler and will execute for each value in the **`temperatures`** array.

Next is an example of a **`while`** loop.

```
int tempCount = 0;
while (tempCount < temperatures.Length)
{
    Console.WriteLine(tempCount);
    tempCount++;
}
```

*Code Listing 32*

The **`while`** loop evaluates the condition and executes if it's true. Notice that I initialized **`tempCount`** to **`0`** and increment **`tempCount`** inside of the loop on each iteration.

Finally, the following example shows how to write a **`do-while`** loop.

```
int tempCount2 = 0;
do
{
    Console.WriteLine(tempCount2++);
}
while (tempCount2 <= temperatures.Length);
```

*Code Listing 33*

A **`do-while`** loop is good for when you want to execute logic at least one time. This example increments **`tempCount2`** as a parameter to **`Console.WriteLine`**. Remember, the postfix operator changes the variable after evaluation.

## Wrapping Up

Here’s a calculator program that pulls together some of the concepts from this chapter, plus some extra features. You can type this into your editor and execute it for practice.

```
using System;
using System.Text;

/*
    Title: Calculator
    By: Joe Mayo
*/

class Calculator
{
    /// <summary>
    /// This is the entry point.
    /// </summary>
    static void Main()
    {
        char firstChar = 'Q';
        bool keepRunning = true;

        do
        {
            Console.WriteLine();
            Console.Write("What do you want to do? (Add, Subtract, Multiply, Divide,
Quit): ");
            string input = Console.ReadLine();
            firstChar = input[0];

            // This is used in both the if statement and the do-while loop.
            keepRunning = !(firstChar == 'q' || firstChar == 'Q');

            double firstNumber = 0;
            double secondNumber = 0;

            if (keepRunning)
            {
                Console.Write("First Number: ");
                string firstNumberInput = Console.ReadLine();
                firstNumber = double.Parse(firstNumberInput);

                Console.Write("Second Number: ");
                string secondNumberInput = Console.ReadLine();
                secondNumber = double.Parse(secondNumberInput);
            }

            double result = 0;
            switch (firstChar)
            {
                case 'a':
                case 'A':
                    result = firstNumber + secondNumber;
                    break;
                case 's':
                case 'S':
                    result = firstNumber - +secondNumber;
                    break;
                case 'm':
                case 'M':
                    result = firstNumber * secondNumber;
                    break;
                case 'd':
                case 'D':
                    result = firstNumber / secondNumber;
                    break;
                default:
                    result = -1;
                    break;
            }

            Console.WriteLine();
            Console.WriteLine("Your result is " + result);

        } while (keepRunning);
    }
}
```

*Code Listing 34*

The previous program demonstrates a **`do-while`** loop, an **`if`** statement, a **`switch`** statement, and a basic console communication with the user.

There are a couple string features here that you haven't seen yet. The first is where the program uses **`Console.ReadLine`** to read input text from the user for the input string. Notice the indexer syntax to read the first character from the string. You can read any character of a string this way. Also, look at the bottom of the program where it prints **`"Your result is " + result`**, which concatenates a string with the number. Using the **`+`** operator for concatenation is a simple way to build strings. Another way to build a string is with a type named **`StringBuilder`**, which you can use like this:

```
StringBuilder sb = new StringBuilder();
sb.Append("Your result is ");
sb.Append(result.ToString());
Console.WriteLine(sb.ToString());
```

*Code Listing 35*

You'll also need to add a **`using System.Text;`** clause to the top of the file. After you've used the concatenate operator, **`+`**, about four times on the same string, you might consider rewriting with a **`StringBuilder`** instead. The **`string`** type is immutable, meaning that you can't modify it. This also means that every concatenation operation causes the CLR to create a new string in-memory.

The calculator program also has multiline and single-line comments that aren't compiled, but help you document the code as you need. Here's the multi-line comment:

```
/*
    Title: Calculator
    By: Joe Mayo
*/
```

*Code Listing 36*

Here's the single-line comment:

```
// This is used in both the if statement and the do-while loop.
```

*Code Listing 37*

An extension of the single-line comment is a convention that uses three slashes and a set of XML tags, known as documentation comments.

```
/// <summary>
/// This is the entry point.
/// </summary>
```

*Code Listing 38*

## Summary

C# has a full set of operators and types that allow you to write a wide range of expressions and statements. With branching statements and loops, you can write logic of your choosing. All of the code in this chapter has been in the **`Main`** method, but clearly that's inadequate and you'll quickly grow out of that. The next chapter explores some new C# features to help organize code with methods and properties.

# Chapter 3 Methods and Properties

Previous chapters show how to write code in the **Main** method. That's the program entry point, but it's normally a lightweight method without too much code. For this chapter, you'll learn how to move your code out of the **Main** method and modularize it so you can manage the code better. You'll learn how to define methods with parameters and return values. You'll also learn about properties, which let you encapsulate object state.

## Starting at Main

We'll use a simpler version of the calculator from the previous chapter to get started. This calculator only performs addition and stops running after one operation.

```
using System;

class Calculator1
{
    static void Main()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);

        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);

        double result = firstNumber + secondNumber;

        Console.WriteLine($"\n\tYour result is {result}.");

        Console.ReadKey();
    }
}
```

*Code Listing 39*

The part of this program that might be new is the **Console.ReadKey** statement at the end of the **Main** method. This allows users to see the result and keeps the program from ending until they press a key. The **\n** in the interpolated string is a newline and **\t** is a tab.

## Modularizing with Methods

Although the previous program is small, a first glance doesn't really tell you what it does. Imagine if it was like the calculator in Chapter 2 or even longer; it would eventually become difficult to understand. When you have to work on this again later, you might need to read many lines of code to understand it. So, it would be better to refactor this. Refactoring is the practice of changing the design of code without changing its functionality; the purpose is to improve the program. The following code sample is a first draft of refactoring this program into methods.

```
using System;

class Calculator2
{
    static void Main()
    {
        double firstNumber = GetFirstNumber();

        double secondNumber = GetSecondNumber();

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }

    static double GetSecondNumber()
    {
        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);
        return secondNumber;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

*Code Listing 40*

Looking at **`Main`**, you can tell what the program does. It reads two numbers, adds the results, and then shows the results to the user. Each of those lines is a method call. The first three methods—**`GetFirstNumber`**, **`GetSecondNumber`**, and **`AddNumbers`**—return a value that is assigned to a variable. The last method, **`PrintResult`**, performs an action without returning a result. Before moving to the next refactoring, let's walk through these methods. The following code listing shows the **GetFirstNumber** method.

```
    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }
```

*Code Listing 41*

At first glance, the signature of this method looks similar to the **Main** method. The differences are that the return type of this method is **double** and the method is named **GetFirstNumber**. All we did was write the method and the code that creates the **firstNumber**. When a method has a return type, a value of that type must be returned. **GetFirstNumber** does that with the **return** statement.

The **GetSecondNumber** method is nearly identical to **GetFirstNumber**. Let's examine **AddNumbers** next.

```
    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }
```

*Code Listing 42*

Notice that **Main** passes the **firstNumber** and **secondNumber** variables to **AddNumbers** as arguments that the **AddNumbers** method can work with as parameters. The return type of **AddNumbers** is **double**, so the method adds and returns the result of the add operation.

Finally, we have the **PrintResult** method.

```
    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
```

*Code Listing 43*

The **PrintResult** method writes the results from its parameter to the console. Notice that **PrintResult** does not have a return type, as indicated by the **void** keyword.

## Simplifying Code with Methods

The last section improved the program because a huge block of code was broken into more meaningful pieces. We can improve this code with some extra refactoring. In particular, the **`GetFirstNumber`** and **`GetSecondNumber`** methods are largely redundant. The following sample shows how to refactor those two methods into one and reduce the amount of code.

```
using System;

class Calculator3
{
    static void Main()
    {
        double firstNumber = GetNumber("First");
        double secondNumber = GetNumber("Second");

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);
        return number;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

*Code Listing 44*

This time I removed **`GetFirstNumber`** and **`GetSecondNumber`** and replaced them with **`GetNumber`**. The only real difference besides variable names is the **`whichNumber string`** parameter.

## Adding Properties

The previous examples performed all of the operations inside of the same class. It was driven from the **`Main`** method and serviced through methods. What if I wanted to reuse the calculator functions in that class and wanted the new class to hold its own values, or state? In this case, moving the calculator methods into a separate **`Calculator`** class would be useful.

The next question to ask is, "How do we get to the state of the class?" For example, if I want to read the result from the **`Calculator`** class, what is the best way to do so? One approach is to use a method named **`GetResult`** that returns the value. Another way in C# is to use a property, which you can use like a field, but works like a method. The following version of the calculator program shows how to refactor methods into a separate class and add properties.

***Note: Refactoring is the practice of changing the design of code without changing its behavior with the goal of improving the code. Martin Fowler's book,*** **Refactoring: Improving the Design of Existing Code*, is a good reference.***

```
using System;

public class Calculator4
{

    double[] numbers = new double[2];

    public double First
    {
        get
        {
            return numbers[0];
        }
    }

    public double Second
    {
        get
        {
            return numbers[1];
        }
    }

    double result;

    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    public void GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);

        if (whichNumber == "First")
            numbers[0] = number;
        else
            numbers[1] = number;
    }

    public void AddNumbers()
    {
        Result = First + Second;
    }

    public void PrintResult()
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

*Code Listing 45*

In the previous code listing, **`First`**, **`Second`**, and **`Result`** are properties. I'll break down the syntax shortly, but first look at how these properties are used inside of the **`AddNumbers`** and **`PrintResults`** methods. **`AddNumbers`** reads the values of **`First`** and **`Second`** and adds those values together and writes to **`Result`**.

Each of these properties looks just like a field or variable; you just read from and write to them. **`PrintResult`** reads the **`Result`** property. However, looking at the definitions of the properties, you can tell right away that they aren't fields.

The **`Result`** property is a typical read and write property with **`get`** and **`set`** accessors. When you read the property, the **`get`** accessor executes. When you write to the property, the **`set`** accessor executes. Notice that there is a **`result`** field (lowercase) prior to the **`Result`** (uppercase) property. The **`get`** accessor reads the value of **`result`** and the **`set`** accessor writes to **`result`**. When using **`set`**, the **`value`** keyword represents what is being written to the property.

This pattern of reading and writing from a single backing store is so common that C# has a shortcut syntax you can use instead. The following code sample shows **`Result`** rewritten as an auto-implemented property.

```
    public double Result { get; set; }
```

*Code Listing 46*

The backing store in auto-implemented properties is implied and handled behind the scenes by the C# compiler. If you need to provide validation on the value being assigned or have a unique way of storing the value, you should resort to full properties where the **`get`** accessor, **`set`** accessor, or both are defined.

In fact, **`First`** and **`Second`** properties have a unique backing store, requiring a fully implemented **`get`** accessor. They read from an array position. Notice that the **`GetNumber`** method figures out which array position to put each number into.

Properties give you the ability to encapsulate the internal operations of your class so you are free to modify the implementation without breaking the interface for consumers of your class.

The following code sample demonstrates how consuming code might use this new **`Calculator4`** class.

```
using System;

class Program
{
    static void Main()
    {
        var calc4 = new Calculator4();

        calc4.GetNumber("First");
        calc4.GetNumber("Second");

        calc4.AddNumbers();

        PrintResult(calc4);

        Console.ReadKey();
    }

    static void PrintResult(Calculator4 calc)
    {
        Console.WriteLine();
        Console.WriteLine($"Your result is {calc.Result}.");
    }
}
```

*Code Listing 47*

The **`Main`** method creates a new instance of **`Calculator4`** and calls public methods. All of the strange internals of **`Calculator4`** are hidden and **`Main`** only cares about the public interface, exposing **`Calculator4`** services for reuse. The **`PrintResult`** method reads the **`Calculator4`** instance **`Result`** property. Again, that's the benefit of methods and properties: callers can use a class without caring how that class does what it does.

## Exception Handling

C# has a feature called structured exception handling that lets you work with situations where your methods aren't able to fulfill their intended purpose. The syntax for managing exception handling is the **`try`**-**`catch`** block. All the code to be monitored for exceptions goes in the **`try`** block, and the code that handles a potential exception goes in a **`catch`** block. The following code listing shows an example.

```
        static void HandleNullReference()
        {
            Program prog = null;

            try
            {
                Console.WriteLine(prog.ToString());
            }
            catch (NullReferenceException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
```

*Code Listing 48*

In C#, any time you try to use a member of a **null** object, you'll receive a **NullReferenceException**. The solution to fix the problem is to assign a value to the variable. The previous example causes a **NullReferenceException** to be thrown in the **try** block by calling **ToString** on the **prog** variable, which is **null**.

Since the code that threw the exception is inside the **try** block, the code stops execution of any of the code in the **try** block and starts looking for an exception handler. The **catch** block **parameter** indicates that it can catch a **NullReferenceException** if the code inside of the **try** block throws that exception type. The body of the **catch** block is where you perform any exception handling.

You can customize exception handling with multiple **catch** blocks. The following example shows code that throws an exception in the **try** block, which is subsequently handled by a **catch** block.

```
        static void HandleUncaughtException()
        {
            Program prog = null;

            try
            {
                Console.WriteLine(prog.ToString());
            }
            catch (ArgumentNullException ex)
            {
                Console.WriteLine("From ArgumentNullException: " + ex.Message);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("From ArgumentException: " + ex.Message);
            }
            catch (Exception ex)
            {
                Console.WriteLine("From Exception: " + ex.Message);
            }
            finally
            {
                Console.WriteLine("Finally always executes.");
            }
        }
```

*Code Listing 49*

The method name is **HandleUncaughtException** because there isn't a specific **catch** block to handle a **NullReferenceException**; the exception will be handled by the **catch** block for the **Exception** type.

You list exceptions by their inheritance hierarchy, with top-level exceptions lower in the list of **catch** blocks. A thrown exception will move down this list of handlers, looking for a matching exception type, and only execute in the **catch** block of the first handler that matches. **ArgumentNullException** derives from **ArgumentException**, and **ArgumentException** derives from **Exception**.

If no **catch** block can handle an exception, the code goes up the stack looking for a potential **catch** block in calling code that can handle the exception type. If no code in the call stack is able to handle the exception, your program will terminate.

The **finally** block always executes if the program begins executing code in the **try** block. If an exception occurs and is not caught, the **finally** block will execute before the program looks at the calling code for a matching catch handler.

You can write a **try**-**finally** block (without **catch** blocks) to guarantee that certain code will execute once the **try** block begins. This is useful for opening a resource, like a file or database connection, and then guaranteeing you will be able to close that resource regardless of whether an exception occurs.

If you encounter a reason why your method can't perform its intended purpose, **throw** an exception. There are many **Exception**-derived types in the .NET FCL that you can use. The following code example pulls together a few concepts you might want to use, such as validating method input and throwing an **ArgumentNullException**.

```
public class Address
{
    public string City { get; set; }
}

internal class Company
{
    public Address Address { get; set; }
}

        // Inside of a class...
        static void ThrowException()
        {
            try
            {
                ValidateInput("something", new Company());
            }
            catch (ArgumentNullException ex) when (ex.ParamName == "inputString")
            {
                Console.WriteLine("From ArgumentNullException: " + ex.Message);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("From ArgumentException: " + ex.Message);
            }
        }

        static void ValidateInput(string inputString, Company cmp)
        {
            if (inputString == null)
                throw new ArgumentNullException(nameof(inputString));

            if (cmp?.Address?.City == null)
                throw new ArgumentNullException(nameof(cmp));
        }
```

*Code Listing 50*

The previous code shows an **Address** class and a **Company** class with a property of the **Address** type. The **try** block of the **ThrowException** message passes a new instance of **Company**, but doesn't instantiate **Address**, meaning that the **Company** instance's **Address** property is **null**.

Inside **ValidateInput**, the **if** statement uses the null referencing operator, **?.**, to check if any of the values between **Company**, **Company**'s **Address** property, or **Address**'s **City** property is **null**. This is a convenient way to check for **null** without a group of individual checks, producing less syntax and simpler code. If any of these values are **null**, the code throws an **ArgumentNullException**.

The argument to the **ArgumentNullException** uses the **nameof** operator, which evaluates to a **string** representation of the value passed to it; it is **"cmp"** in this case. This code isn't enclosed in a **try** block, so control returns to the code calling this method.

Back in the **ThrowException** method, the thrown exception causes the code to look for a handler suitable for this exception type. The exception type is **ArgumentNullException**, but the **catch** block for **ArgumentNullException** won't execute. That's because the **when** clause following the **ArgumentNullException catch** block parameter is checking for a **ParamName** of **"inputString"**. This **when** clause is called an exception filter. As mentioned previously, the parameter name passed to the **ArgumentNullException** during instantiation was **"cmp"**, so there isn't a match. Therefore, the code continues looking at **catch** handlers.

Since **ArgumentNullException** derives from **ArgumentException** and there is no exception filter, the **catch** handler for **ArgumentException** executes. The exception is now handled.

***Tip: It's typically better to throw and handle specific exceptions, rather than their parent exceptions. This adds more fidelity and meaning to the exception and makes debugging easier.***

## Summary

Methods help you organize code into named functions that you can call to perform operations. Their name documents what the code does. Also, methods are useful to help avoid duplicating the same code in multiple places. Properties are used like fields and look like fields from the perspective of code using the property's class. However, properties are more sophisticated in that they have **`get`** and **`set`** accessors that let them work like methods and perform more sophisticated work, like validation or special value handling. Both methods and properties help define the interface of a class to consumers and let you encapsulate the internal operations of a class, which makes it more reusable. You can use **`try`**-**`catch`** blocks to handle exceptions and **`try`**-**`finally`** blocks to guarantee critical code executes. Use the **`throw`** statement whenever a method you're writing can't fulfill its intended purpose.

# Chapter 4  Writing Object-Oriented Code

C# is an object-oriented programming (OOP) language. It supports inheritance, encapsulation, polymorphism, and abstraction. This chapter shows you how C# supports OOP.

## Implementing Inheritance

In C#, inheritance defines a relationship between two classes where a derived class can reuse members of a base class. A simple way to view this is that a derived class is a more specific version of a base class. We will reuse the calculator example from previous chapters, but alter it to provide two different types of calculators: scientific and programmer. Since they're both calculators, it could be useful to create a relationship with a **`Calculator`** base class and **`ScientificCalculator`** and **`ProgrammerCalculator`** derived classes, like this:

- **`Calculator`**
  - **`ScientificCalculator`**
  - **`ProgrammerCalculator`**

In C#, you would express this relationship as follows.

```
public class Calculator { }
public class ScientificCalculator : Calculator { }
public class ProgrammerCalculator : Calculator { }
```

*Code Listing 51*

As you can see, we added a colon suffix (as the inheritance operator) to the derived class and specified the base class it is derived from. The following figure illustrates the inheritance relationship between these classes.

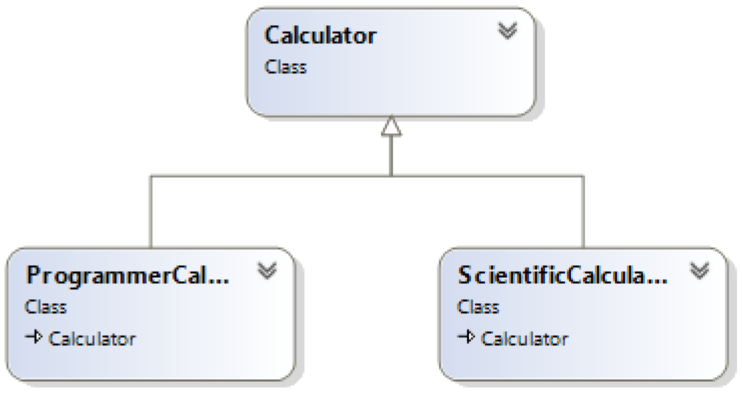


*Figure 1: Calculator Inheritance Diagram*

You can assume that **`Calculator`** will have all of the standard operations like addition, subtraction, and more that all calculators have. The following code listing is an expanded example that shows the base class with a common method, and the derived classes with specialized methods that only belong to those classes.

```
using System;

public class Calculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public double Power(double num, double power)
    {
        return Math.Pow(num, power);
    }
}

public class ProgrammerCalculator : Calculator
{
    public int Or(int num1, int num2)
    {
        return num1 | num2;
    }
}
```

*Code Listing 52*

The methods of the derived classes in the previous example used the FCL **`Math`** class for **`Power`**, which has many more math methods you can use in your own code, and also used the built-in C# **`|`** operator for **`Or`**. The following example shows how to write code that takes advantage of inheritance with the previous classes.

```
using System;

public class Program
{
    public static void Main()
    {
        ScientificCalculator sciCalc = new ScientificCalculator();
        double powerResult = sciCalc.Power(2, 5);
        Console.WriteLine($"Scientific Calculator 2**5: {powerResult}");
        double sciSum = sciCalc.Add(3, 3);
        Console.WriteLine($"Scientific Calculator 3 + 3: {sciSum}");

        ProgrammerCalculator prgCalc = new ProgrammerCalculator();
        double orResult = prgCalc.Or(5, 10);
        Console.WriteLine($"Programmer Calculator 5 | 10: {orResult}");
        double prgSum = prgCalc.Add(3, 3);
        Console.WriteLine($"Programmer Calculator 3 + 3: {prgSum}");

        Console.ReadKey();
    }
}
```

*Code Listing 53*

Both the **`ScientificCalculator`** instance, **`sciCalc`**, and the **`ProgrammerCalculator`** instance, **`prgCalc`**, call the **`Add`** method. Further, those classes don't define their own **`Add`**, but they do derive from **`Calculator`** and therefore inherit **`Calculator`**'s **`Add`**.

The ability to inherit isn't always guaranteed; the next section explains more about when a class member is visible to other classes.

## Access Modifiers and Encapsulation

In the previous example, all classes and methods had **`public`** modifiers, meaning that any other class or code can see and access them in code. You can leave off access modifiers and accept defaults. In that case, a class access becomes what is known as **`internal`**, and class members default to **`private`**.

Classes can only be **`internal`** or **`public`**. If they're **`internal`**, they can only be accessed by code inside of the assembly they are contained in.

Available access modifiers for class members include **`public`**, **`private`**, **`internal`**, **`internal protected`**, and **`protected`**. The **`public`** and **`internal`** modifiers have the same meaning for class members as they do for classes.

The default modifier for class members, **`private`**, means that code outside the class can't use that member; only other members within the same class can use it. This is useful if you want to modularize a method by breaking it into different supporting methods, but the supporting methods have no meaning outside of the class.

The **`protected`** modifier allows derived classes inside and outside the assembly to use the **`protected`** base class member. The **`internal protected`** modifier further restricts protected behavior only to derived classes inside the same assembly.

Most of the access modifier defaults and behaviors apply to **`struct`** types as well as classes, except for **`protected`** and **`internal protected`**, as I'll explain next.

## Designing Types: Class vs. Struct

A **`struct`** is another C# type that looks similar to a **`class`**, but has different behavior. A **`struct`** can't derive from another **`class`** or **`struct`**. Since implementation inheritance doesn't apply to a **`struct`**, neither do **`protected`** and **`internal protected`** modifiers. A **`struct`** does have interface inheritance, which I'll explain more in the Exposing Interfaces section later in this chapter.

Additionally, a **struct** copies by value, as opposed to a **class** which copies by reference. The difference is that if you pass a **struct** instance to a method, the method gets a brand new copy of the **struct** value. If you copy a **class** instance to a method, the method gets a copy of a reference to the **class** in the heap, which is an area in computer memory that the CLR uses to allocate space for reference type objects. These facts help you decide whether you should design a type as a **class** or **struct**. Imagine a type with many properties and how it would hurt performance if you had to pass it by value to a method as a **struct**; the state of that type is copied to the stack, which is memory the CLR allocates for every method call to hold items like parameters and local variables. In that case, the proper design decision might be to define the type as a **class** so that only the reference is copied.

Most of the built-in types, such as **int**, **double**, and **char**, are value types. If you have a type with those semantics—small and a single value—then designing a type as a **struct** might be a benefit. Otherwise, designing a type as a **class** is fine. Here's an example of a type that might make a good **struct**.

```
public struct Complex
{
    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public double Real { get; set; }

    public double Imaginary { get; set; }

    public static Complex operator +(Complex complex1, Complex complex2)
    {
        Complex complexSum = new Complex();
        complexSum.Real = complex1.Real + complex2.Real;
        complexSum.Imaginary = complex1.Imaginary + complex2.Imaginary;
        return complexSum;
    }
    public static implicit operator Complex(double dbl)
    {
        Complex cmplx = new Complex();
        cmplx.Real = dbl;
        return cmplx;
    }
    // This is not a safe operation.
    public static explicit operator double(Complex cmplx)
    {
        return cmplx.Real;
    }
}
```

*Code Listing 54*

**Complex** could make a good **struct** because you might have a lot of mathematical operations, and it would be more efficient to pass a copy of the numbers on the stack rather than letting the CLR allocate memory as it does for a **class**.

**Complex** has a constructor, named after the class itself, with a couple parameters. This makes it easy to initialize a new instance of **Complex**.

There are a few operator overloads in **Complex**: an addition operator and two conversion operators. The addition operator lets you add two complex numbers. Where the operator identifier (**+**) precedes the parameter list, the values to be added are specified in the parameters, and the return type is part of the signature. Operators are always **static**.

The two conversion operators let you make assignments between the containing type and another type of your choice. The type assigned to is the operator identifier and the type being assigned is the parameter. The **implicit** modifier means the conversion is safe and the **explicit** modifier means the conversion has the potential to lose data or provide an invalid result. For example, assigning a **double** to an **int** would be **explicit** because of loss of precision, and the explicit conversion in the previous example causes loss of the imaginary part of the number. The following code sample demonstrates how **Complex** could be used.

```
using System;

class Program
{
    static void Main()
    {
        Complex complex1 = new Complex();
        complex1.Real = 3;
        complex1.Imaginary = 1;

        Complex complex2 = new Complex(7, 5);

        Complex complexSum = complex1 + complex2;

        Console.WriteLine(
            $"Complex sum - Real: {complexSum.Real}, " +
            $"Imaginary: {complexSum.Imaginary}");

        Complex complex3 = 9;

        double realPart = (double)complex3;

        Console.ReadKey();
    }
}
```

*Code Listing 55*

The **Main** method instantiates **complex1** and then populates its values. Next, **Main** instantiates **complex2** by using the **Complex** constructor, which is simpler initialization code.

You can also see how natural it is to use the addition operator, rather than an **`Add`** method used in previous **`Calculator`** demos.

Because there's an implicit conversion from **`int`** to **`double`** and **`Complex`** has an implicit conversion operator from **`double`** to **`Complex`**, **`Main`** is able to assign **`9`** to **`complex3`**. The same can't be said for assigning **`complex3`** to **`realPart`** because **`Complex`** to **`double`** is an **`explicit`** conversion operator in the **`Complex`** type. Any time you have an **`explicit`** conversion, you must use a cast operator, as in **`(double)complex3`**.

One of the items you need to watch out for when working with value types is a concept referred to as boxing and unboxing. Boxing occurs any time you assign a value type to **`object`**, and unboxing occurs when you assign **`object`** to a value type. The following code demonstrates one scenario where this could happen.

```
ArrayList intCollection = new ArrayList();
intCollection.Add(7);
int number = (int)intCollection[0];
```

*Code Listing 56*

An **`ArrayList`** is a collection class belonging to the **`System.Collections`** namespace. It is more powerful than an array and operates on type **`object`**. The **`Add`** method accepts an argument of type **`object`**. Since all types derive from **`object`**, an **`ArrayList`** is flexible enough to allow you to work with objects of any type. Boxing occurs when passing **`7`** to the **`Add`** method because **`7`** is an **`int`** (a value type) and is converted to **`object`**. What is really happening is that the CLR creates a boxed **`int`** in memory. Since the **`ArrayList`** holds type **`object`**, you also need to perform a conversion to unbox a value when reading from the **`ArrayList`**. The **`(int)`** cast operator converts from **`object`** (the boxed **`int`**) to **`int`** when reading the first element of **`intCollection`**.

The problem that boxing and unboxing cause is related to performance. In this situation, the reason you would use a collection is because you want to hold a lot of **`int`** values, which could be hundreds or thousands. Think about all the time spent accessing that **`ArrayList`** and incurring the performance penalty of boxing and unboxing on each operation.

***Note:*** *`ArrayList`* ***is an old collection class that existed in C# v1.0 and is no longer used in modern development. C# v2.0 introduced generics, which use new collection classes that are strongly typed and avoid the boxing and unboxing penalties. While the*** *`ArrayList`* ***example is unlikely today, this scenario still highlights the performance penalty of any other situation where you might be assigning a value type to an object type.***

Another difference between **`class`** (reference types) and **`struct`** (value types) is equality evaluation. Value type equality works by comparing the corresponding members of the **`struct`**. Reference type equality works by verifying that references are equal. In other words, structs are equal if their values match, but classes are equal if they reference the same object in memory. In the later section on polymorphism, you'll learn how to override the **`object.Equals`** method to give you more control over class equality.

## Creating Enums

An **`enum`** is a value type that lets you create a set of strongly typed mnemonic values. They're useful when you have a finite set of values and don't want to represent those values as strings or numbers. Here's an example of an **`enum`**.

```
public enum MathOperation
{
    Add,
    Subtract,
    Multiply,
    Divide
}
```

*Code Listing 57*

Like a **`struct`**, an **`enum`** is a value type. You use the **`enum`** keyword as the type definition. The previous **`enum`** is named **`MathOperation`** and has four members. The following example shows how you can use this **`enum`**.

```
using System;
using static MathOperation;

class Program
{
    static void Main()
    {
        string[] possibleOperations = Enum.GetNames(typeof(MathOperation));

        Console.Write($"Please select ({string.Join(", ", possibleOperations)}): ");

        string operationString = Console.ReadLine();

        MathOperation selectedOperation;

        if (!Enum.TryParse<MathOperation>(operationString, out selectedOperation))
            selectedOperation = MathOperation.Add;

        switch (selectedOperation)
        {
            case MathOperation.Add:
                Console.WriteLine($"You selected {nameof(Add)}");
                break;
            case MathOperation.Subtract:
                Console.WriteLine($"You selected {nameof(Subtract)}");
                break;
            case MathOperation.Multiply:
                Console.WriteLine($"You selected {nameof(Multiply)}");
                break;
            case MathOperation.Divide:
                Console.WriteLine($"You selected {nameof(Divide)}");
                break;
        }
        Console.ReadKey();
    }
}
```

*Code Listing 58*

The FCL has an **Enum** class that lets you work with enums and the previous **Main** method shows how to use a couple of its methods. **Enum.GetNames** returns a **string** array, representing the names in the **enum**, specified with the **typeof** operator. The **string.Join** method, the expression in the interpolated string of the **Console.WriteLine**, creates a comma-separated string of these names.

The **Enum.TryParse** method in the previous example takes a string and produces an **enum** of the type specified in the type parameter, which is **MathOperation** in this case. The **out** parameter means that **TryParse** will return the parsed value in the **selectedOperation** variable. This is practical because the return type of the **TryParse** is **bool**, allowing you to know whether the input string, **operationString**, is valid.

The **selectedOperation** variable is of type **MathOperation**. The default syntax for enums is to prefix them with the enum type name, as in **MathOperation.Add**. However, you can also add a **using static** clause to the top of the file, allowing you to only specify the member name, as the previous example shows in the **switch** statement. A **switch** statement can operate on numbers, strings, or enums.

## Enabling Polymorphism

Polymorphism lets derived classes specialize a base class implementation. The mechanism to allow polymorphism is to decorate a base class method with the **virtual** modifier and decorate the derived class method with the **override** modifier. If you were designing the **Calculator** class, you could allow derived classes to implement their own improved or specialized versions of the **Add** method, as shown in the following sample.

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}

public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ScientificCalculator Add called.");
        return base.Add(num1, num2);
    }
}
```

*Code Listing 59*

Polymorphism is opt-in for C#. Notice that the **`Add`** method in the base class **`Calculator`** has a **`virtual`** modifier. Polymorphism doesn't occur unless a base class method has the **`virtual`** modifier. Also, notice that the derived classes **`ScientificCalculator`** and **`ProgrammerCalculator`** have **`override`** modifiers. Again, these methods won't be called polymorphically unless they have the **`override`** modifier. Additionally, a method with the **`override`** modifier is also **`virtual`** for any of its derived classes.

With polymorphism, the overridden method in derived classes executes at runtime. If you wanted to call the base class implementation of that method, call the base class method with the **`base`** keyword. **`ScientificCalculator`** calls **`base.Add(num1, num2)`** to call the **`Add`** method in **`Calculator`**. Here's an example of how this works.

```
using System;

public class Program
{
    public static void Main()
    {
        Calculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        Calculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");

        Console.ReadKey();
    }
}
```

*Code Listing 60*

The output for this program would be:

```
ScientificCalculator Add called.

Calculator Add called.

Scientific Calculator 2 + 5: 7

ProgrammerCalculator Add called.

Programmer Calculator 5 + 10: 15
```

`Main` assigns instances of `ScientificCalculator` and `ProgrammerCalculator` to variables of type `Calculator`. As you saw in the previous listing, `ScientificCalculator` and `ProgrammerCalculator` are derived types and `Calculator` is their base type. The derived instances are the runtime type—the actual type when the program is running—and the base class is the compile-time type. The runtime-type overrides execute at runtime.

Looking at the definition of `Add` in `ScientificCalculator`, `Calculator`, and `Main`, and looking at the output, you can trace the polymorphic behavior of this program. `Main` calls `Add` on the `ScientificCalculator` instance. `ScientificCalculator.Add` executes because it overrides the `virtual` `Calculator.Add` method. After writing the first line of output, `ScientificCalculator.Add` calls the `Calculator.Add` method with the `base` keyword. `Calculator.Add` prints the second line to the output, performs the addition calculation, and returns the sum. `ScientificCalculator.Add` returns the return value from `Calculator.Add`. `Main` assigns the return value from `ScientificCalculator.Add` to the `sciCalc` variable and prints the results into the third line of the output. Tracing the call to `ProgrammerCalculator.Add` is similar, except that there is no call to the `Calculator.Add` in the base class.

Another example of when you would want to use polymorphism is in defining reference type equality. By default, reference types are only equal if their references are the same. The following example shows how to control reference type equality.

```
public class Customer
{
    int id;
    string name;

    public Customer(int id, string name)
    {
        this.id = id;
        this.name = name;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (obj.GetType() != typeof(Customer))
            return false;
        Customer cust = obj as Customer;

        return id == cust.id;
    }

    public static bool operator ==(Customer cust1, Customer cust2)
    {
        return cust1.Equals(cust2);
    }

    public static bool operator !=(Customer cust1, Customer cust2)
    {
        return !cust1.Equals(cust2);
    }

    public override int GetHashCode()
    {
        return id;
    }

    public override string ToString()
    {
        return $"{{ id: {id}, name: {name} }}";
    }
}
```

*Code Listing 61*

Since all classes implicitly derive from **`object`**, they can **`override`** object **`virtual`** methods **`Equals`**, **`GetHashCode`**, and **`ToString`**. **`Customer`** overrides **`Equals`**. When you override **`Equals`**, check for **`null`** and type equality before working with the objects to prevent callers from accidentally comparing **`null`** or incompatible types. **`Customer`** instances are equal if they have the same **`id`**.

**`Customer`** has a constructor that initializes the state of the class. The **`this`** operator lets you access members of the containing instance and helps avoid ambiguity.

When implementing custom equality, you should also overload the equals and not equals and override the **`GetHashCode`** method. The default implementation of **`GetHashCode`** is a system-defined object **`id`**, so you could override this to achieve a better distribution of values in a hash.

***Tip: You can escape `{` and `}` characters that you don't want to evaluate as expressions by doubling them as `{{` and `}}` respectively in string interpolation.***

The following is an example of how to check equality of **`Customer`** instances.

```
using System;

class Program
{
    static void Main()
    {
        Customer cust1 = new Customer(1, "May");
        Customer cust2 = new Customer(2, "Joe");

        Console.WriteLine($"cust1 == cust2: {cust1 == cust2}");

        Customer cust3 = new Customer(1, "May");

        Console.WriteLine($"\ncust1 == cust3: {cust1 == cust3}");
        Console.WriteLine($"cust1.Equals(cust3): {cust1.Equals(cust3)}");
        Console.WriteLine($"object.ReferenceEquals(cust1, cust3):
{object.ReferenceEquals(cust1, cust3)}");

        Console.WriteLine($"\ncust1: {cust1}");
        Console.WriteLine($"cust2: {cust2}");
        Console.WriteLine($"cust3: {cust3}");

        Console.ReadKey();
    }
}
```

*Code Listing 62*

When using the **`==`** operator, the code calls the operator overload and **`Equals`** calls the equals method as expected. **`ReferenceEquals`** is an **`object`** method that is useful because it allows reference equality checking in case the type defined a custom **`Equals`** override.

If **`Customer`** had not overridden **`ToString`**, the last three **`Console.WriteLine`** statements in the previous code listing would have printed the type name, which is the default behavior of **`ToString`**.

## Writing Abstract Classes

In previous examples, you could create an instance of **`Calculator`**. However, it may or may not make sense to instantiate a base class. A base class may serve only as a reusable type for common functionality of similar derived classes and to enable polymorphism, yet not have substance enough to be used on its own. In that case, you can modify the class definition as **`abstract`**, as shown in the following sample.

```
public abstract class Calculator
{
    // ...
}
```

*Code Listing 63*

In an **`abstract`** class, you can have **`virtual`** or non-virtual members. Additionally, you can have **`abstract`** methods. An **`abstract`** method doesn't have an implementation. Derived classes should specify the implementation and you don't want a default implementation in the base class that might not make sense. The purpose of an **abstract** method is to specify an interface that derived classes must implement. In the case of **Calculator**, you could define an **abstract Add** method as shown in the following code example.

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

*Code Listing 64*

The **Add** method has an **abstract** modifier. This method is implicitly virtual, but can't be called by a derived class because it doesn't have an implementation. The semicolon is required to terminate the **abstract** method signature. When an **abstract** class has **abstract** methods, all derived classes must **override** the **abstract** method. The **Main** method in the previous section still runs if you change the definition of the non-abstract **Calculator** class to the previous **abstract Calculator**.

Abstract classes are nice for situations where you want to have some default behavior, specify what the public interface of a class is, and support polymorphism. However, there are some limitations in that a C# class can have only one base class. Additionally, a struct can't inherit another class or struct, so they don't help if you need to write code that allows you to replace any number of value types with a base class implementation. There is an alternative, which I'll discuss next.

## Exposing Interfaces

If you only wanted a base class that specified an interface for a common set of operations, you could create an abstract class with only abstract methods. This ensures that all derived classes have those abstract methods. However, there's a better alternative, named after what it does: an **interface**.

The benefit of the **interface** type is that both **class** and **struct** types can inherit multiple interfaces. You can also implement polymorphism with interfaces. They don't have any implementation and you must write the implementation in your derived class. The following code listing shows the **Calculator** class rewritten as an interface.

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

*Code Listing 65*

Instead of **class** or **struct**, you'll use the **interface** type. A common convention for interface identifiers is the **I** prefix, as in **ICalculator**. Interface methods are implicitly public and virtual, so you don't need access, abstract, or virtual modifiers. Like **abstract** methods, **interface** methods have a signature, but don't have an implementation. Developers provide that implementation in their classes that derive from interfaces. The following code sample is a revision of the previous classes to implement the **`ICalculator`** interface.

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

*Code Listing 66*

Deriving from an interface uses the same syntax as deriving from a class in that you add a colon and interface name after the class name. Unlike virtual methods, you don't use an **`override`** modifier on methods.

A derived class implementation must be **`public`**. This make sense because an interface defines a contract that any derived class will have members defined in the interface. That means any time you use a class through its interface, you know that it will have the members defined by an interface. The following code example is a modification of the **`Main`** method that uses the **`ICalculator`** interface.

```
using System;

public class Program
{
    public static void Main()
    {
        ICalculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        ICalculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");
        Console.ReadKey();
    }
}
```

*Code Listing 67*

The only syntax difference between this example and the one previous to that is the compile-time type of **`sciCalc`** and **`prgCalc`** is **`ICalculator`**. Because each variable is an **`ICalculator`**, you can be guaranteed that the runtime type implements the members of that interface.

Interfaces can also inherit other interfaces. In that case, derived classes must implement all members of each interface in the inheritance chain. Also, a **`class`** or **`struct`** can implement multiple interfaces, which is demonstrated in the following sample.

```
public interface ICalculator { }
public interface IMath { }

public class ScientificCalculator : ICalculator, IMath
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

*Code Listing 68*

After the first interface, additional interfaces appear in a comma-separated list. A class or struct must implement the methods of all interfaces it is derived from.

## Object Lifetime

The lifetime of a value type (**`struct`** or **`enum`**) depends on where it's allocated. Parameter and variable value type instances reside on the stack and exist for as long as they are in scope. Reference type instances (**`class`**) begin life when their constructors execute. The CLR allocates their space on the managed heap, and they exist until the CLR garbage collector (GC) cleans them up.

You can use constructors to initialize a class. While doing so, you can also affect initialization of static state, base types, and other constructor overloads. The following demo shows several features of class initialization.

```
using System;

public class Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    public DateTime Created { get; } = DateTime.Now;

    static Calculator()
    {
        Console.WriteLine("static Calculator()");
    }

    public Calculator()
    {
        Console.WriteLine("public Calculator()");
    }

    public Calculator(int val)
    {
        Console.WriteLine("public Calculator(int)");
    }
}
```

*Code Listing 69*

**`Calculator`** has a **`static`** constructor and two instance constructor overloads. A **`static`** constructor executes one time for the lifetime of the object and before the first constructor executes. The following sample is a derived class with similar members.

```
using System;

public class ScientificCalculator : Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    static ScientificCalculator()
    {
        Console.WriteLine("static ScientificCalculator()");
    }

    public ScientificCalculator() : this(0)
    {
        Console.WriteLine("public ScientificCalculator()");
    }

    public ScientificCalculator(int val)
    {
        Console.WriteLine("public ScientificCalculator(int)");
    }

    public ScientificCalculator(int val, string word) : base(val)
    {
        Console.WriteLine("public ScientificCalculator(int, string)");
    }
    public double EndAngle { get; set; }
}
```

*Code Listing 70*

**`ScientificCalculator`** derives from **`Calculator`** and has similar constructors, except for the **`this`** and **`base`** operators. Using the **`this`** operator calls the constructor overload with the matching parameters. Since **`0`** is an **`int`**, the default (no parameter) constructor calls **`ScientificCalculator(int val)`** first. The **`base`** operator calls the matching constructor in the base class, so calling **`base(0)`** calls **`Calculator(int val)`** first. The following code listing is a program that instantiates these classes.

```
using System;

class Program
{
    static void Main()
    {
        var calc1 = new ScientificCalculator();

        var calc2 = new ScientificCalculator(0, "x")
        {
            EndAngle = 360
        };

        Console.ReadKey();
    }
}
```

*Code Listing 71*

And here is the program's output:

```
static ScientificCalculator()

static Calculator()

public Calculator()

public ScientificCalculator(int)

public ScientificCalculator()

public Calculator(int)

public ScientificCalculator(int, string)
```

Viewing the output, you can see what executes first. Here are the rules that govern the instantiation of these classes:

- Static constructors execute before instance constructors.

- Static constructors execute one time for the life of the program.
- Base class constructors execute before derived class constructors.
- The **`this`** operator causes an overloaded constructor that matches the **`this`** parameter list to execute first.
- The base class default constructor executes, unless the derived class uses base to explicitly select a different base class constructor overload.
- This is not shown in the output of the previous example, but static fields initialize before the static constructor and instance fields initialize before instance constructors.
- Auto-implemented property initializers, such as **`Created`**, initialize at the same time as fields.
- Properties in object initialization syntax execute last as object initialization syntax is equivalent to populating the property through the instance variable after instantiation.

***Note: In Visual Studio, you can set break points in the code and use the Immediate Window to inspect field values. You can experiment with different object initialization scenarios to get a feel for the initialization sequence.***

Of all these lifecycle events, garbage collection (GC) is the least predictable. The CLR optimizes resources and runs GC when it needs to. This means that the lifetime of a reference type object is non-deterministic. There's a rich body of theoretical discussion around the how and why of GC, but I'll restrict that debate to the practical consideration of resource management. This includes closing files, database connections, operating system handles, and more.

To release resources, there's a pattern commonly referred to as the Dispose Pattern. It relies on the **`IDisposable`** interface, flags that manage the disposal state of the object, and a destructor. The following code has constructor and **`Dispose`** method comments that imply a scenario where the class could be logging operations during its lifetime, and the log should be opened during instantiation and closed when the object is no longer needed.

```
using System;

public class Calculator : IDisposable
{
    static Calculator()
    {
        // Initialize log file stream.
    }

    #region IDisposable Support
    private bool disposedValue = false; // To detect redundant calls.

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
        {
            if (disposing)
            {
                // TODO: dispose managed state (managed objects).
                // Close log file stream.
            }
            // TODO: free unmanaged resources (unmanaged objects) and override a
finalizer below.
            // TODO: set large fields to null.

            disposedValue = true;
        }
    }

    // TODO: override a finalizer only if Dispose(bool disposing) above has code to
free unmanaged resources.
    // ~Calculator() {
    //   // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
    //   Dispose(false);
    // }

    // This code added to correctly implement the disposable pattern.
    public void Dispose()
    {
        // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
        Dispose(true);
        // TODO: uncomment the following line if the finalizer is overridden above.
        // GC.SuppressFinalize(this);
    }
    #endregion
}
```

*Code Listing 72*

The code between **`#region`** and **`#endregion`** is automatically generated by VS. To generate this code, select **`IDisposable`** in the editor and the Quick Action icon (a light bulb) will appear. Open the Quick Action menu and select **`Implement interface with Dispose pattern`**. The **`#region`** and **`#endregion`** let VS fold the code so you won't have to see it in the editor.

The **`Calculator`** class implements the **`IDisposable`** interface, which is only the **`Dispose`** method. The constructor initializes a resource you want to open, like a file handle or database, and the GC calls the destructor, **`~Calculator()`**, if it's uncommented. The **`Dispose()`** method calls **`Dispose(bool)`** with a **`true`** argument and **`~Calculator()`** calls **`Dispose(bool)`** with a **`false`** argument. This lets **`Dispose(bool)`** know whether it should clean up managed resources that belong to the CLR or unmanaged resources that belong to the operating system. The flag **`disposedValue`** helps to prevent the object from being disposed more than one time.

The following sample shows how calling code can use this class, disposing it when it is no longer needed.

```
        ScientificCalculator calc3 = null;
        try
        {
            calc3 = new ScientificCalculator();
            // Do stuff.
        }
        finally
        {
            if (calc3 != null)
                calc3.Dispose();
        }
```

*Code Listing 73*

This shows the reason **try-finally** exists, to guarantee that resources can be closed or disposed. Because the **finally** block is guaranteed to execute after code in the **try** block starts, **calc3** can be safely disposed. While that works, it’s more verbose than necessary. The following listing simplifies the code.

```
        using (var calc4 = new ScientificCalculator())
        {
            // Do stuff.
        }
```

*Code Listing 74*

The **using** statement accepts parameters with any type that implements **IDisposable**. It takes care of calling **Dispose()** after the block completes execution. Behind the scenes, the logic is similar to the previous **try-finally** block.

## Summary

C# supports object-oriented programming. For inheritance, you have single inheritance for classes, multiple inheritance for interfaces, and structs that can only inherit interfaces. Use abstract classes for classes that shouldn’t stand alone, but provide interface and structure to derived classes. Use interfaces when you don’t have implementation, need value type (struct) polymorphism, or need to implement multiple interfaces. I also discussed structs and how they are ideal for situations where copy by value leads to performance gains and value type semantics make sense. Unlike interfaces that you need to be public, use encapsulation to hide the internal workings that you don’t want other developers to use in their code. Polymorphism is a powerful concept that allows you to write a single algorithm that is coded the same for every instance, yet allows each instance to vary with an implementation specific to the runtime type of the instance. Pay attention to the sequence of object instantiation to ensure your types initialize correctly. If you need to dispose a type, make that type implement **IDisposable** with the Dispose Pattern. You can use a **using** statement to simplify the instantiation and safe cleanup of the type.

# Chapter 5  Handling Delegates, Events, and Lambdas

Much of the user interface work you'll do is event based, and C# supports this through type members called events. For events to work, you need some infrastructure to specify methods that can be called, which surfaces through delegates and lambdas. This chapter will explain how delegates, events, and lambdas work in C#.

## Referencing Methods with Delegates

Delegates have a few capabilities in C#: referencing methods, dispatching multiple methods, asynchronous execution, and event typing. This can be confusing because many other language features serve only a single purpose. Differentiating and comparing all of these capabilities of C# delegates adds complexity that you might not be familiar with. This discussion is going to cut the feature list somewhat to hopefully illuminate delegates and make them less complex as you move forward with practical implementation. In particular, I'll focus on delegates as method references and event types.

***Note: I'll avoid deep discussion of delegate multi-cast and asynchronous execution because they're rarely used and largely replaced by other language features. For example, events support multi-cast dispatch and C# 5.0 introduces a capability referred to as async.***

Let's first examine the role of a delegate as a reference to a method. To do this, the delegate specifies the signature of a method that it can reference, like this:

```
public delegate double Add(double num1, double num2);
```

*Code Listing 75*

You might notice that a delegate looks like an abstract method, except it has the **`delegate`** type definition keyword. A **`delegate`** definition is a reference type, just like a class, struct, or interface. The previous **`delegate`** definition is for a delegate type named **`Add`** that takes two **`double`** arguments and returns a result of type **`double`**. Just like other types, delegate accessibility can only be **`public`** or **`internal`** and is **`internal`** by default.

There are esoteric uses of delegates that I won't get into, but I do want to focus on the most practical and common way to use delegates: as event types.

## Firing Events

Events are type members that allow a type to notify other types of things that have happened. A very common example is a user interface with a button. You'll want to write code that does something when a user clicks that button. Here are the pieces you need to make that happen:

1. A **`Button`** class, which is typically supplied by the UI technology you're using.
2. An event member of the **`Button`** class, named **`Clicked`**.
3. The UI technology takes care of knowing when that **`Clicked`** event should fire. I'll use a **`SimulateClick`** method in an upcoming code listing.
4. The event has a delegate type. Only methods that conform to the signature of that delegate type can assign to this delegate.
5. Your code defines a method to be called when that event fires.
6. The method you write must have a signature that matches the delegate type of the event. If the method signature doesn't match the event delegate type signature, the compiler won't let you assign that method to the delegate.

As you can see, delegates have a lot of moving parts. In particular, pay attention to #6. Delegates prevent you, or anyone, or anything from assigning an arbitrary method to an event. Here's an example that defines a delegate and a class with an event of that delegate type.

```
using System;

public class ClickEventArgs : EventArgs
{
    public string Name { get; set; }
}

public delegate void ClickHandler(object sender, ClickEventArgs e);

public class CalculatorButton
{
    public event ClickHandler Clicked;
}
```

*Code Listing 76*

An event can be a member of a class, struct, or interface. If it is an interface member, it means that classes or structs that implement that interface must also have the event in their definitions. An event has the **`event`** modifier and adheres to the same accessibility rules as other type members like methods and properties.

If a delegate serves the purpose you need, you can use it. In fact, the FCL includes many reusable types, including reusable delegate types that you can use without needing to create your own. There's even a .NET type named **`EventHandler`** that nearly matches the signature of **`ClickHandler`**, where the **`sender`** is typically the source of the event, and **`EventArgs`** is a base class you can derive from to create your own custom type for sharing information with methods and event calls when fired. The previous code, with **`ClickEventArgs`**, derives from **`EventArgs`**, a type that comes with the .NET Framework. The following example simulates an event to demonstrate how to write a method that handles that event in code.

```
using System;

public class CalculatorButton
{
    public event ClickHandler Clicked;
    public void SimulateClick()
    {
        if (Clicked != null)
        {
            ClickEventArgs args = new ClickEventArgs();
            args.Name = "Add";

            Clicked(this, args);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Program prg = new Program();
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += new ClickHandler(prg.CalculatorBtnClicked);
        calcBtn.Clicked += prg.CalculatorBtnClicked;

        calcBtn.SimulateClick();

        Console.ReadKey();
    }

    public void CalculatorBtnClicked(object sender, ClickEventArgs e)
    {
        Console.WriteLine(
            $"Caller is a CalculatorButton: {sender is CalculatorButton} and is named
{e.Name}");
    }
}
```

*Code Listing 77*

Again, this example has a lot of moving parts, but they follow the list at the start of this section about defining and using events. First, notice that **`CalculateButton`** has a new method, **`SimulateClick`**. Since we simplified the code by avoiding the UI, we have to fake a user clicking a button. That said, **`SimulateClick`** demonstrates the proper way to fire an event in your own code. Before firing an event, make sure that a user has assigned methods to the event by checking for **`null`**. Whenever no methods are assigned, the event will be **`null`**. **`SimulateClick`** sets up the **`ClickEventArgs`** parameter. In this case, it's only a **`Name`** property, but you would provide any relevant information available for the **`EventArgs`** type you were using and what information a method that received this event might need. Next, fire the event by calling it like a method. This causes the event to call each method assigned to it, one by one, in the order they were assigned. The first parameter is the **`this`** keyword, which indicates that the instance of the containing type, **`CalculatorButton`**, gets passed to the methods. The second is the **`ClickEventArgs`** variable that was previously instantiated and had its **`Name`** property set.

The **`Main`** method shows how to assign methods to events. Notice the **`+=`** operator is used twice to assign two methods to the **`CalculateButton`** instance, **`calcBtn`**, and **`Clicked`** event. The **`CalculatorBtnClicked`** method is an instance method, so the **`prg`** instance provides access to that method during assignment.

The first assignment creates a new instance of the **`ClickHandler`** delegate. Delegates are types and you can instantiate them. You instantiate delegates with a method, which becomes the method the delegate refers to. Remember how I explained that delegates are references to methods? In this case, the new instance of the **`ClickHandler`** delegate refers to the **`CalculatorBtnClicked`** method. The second assignment shows a newer and simpler syntax for accomplishing the same task as the first; it just uses the method name. This is called delegate inference and means that since the method assigned to the event has the same signature of the event's delegate, the C# compiler will take care of instantiating the delegate with that method behind the scenes for you.

Finally, calling **`SimulateClick`** on the **`CalculatorButton`** instance, **`calcBtn`**, fires the event as explained previously. Regardless of whether the program assigns the same method to the event twice, firing the event causes all assigned delegates to fire, which calls the methods assigned to each delegate to execute. Therefore, the **`CalculatorBtnClicked`** method will execute two times.

You might wonder why I had to define **`SimulateClick`** inside of **`CalculatorButton`** instead of just firing the event from **`Main`**. The reason is because external code can't fire an event. An event can only be fired from inside of its containing type.

Instead of assigning named methods, you can assign code blocks directly to events.

## Working with Lambdas

A lambda is a nameless method. Sometimes you have a block of code that serves one specific purpose and you don't need to define it as a method. Methods are nice because they let you modularize your code and have something to refer to with delegates and call from multiple places. But a lot of times you just need to run some code for a specific operation. Lambdas are quick and simple ways to assign and execute a block of code.

***Note: A lambda is also a very sophisticated language feature that lets you translate between parse trees and code. This is a core feature of Language Integrated Query (LINQ), which I'll discuss more in Chapter 7. For daily practical use, working with lambdas as parse trees is rare.***

Just like methods, lambdas can have parameters, a body, and can return values. The following code listing is an example of a lambda.

```
using System;

public class Program
{
    public static void Main()
    {
        Action hello = () => Console.WriteLine("Hello!");
        hello();

        Console.ReadKey();
    }
}
```

*Code Listing 78*

**`Action`** is a reusable delegate in the .NET Framework, and **`hello`** is variable of the **`Action`** delegate type. The lambda starts with an empty parameter list, meaning no parameters. The **`=>`** operator separates the parameter list and body and is referred to as either “such that” or “goes to”. Next, you see the body, which is a single statement. Since hello is a delegate, you can call it just like a method and it will execute the lambda, which prints “Hello!” to the console.

With a single statement, you don’t need to use curly braces on the body, but you do with multiple statements, as in the following example.

```
using System;

public class Program
{
    public static void Main()
    {
        Predicate<string> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }



    public static void ValidateInput(Predicate<string> validator)
    {
        string input = "Hello";
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

*Code Listing 79*

The previous example assigns a lambda to the .NET Framework's **`Predicate`** delegate, which is designed to return a **`bool`**. The lambda has a single parameter, **`word`**, of type **`string`**. Since the lamba has more than one statement, it requires curly braces. The example shows how to pass the lambda both as a variable and as an entire lambda.

**`Predicate`** is a generic delegate. The type parameter is set to **`<string>`**, meaning that the lambda parameter is type **`string`**. You'll learn more about generics in Chapter 6.

The **`ValidateInput`** method passes a **`string`** to **`validator`** and assigns the results to the **`isValid`** variable. It's just like a method call, except there isn't a method, just code; it is quick to write and limited in scope.

Another way to use lambdas is with events. The following example shows a different way to write an event handler method for the **`Clicked`** event in the previous **`CalculatorButton`** example.

```
using System;

public class Program
{
    public static void Main()
    {
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += (object sender, ClickEventArgs e) =>
        {
            Console.WriteLine(
                $"Caller is a CalculatorButton: {sender is CalculatorButton} and is 
named {e.Name}");

            Console.WriteLine(message);
        };

        calcBtn.SimulateClick();

        Console.ReadKey();
    }
}
```

*Code Listing 80*

The first thing you might notice in this example is that the delegate assignment to the **`Clicked`** event is now a lambda. If you have two or more parameters, they must be enclosed in parentheses as a comma-separated list. If the body of the lambda includes two or more lines, they must be terminated with semicolons and enclosed in braces. Notice that the signature of the lambda matches the **ClickEventHandler** defined previously.

## More FCL Delegate Types

In addition to the **Action** and **Predicate<T>** delegates you've seen in previous examples, the FCL has a set of delegates named **Func<T>** that you can reuse at will. Here's an example that rewrites the previous example, using **Func<T, TResult>** instead of **Predicate<T>**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class Program
{
    public static void Main()
    {
        Func<string, bool> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }

    public static void ValidateInput(Func<string, bool> validator)
    {
        string input = "Hello";
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

*Code Listing 81*

This is nearly identical to the previous program, except it uses **Func<string, bool>** instead of **Predicate<string>**. As mentioned previously, both **Func<T, TResult>** and **Predicate<T>** are generic delegates. The type specifications in angle brackets are plug-ins for types applied to parameters and return types. The following listing shows the **Predicate<T>** delegate as defined in the FCL.

```
public delegate bool Predicate<T>(T obj);
```

*Code Listing 82*

It refers to a method that returns a **`bool`**, but accepts a parameter of type **`T`**. So, **`Predicate<string>`** means that the parameter to the method referred to is a **`string`**. Similarly, here's the FCL definition of **`Func<T, TResult>`**.

```
public delegate TResult Func<T, TResult>(T arg);
```

*Code Listing 83*

This shows that **`Func<T, TResult>`** accepts a parameter of type **`T`** and returns a value of type **`TResult`**. In Code Listing 81, **`Func<string, bool>`** refers to a method with a parameter of type **`string`** that returns type **`bool`**.

For convenience, the FCL offers 18 overloads of the **`Func`** delegate, allowing between 0 and 16 input parameters and 1 return parameter type. This covers many scenarios, and you can reuse the provided delegates from the FCL to go a long way. You can create your own delegates only when you need to.

## Expression-Bodied Members

While not necessarily lambdas, expression-bodied members give you some shorthand syntax for properties and methods. The following listing provides an example.

```
using System;

class Program
{
    public static string Today => DateTime.Now.ToShortDateString();

    public static void Log(string message) => Console.WriteLine(message);

    public static void Main()
    {
        Log($"{Today} is a good day.");

        Console.ReadKey();
    }
}
```

*Code Listing 84*

The **`Program`** class defines the **`Today`** property and **`Log`** method as expression-bodied members. **`Main`** shows how these members are used like normal methods and properties.

## Summary

You learned about delegates, events, and lambdas. A delegate is a reference to a method. You can pass a delegate around in code or assign it to an event. The method a delegate refers to must have the same signature of the delegate. An event is a type member, defined with a delegate type. You can assign as many delegates to an event as you need and each one executes when the event fires. You can only fire an event from within the type the event is defined in. Instead of methods, you can use lambdas when you don't need to define a separate method. You can refer to a lambda with a delegate, pass a lambda as a parameter, or assign a lambda to an event. Expression-bodied members let you write properties and methods with a shorthand syntax.

# Chapter 6  Working with Collections and Generics

You've seen arrays in previous chapters and they can be useful in scenarios where you need a fixed size, strongly typed list of objects. However, there are many times when you need to organize objects into different types of data structures like lists, queues, stacks, and dictionaries. These capabilities are available to C# developers via collection classes in the .NET Framework.

A core part of working with collections is the use of generics, which allow you to use parameterized code. This allows you to strongly type your collections. You can even write your own code that uses generics, allowing you to create strongly typed reusable libraries.

***Note: The first version of .NET offered a collection library based on `Object` which isn't strongly typed. Since all .NET types are assignable to `Object`, this worked. However, you had to write a lot of code that used cast operators to convert from `Object` back to the type you added to the collection. Generics solves this problem, and using generic collections is standard practice in .NET today.***

## Using Collections

.NET collection classes let you work with data in many different ways. Instead of an array, you can use a **`List`**. If you need a first-in first-out set of items, you can use a **`Queue`**. If you need to work with items that have unique IDs, you can use a **`Dictionary`**. With generics, you can build your own collection to manage data any way that you need.

***Tip: Check out the System.Collections.Generic namespace before writing your own collection; you might find that what you need is already written.***

A common collection is a **`List`**, which is a nice replacement for an array. The following listing provides an example.

```
using System;
using System.Collections.Generic;

public class Company
{
    public string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        List<string> names = new List<string>();
        names.Add("Joe");
        names.Insert(0, "Car");
        names.Add("Jill");
        names[0] = "Building";
        names.RemoveAt(0);
        Console.WriteLine($"First name: {names[0]}");

        IList<Company> companies = new List<Company>
        {
            new Company { Name = "Syncfusion" },
            new Company { Name = "Microsoft" },
            new Company { Name = "Acme" }
        };

        foreach (Company cmp in companies)
            Console.WriteLine(cmp.Name);
        Console.ReadKey();
    }
}
```

*Code Listing 85*

The previous program demonstrates the versatility of collections. You have a **`List`** of **`string`**, which only holds objects of type **`string`**. This is a generic collection, meaning that its type parameters, inside **`<`** and **`>`**, specify the type of object the collection works with. Each item added is appended to the list and the list grows dynamically. The **`Insert`** operation adds a new **`string`** at the first position of the list and pushes down the first, **`"Joe"`**, into the second position at index 1. The second **`Add`** puts **`"Jill"`** at index 2. Notice how you can use indexer (array-like) syntax to access elements of the list. The **`RemoveAt`** deleted the string at the first index of the collection, moving **`"Joe"`** to 0 and **`"Jill"`** to 1.

The second **`List`** in **`Main`** shows how to use custom types. Since **`List`** derives from **`IList`**, you can assign the instance to that interface. This is convenient because it means you can create code that operates on an **`IList`**; whether the caller passes in a **`List<T>`** or any other collection type that derives from **`IList`**, your code will still work.

The example also uses collection initialization syntax, where you can instantiate a comma-separated list of the collection type that populates the **`List`**. The **`foreach`** statement iterates through the collection, printing each item.

The previous example uses a **`foreach`** loop, but you could also use the **`ForEach`** method of **`List`**, as shown in the following example.

```
        List<Company> companyList = companies as List<Company>;
        companyList.ForEach(cmp => Console.WriteLine(cmp.Name));
```

*Code Listing 86*

The first line uses the **`as`** operator to convert the **`IList<Company>`** to **`List<Company>`**. With an instance of **`List<T>`**, you can call the **`ForEach`** method, which takes a lambda parameter. This lambda executes for each of the items in the **`List<T>`** and the lambda parameter, **`cmp`**, contains the current item.

This should give you an idea of how **`List`** works. There are more methods available that you can learn about by reading the documentation for the **`List`** class.

Another useful collection is a **`Dictionary`**. It works like a hash table where you store and retrieve objects by index as shown in the following sample.

```
using System;
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        Dictionary<int, Customer> customers = new Dictionary<int, Customer>();
        Customer jane = new Customer { ID = 0, Name = "Jane" };
        Customer joe = new Customer { ID = 1, Name = "Joe" };
        customers.Add(jane.ID, jane);
        customers[joe.ID] = joe;

        foreach (int key in customers.Keys)
            Console.WriteLine(customers[key].Name);

        Dictionary<int, Customer> customers2 = new Dictionary<int, Customer>
        {
            [0] = new Customer { ID = 0, Name = "Chris" },
            [1] = new Customer { ID = 1, Name = "Alex" }
        };

        Console.ReadKey();
    }
}
```

*Code Listing 87*

A **`Dictionary`** in the previous example takes two type parameters for the key and value, respectively. The first example instantiates the dictionary to take an **`int`** key and **`Customer`** value. The **`Customer`** class has two properties, where the **`ID`** will be used as a key for the dictionary. Notice the two different ways you can add values to a dictionary, via the **`Add`** method or indexer assignment. The first parameter to **`Add`** is the index and the second is the value. When using the indexer, put the index in brackets and assign the value. Just as in other collections, there are many methods available and you should review the documentation of that collection.

The **foreach** loop shows how to iterate through **Dictionary** items. A **Dictionary** has a **Keys** property, which is a collection of keys, and a **Values** property, which is a collection of values (the **Customer** instances in the previous example). Notice how the loop uses the indexer **customers[key]** to access the value associated with each key.

The second **Dictionary** in the example shows how to use the dictionary initializer syntax. Just assign the value to the index that matches the key for that value.

## Writing Generic Code

One of the primary applications of generics is to support collections. In the previous section, you saw how to use collections. You could also write your own collection class. If you wrote a generic linked list, you would need a **Node** class to hold an object and reference the next in the list, and a **LinkedList** collection class that performed list operations. The **Node** class in the following listing contains an object instance.

```
class Node<T>
{
    public T Item { get; set; }
    public Node<T> Next;

    public Node(T item)
    {
        Item = item;
    }
}
```

*Code Listing 88*

The **<T>** syntax makes the **Node<T>** class generic. Whenever code instantiates a **Node**, it specifies a type that replaces **T**. Anywhere you're using an object of that type, specify **T**. **Node<T>** doesn't have an access modifier because it's only used with the code inside this assembly and the default internal accessibility is appropriate. The following sample shows how to instantiate a **Node<T>**.

```
Node<string> name = new Node<string>(“May”);
```

*Code Listing 89*

Here, you see **Node<string>** as the type, meaning that all of the places you see **T** inside of the **Node** class are now **string**. You're protected from passing an **int**, **decimal**, or any other type to the constructor of this class because it will only hold a **string**. It is strongly typed.

Next, you need a collection that holds **Node<T>** instances as a linked list, as shown in the following listing.

```
Using System;
using System.Collections;
using System.Collections.Generic;

public class LinkedList<T> : IList<T>
{
    Node<T> head;
    Node<T> tail;

    public void Add(T item)
    {
        var node = new Node<T>(item);

        if (head == null)
            head = node;
        else
            tail.Next = node;

        tail = node;
    }

    // Other IList members…
}
```

*Code Listing 90*

The **`LinkedList`** class is generic and holds items of the type it's instantiated as. The **`IList<T>`** interface belongs to the FCL and facilitates creating collections. As you would expect with interfaces, developers who write code to the **`IList`** interface can use this collection too. The **`LinkedList`** class implements all the members of the **`IList<T>`** interface, as it must.

**`Add`** is a minimal implementation, but illustrates some concepts of working with generics. Even though the code instantiates a new **`Node<T>`**, the actual type will be the same as the type that **`LinkedList`** is defined as. The same concept applies to the interface where **`IList<T>`** becomes the same type as **`LinkedList`**. The following example instantiates a **`LinkedList<T>`**.

```
public class Program
{
    public static void Main()
    {
        var llist = new LinkedList<string>();
        llist.Add("Jamie");
        llist.Add("Ron");
        //...

        Node<string> name = new Node<string>("May");
    }
}
```

*Code Listing 91*

This shows that you instantiate and use your generic collection like any other collection. Just supply the type during instantiation and the collection will work with objects of that type.

Any place you see the **`object`** type being used is a potential candidate for creating a generic type. All types inherit the **`object`** type, which is why you'll see types in the FCL and elsewhere work with object type values.

You can also create generic methods. The following example shows a couple factory methods where one is type **`object`** and the other is generic.

```
using System;

public class CustomerReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static object Create(Type reportType)
    {
        switch (reportType.ToString())
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt = (CustomerReport)ReportFactory.Create(typeof(CustomerReport));
        Console.ReadKey();
    }
}
```

*Code Listing 92*

What you should get out of the previous **`ReportFactory`** implementation is that there's a lot of duplication in the code and the use of cast and **`typeof`** operators in the **`Main`** method includes more syntax than necessary. You can probably see where this code might become less maintainable with more complexity. The following example shows how to refactor the **Create** method into a generic method.

```
using System;

public abstract class Report { }

public class CustomerReport : Report
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report
{
    public DateTime Date { get; set; }
}



public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : Report
    {
        switch (typeof(TReport).Name)
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return (TReport)(Report)custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return (TReport)(Report)ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

*Code Listing 93*

The **Create** method has a new type parameter, **TReport**. You've seen the use of just **T** in previous examples, but sometimes—as in **Dictionary<TKey** and **TValue>**—you have to differentiate between multiple type parameters or make the code more self-documenting. The return type is now strongly typed too. The code is able to cast from the derived type to **`Report`**, and then to **`TReport`** to return the proper type. This is allowed because of the generic constraint, where **`TReport : Report`** says that **`TReport`** must derive from **`Report`**. The calling code is much simpler.

The **`Create<TReport>`** method is still longer than it has to be and contains too much duplication. We can solve that problem with generic constraints. A constraint does what the name implies: it limits how generic a type can be. You saw the base class constraint on **`Report`** in the previous code. The following table describes all available constraints.

*Table 3: Generic Type Constraints*

| Constraint | Description |
|---|---|
| interface | Type must implement specified interfaces. |
| base class | Type must derive from specified base class. |
| class | Type must be a reference type. |
| struct | Type must be a value type. |
| new | Type must have a default (no parameter) constructor. |

We need two constraints to simplify our code: **`interface`** and **`new`**. The following example shows how they can be used.

```
using System;

public interface IReport
{
    DateTime Date { get; set; }
}

public class CustomerReport : IReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report, IReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : IReport, new()
    {
        return new TReport() { Date = DateTime.Now };
    }
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

*Code Listing 94*

In this demo, there's a new interface, **`IReport`**, which **`CustomerReport`** and **`OrdersReport`** derive from. Since we know the classes we expect are **`IReport`**, we can make assumptions about the type and write code that operates on any **`IReport`**.

The **`Create<TReport>`** method has additional syntax following the method signature. To specify a constraint, follow the **`where`** keyword with the type being constrained, append a semicolon, and then add a comma-separated list of constraints from the previous table. This example uses an interface and **`new()`** constraint. The **`new()`** constraint means we can create a new instance of a type, **`new TReport()`**. Further, since the type is an **`IReport`**, we know it has a **`Date`** property and can populate its **`Date`** property. Gone are the duplication and excessive code, simplified by generic code in both implementation and use.

***Tip: You can also create generic delegates. As usual, you should seek to reuse types already present in the FCL. A popular reusable delegate in the .NET Framework is EventHandler<TEventArgs>. In fact, you can replace all the references to ClickHandler in Chapter 5 with EventHandler<ClickEventArgs> and your code will still work.***

## Summary

You've seen how to use generics and that they let you write reusable code. The .NET collection classes are more versatile than arrays and allow you to manage your objects in ways that better help the design of your application.

# Chapter 7  Querying Objects with LINQ

Language-Integrated Query (LINQ) allows you to query data with a SQL-like syntax. LINQ can be used with many different types of data and both Microsoft and third parties have built LINQ providers to access a wide range of data sources. This chapter narrows that list by showing you how to use LINQ to Objects. Once you know LINQ to Objects, understanding other LINQ providers is easy because of similar syntax.

## Getting Started

Before you write any LINQ code, remember to add a **`using`** declaration to the **`System.Linq`** namespace at the top of your file. Each example in this chapter will use the following class, containing collections to work with:

```
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
public class Order
{
    public int CustomerID { get; set; }
    public string Description { get; set; }
}

public static class Company
{
    static Company()
    {
        Customers = new List<Customer>
        {
            new Customer { ID = 0, Name = "May" },
            new Customer { ID = 1, Name = "Gary" },
            new Customer { ID = 2, Name = "Jennifer" }
        };
        Orders = new List<Order>
        {
            new Order { CustomerID = 0, Description = "Shoes" },
            new Order { CustomerID = 0, Description = "Purse" },
            new Order { CustomerID = 2, Description = "Headphones" }
        };
    }

    public static List<Customer> Customers { get; set; }
    public static List<Order> Orders { get; set; }
}
```

*Code Listing 95*

These are collections of objects in memory. To make the collection easier to query, **`Company`** is a **`static`** class, with a **`static`** constructor that initializes **`static`** properties. If you abstract this concept, that data could have been read from a file, database, or REST service. Regardless of the data source or the LINQ provider, the basic LINQ syntax remains the same.

## Querying Collections

To query data, you only need the **`from`** and **`select`** keywords. Remember to add a **`using`** clause for the **`System.Linq`** namespace. The syntax looks like SQL, as you can see in the following example.

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Program
{
    public void Main()
    {
        IEnumerable<Customer> customers =
            from cust in Company.Customers
            select cust;

        foreach (Customer cust in customers)
            Console.WriteLine(cust.Name);
    }
}
```

*Code Listing 96*

LINQ to Objects queries result in a collection of type **`IEnumerable<T>`**. In this case, it’s a collection of **`Customer`** objects. The **`from`** keyword specifies a range variable, **`cust`**, which holds each object from the collection. You specify the collection after the **`in`** keyword.

The **`select`** defines what to query. In this example, you’re just returning the whole object. In fact, the collection you get is identical to what is in **`Company.Customers`**. This isn’t particularly useful in LINQ to Objects, but is very useful if the data was read from an external data source, like a database where you just wanted to get a collection of objects into memory for further manipulation. The **`select`** allows you to reshape the data you get back into various projections. The following is a query that gets the customer name.

```
        IEnumerable<string> customers2 =
            from cust2 in Company.Customers
            select cust2.Name;
```

*Code Listing 97*

The **select** uses the **cust2** variable to access the **Name**, resulting in a collection of **string** (the **Name** property's type). Sometimes you need a whole different object, where that object might be defined as:

```
public class CustomerViewModel
{
    public string Name { get; set; }
}
```

*Code Listing 98*

And a new projection could be written as:

```
        IEnumerable<CustomerViewModel> customerVMs =
            from custVM in Company.Customers
            select new CustomerViewModel
            {
                Name = custVM.Name
            };
```

*Code Listing 99*

Here, **select** instantiates a new **CustomerViewModel**. Then it populates values, using object initialization syntax, to assign the **custVM.Name** to the new object's **Name** property. This results in a collection of type **CustomerViewModel**.

The previous example assumed you needed to work with a specifically typed collection. However, what if you don't care what type the collection is and what if you didn't want to create a new class just to do manipulation in a single algorithm? In that case, you could use an anonymous type, as shown in the following listing.

```
        var customers3 =
            from cust3 in Company.Customers
            select new
            {
                Name = cust3.Name
            };
        foreach (var cust3 in customers3)
            Console.WriteLine(cust3.Name);
```

*Code Listing 100*

Anonymous types don't have names you can use, even though C# might create an internal name for its own use. To work around this problem, use the **var** keyword as the type. Notice how the projection uses **new** without a type name: an anonymous type. You can define whatever properties you want for an anonymous type; just write them in. Notice also that you can use **var** in the **foreach** loop.

If you need to return a collection from a method, create a new (named) type and project into that. Anonymous types are designed for situations limited to the scope in which they are used. You'll see the **`var`** keyword used elsewhere in code, but the reason it was added to the language was to support this scenario. The following listing shows a common way to use **`var`**, other than the previous scenario.

```
var customer = new Customer();
```

*Code Listing 101*

The previous statement is shorter than specifying the object type of the variable, which is redundant in this case and is obviously **`Customer`**. However, the following example is less obvious.

```
var response = DoSomethingAndReturnResults();
```

*Code Listing 102*

The problem in the previous statement is that just reading the code doesn't tell you what type **`var`** is. You don't know whether it's a single object or a collection. In this case, the code might be more maintainable by specifying the type.

***Note: A common misconception is that `var` is dangerous because it behaves like `object`, allowing you to set the variable to any type. This is not true. When you use `var`, the code is still strongly typed. Once you assign a value to a variable of type `var`, you can't assign any other type to that variable. In the previous examples, `customers` is an instance of type `Customer`. You can't write code later to assign an object of another instance type—for example, an `Order` type—to that variable.***

## Filtering Data

You can filter a collection with the **`where`** clause, as shown in the following example.

```
        var customers4 =
            from cust4 in Company.Customers
            where cust4.Name.Length > 3 && !cust4.Name.StartsWith("G")
            select cust4;

        foreach (var cust4 in customers4)
            Console.WriteLine(cust4.Name);
```

*Code Listing 103*

In the previous listing, a customer's name must be longer than **3**, which filters the list down to **Gary** and **Jennifer**. The clause to the right of the **&&** operator filters that list even further to the name whose first character is not **"G"**.

In LINQ to Objects, you can create complex conditions in the **where** clause using logical operators, parentheses for grouping, and any other logic to filter results. You can even call another method that will evaluate the current object being evaluated. The result of the **where** clause must evaluate to a **bool**. Other LINQ providers might restrict the type of expressions in a **where** clause, so you'll have to review documentation for that particular provider to learn more.

## Ordering Collections

In LINQ, the **orderby** clause lets you sort collection results. The following listing demonstrates this.

```
        var customers5 =
            from cust5 in Company.Customers
            orderby cust5.Name descending
            select cust5;

        foreach (var cust5 in customers5)
            Console.WriteLine(cust5.Name);
```

*Code Listing 104*

In this example, the **orderby** clause sorts the list by the customer name in **descending** order. The default order is ascending, which you'll get by either omitting **descending** or specifying **ascending** instead. The output is:

```
May
Jennifer
Gary
```

## Joining Objects

Sometimes you'll have two different collections of objects or related tables in a database and you need to join them together. To do this, use the **join** clause.

```
        var customerOrders =
            from cust in Company.Customers
            join ord in Company.Orders
                on cust.ID equals ord.CustomerID
            select new
            {
                ID = cust.ID,
                Customer = cust.Name,
                Item = ord.Description
            };

        foreach (var custOrd in customerOrders)
            Console.WriteLine(
                $"Customer: {custOrd.Customer}, Item: {custOrd.Item}");
```

*Code Listing 105*

After the **`from`** clause, you can use one or more **`join`** clauses to access the types you need. The **`on`** keyword lets you specify the keys to match between tables. This example creates a projection on an anonymous type to create a report based on the joined information. This was a normal join, which omits any **`Customers`** where there isn't a matching **`Order`**. The following example lets you do the equivalent of a left join.

```
        var customerOrders2 =
            from cust in Company.Customers
            join ord in Company.Orders.DefaultIfEmpty()
                on cust.ID equals ord.CustomerID
            select new
            {

                ID = cust.ID,
                Customer = cust.Name,
                Item = ord.Description
            };

        foreach (var custOrd2 in customerOrders)
            Console.WriteLine(
                $"Customer: {custOrd2.Customer}, Item: {custOrd2.Item}");
```

*Code Listing 106*

The difference here is the call to **`DefaultIfEmpty`**, which includes the **`Customer`** with the **`Name Gary`**, even though there aren't any orders in the join that match his **`ID`**.

## Using Standard Operators

You've seen basic LINQ syntax, but there's much more available in the form of standard query operators. There are literally dozens of standard query operators, and you can view all of them on MSDN at https://msdn.microsoft.com/en-us/library/vstudio/bb397896(v=vs.120).aspx.

The following code listings are a grab bag of examples, demonstrating how to use standard query operators that you might find useful.

So far, you've been working with **`IEnumerable<T>`**, where **`T`** is the projected type of the query. There are a set of standard query operators that will return different collection types, including **`ToList`**, **`ToArray`**, **`ToDictionary`**, and more. Here's an example that turns the results into a **`List`**.

```
        var custList =
            (from cust in Company.Customers
             select cust)
            .ToList();
        custList.ForEach(cust => Console.WriteLine(cust.Name));
```

*Code Listing 107*

The previous code enclosed the query in parentheses and then called the **`ToList`** operator. The **`ForEach`** method on **`List<T>`** lets you pass a lambda.

LINQ queries use deferred execution. This means that the query doesn't execute until you execute a **`foreach`** loop or call one of the standard query operators, like **`ToList`**, that requests the data.

You've seen how the C# **`select`**, **`where`**, **`orderby`**, and **`join`** keywords help build queries. Each of these queries have a standard query operator equivalent. These standard query operators use a fluent syntax and give you a different way to perform the same query as their matching language syntax. Some people prefer the fluent style and others prefer the language syntax, but the method you choose is really a personal preference. The following is an example of the **`Where`** and **`Select`** operators, which mirror the **`where`** and **`select`** language syntax clauses.

```
        var customers6 =
            Company.Customers
                   .Where(cust => cust.Name.StartsWith("J"));
        foreach (var cust6 in customers6)
            Console.WriteLine(cust6.Name);

        var customers7 =
            Company.Customers.Select(cust => cust.Name);
        foreach (var cust7 in customers7)
            Console.WriteLine(cust7);
```

*Code Listing 108*

The **`Where`** lambda must evaluate to a **`bool`** and the **`Select`** lambda lets you specify the projection.

You can perform set operations like **`Union`**, **`Except`**, and **`Intersect`**. The following listing is an example of **`Union`**.

```
        var additionalCustomers =
            new List<Customer>
            {
                new Customer { ID = 1, Name = "Gary" }
            };
        var customerUnion =
            Company.Customers
                   .Union(additionalCustomers)
                   .ToArray();
        foreach (var cust in customerUnion)
            Console.WriteLine(cust.Name);
```

*Code Listing 109*

Just pass a compatible collection and **Union** will produce a combined collection of all objects. I used the **ToArray** operator in this example too, which results in an array of the collection type, **Customer**.

There is a useful set of operators for selecting **First**, **FirstOrDefault**, **Single**, **SingleOrDefault**, **Last**, and **LastOrDefault**. The following example demonstrates **First**.

```
        Console.WriteLine(Company.Customers.First().Name);
```

*Code Listing 110*

The only thing about using **First** this way is the possibility of an **InvalidOperationException** with the message “Sequence contains no elements.” This sequence contains elements, but this isn’t guaranteed. You would be safer using the operator with the **OrDefault** suffix, as in the following listing.

```
        var empty =
            Company.Customers
                   .Where(cust => cust.ID == 999)
                   .SingleOrDefault();

        if (empty == null)
            Console.WriteLine("No values returned.");
```

*Code Listing 111*

The previous example writes **"No values returned."** Because there isn’t a customer with **ID == 999**, the **SingleOrDefault** returns **null**, which is the default value of a reference type object.

These were only a handful of available operators, but hopefully you have a sense for the wealth of support in language syntax as well as the standard query operators that comprise LINQ.

## Summary

LINQ allows you to use SQL-like syntax to query data. The LINQ provider used in this chapter is LINQ to Objects, which lets you query objects in memory, but there are many other LINQ providers for other data sources. Use a **from** to specify the collection being queried and a **select** to shape the results. The **where** clause lets you filter results and takes a **bool** expression to evaluate if a given object should be included. The **orderby** clause lets you sort results. The **join** clause lets you combine two collections. Standard query operators extend LINQ and make it even more powerful than the language keywords.

# Chapter 8 Making Your Code Asynchronous

In version 5, C# introduced the capability to write and call code asynchronously, commonly referred to as async. To understand async, it's useful to consider the normal behavior of code, which is synchronous. In synchronous code you call a method, wait for it to complete, and move on to the rest of the code. The primary point of this behavior is that the thread calling the synchronous method is also executing the synchronous code in that method. If that synchronous method runs for a long time, your UI will become unresponsive and your users might not know whether the program crashed or if they should just wait.

Asynchronous code improves this situation by allowing the long-running operation to continue on a separate thread, and free your calling thread to resume its responsibilities. When the calling thread is the UI thread, the application becomes responsive again and you can show a status or busy indicators, or let the user operate another part of the program while the asynchronous process runs. When the asynchronous process returns, you can interact with users in some way, if that makes sense for your application. In the past, writing this asynchronous code has been a challenge. Though the task of writing asynchronous code has improved with new patterns and libraries, the C# async features makes asynchronous programming much easier.

There are a couple different perspectives of async that determine how you code: library consumer or library creator. From a consumer perspective, you make assumptions about async code based on an implied contract. However, from the library creator perspective, you have additional responsibilities to ensure your code provides the async contract users expect.

## Consuming Async Code

C# has two keywords that support async: **`async`** and **`await`**. Decorating a method with the **`async`** modifier says that the method can contain async code. You use the **`await`** keyword on a **`Task`** to start an async operation.

```
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        Program.CreateFileAsync("test.txt").Wait();
    }




    public static async Task CreateFileAsync(string filename)
    {
        using (StreamWriter writer = File.CreateText(filename))
            await writer.WriteAsync("This is a test.");
    }
}
```

*Code Listing 112*

In the previous program, the **`CreateFileAsync`** method is asynchronous. You can tell by the **`async`** modifier on the method. You need to add **`using`** clauses for the **`System.IO`** and **`System.Threading.Tasks`** namespaces for writing to a file and async **`Task`** support, respectively. The **`File`** class is part of the FCL and its **`CreateText`** method returns a **`StreamWriter`** that you use to write to the file.

***Note: Appending a method name with `Async` is not required, but it is a common convention.***

The proper way to call an async method is to **`await`** its **`Task`** or **`Task<T>`**. The **`WriteAsync`** method returns **`Task`**, which means you can **`await`** it.

The **`using`** statement closes the file when its enclosing block completes. In this case, the block is only a single line, so no curly braces are required.

Part of the async contract is an expectation that some code in the library you're using will run the operation on another thread, releasing your thread for other operations; that's what **`WriteAsync`** does too. So, the thread returns to the code calling this async method. But the caller in this program is the **`Main`** method, which is calling **`Wait()`** on the **`Task`** returned from **`CreateFileAsync`**. This keeps the program from ending before the thread that's running the async operation completes.

***Warning: The previous example is a console application, which doesn't have the underlying infrastructure (referred to as a synchronization context) to manage proper thread handling. Therefore, it was necessary to `Wait()` on the task returned from `CreateFileAsync`. In a normal UI application, you will have a synchronization context, meaning you won't have to worry about the program ending, and won't need to call `Wait()` on an async method. The preferred method of waiting on an async method is via `async` and `await` as shown in the `CreateFileAsync` method. In fact, you should never call `Wait` on an async method. That's because when the second thread returns from doing work on the `async` call, it will attempt to marshal the call back onto the calling thread. If that calling thread is in a synchronous `Wait()`, the thread will be blocked, preventing the second thread from performing that marshaling operation. Then you'll have a deadlock. To prevent deadlock, never call `Wait()`, use `async` and `await` instead.***

The **`async`** modifier is required on the method if you use **`await`**. If a method has the **`async`** modifier, but no **`await`**s, C# will give you a compiler warning and let you know that the method will run synchronously.

## Async Return Types

With **`async`**, you can **`await`** any awaitable type. The FCL has **`Task`** and **`Task<T>`**, which are awaitable and are what you should use in most situations. Returning **`Task`** means that the method does not return a value, which is what you saw with the previous **`CreateFileAsync`** method.

***Tip: Stephen Toub's blog post "await anything;" explains how to create a custom awaitable type and is a good reference if you see it as a way to improve your code. You can read it at [http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx).***

Use **`Task<T>`** when your method returns a value. The following listing shows an example.

```
public async Task<string> ReturnGreeting()
{
    await Task.Delay(1000);
    return "Hello";
}
```

*Code Listing 113*

**`Task.Delay`** is a way to sleep the thread by a number of milliseconds, but I'll be using it in more examples to simplify code and as a placeholder for where you would normally add async code.

The previous example shows a return type of **`Task<string>`**. The method only returns the string **`"Hello"`** instead of an instance of **`Task<string>`** because the C# compiler takes care of that for you.

An async method can return **`void`** rather than an awaitable type too. This is done in the following listing.

```
public async void SayGreeting()
{
    await Task.Delay(1000);
    Console.WriteLine("Hello");
}
```

*Code Listing 114*

This method executes asynchronously, but async **`void`** methods have important caveats you must be aware of: they aren't awaitable, and they don't protect against exceptions, but they are necessary for scenarios like event handling where the method must be **`void`**.

Since you can only await awaitable types like **`Task`** and **`Task<T>`**, there's no way to await an async **`void`** method. The implication of this is that when a library's code starts another thread, it allows the calling thread to return. Calling an async **`void`** method means you can't wait until that method completes and you won't ever know when or if the method completes. As with anything, there are no absolutes and one could argue that it would be possible to write some cross-thread communication mechanism, but I'm referring to the general out of the box behavior, which will lead to some important implications. Because of this behavior, there are pros and cons on when you should use an async **`void`** method.

The largest problem with async **`void`** methods is that you can't throw an exception back to the calling code. With **`Task`** and **`Task<T>`** returning methods, you can **`await`** and wrap the async method call in a **`try-catch`**, but you can't do that with async **`void`** methods. If an async **`void`** method throws an unhandled exception, the application will crash.

With such problems, it would be easy to assume that async **`void`** should not be used at all. However, the C# language designers added async **`void`** for one specific reason: to support event handling. Event handlers in the .NET Framework follow a pattern where their delegates return **`void`**. Therefore, you can't use an awaitable type, like **`Task`** or **`Task<T>`**, and must assign async **`void`** methods as event handlers.

In UI applications, a UI control might fire an event, async **`void`** methods assigned to the event execute, the async code starts a new thread and releases the UI thread, and the UI thread returns and processes messages to keep the UI responsive. So, using async **`void`** as event handlers is appropriate.

## Developing Async Libraries

Writing an async library is mostly normal coding, but the key thing to keep in mind is what is happening to the thread. First, all code executes by default on the calling thread. Second, you need to marshal execution onto a new thread and release the calling thread to the caller.

### Understanding What Thread the Code is Running On

The following code doesn't necessarily make any logical sense, but represents the potential structure of some library code that you might write. In particular, it demonstrates what happens with threads before and after the first **`await`** in your **`async`** method. In the following code, **`UserInfo`** is just a type to hold and return data. **`UserService`** and **`AddressService`** have async methods that the **`GetUserInfoAsync`** method calls.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

public class UserInfo
{
    public string Info { get; set; }
    public string Address { get; set; }
}

class UserService
{
    internal static async Task<string> GetUserAsync(string user)
    {
        // Do some long running synchronous processing.
        return await Task.FromResult(user);
    }
}

class AddressService
{
    internal static async Task<string> GetAddressAsync(string user)
    {
        return await Task.FromResult(user);
    }
}

public class UserSearch
{
    public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
    {
        var userName =
            (from name in names
             where name.StartsWith(term)
             select name)
            .FirstOrDefault();

        var user = new UserInfo();
        user.Info = await UserService.GetUserAsync(userName);
        user.Address = await AddressService.GetAddressAsync(userName);

        return user;
    }
}
```

*Code Listing 115*

Remember, you're writing reusable library code, so it could be called from many different technologies, such as WPF, Windows Store apps, Windows Phone, and more. What's common about these type of applications is that a user interacts with the UI controls and those UI controls fire events. This means an async **`void`** event handler awaits your **`GetUserInfoAsync`** method.

When the event handler code calls your code, it's running on the UI thread. Your code will continue running on the UI thread until some other code explicitly marshals the call to another thread and releases the UI thread.

***Note: More accurately, the thread calling your code might not necessarily be the UI thread if there was another async method that called your code and already released the UI thread. However, defensive coding is a safe approach because there exists the possibility that your code will be called on the UI thread by some developer in the future.***

Notice the LINQ query in **`GetUserInfoAsync`** before reaching the first **`await`**. That is synchronous code that runs on the calling thread, which could also be the UI thread. The issue here is that the UI thread is tied up doing work in your async method, rather than returning to the UI. Imagine a UI with a progress indicator that locks up because your async method is holding onto the UI thread and doing a lot of processing before the first async call.

The code is still on the UI thread when it calls **UserService.GetUserAsync**. I added a comment to **GetUserAsync** to represent more long-running synchronous processing that is also running on the UI thread. Finally, awaiting **Task.FromResult** releases the UI thread and the rest of the code runs asynchronously. That's because **Task.FromResult** implements the async contract properly. Before showing you how to fix this problem, let's look at the rest of the code so you can understand how it runs.

When the code returns from **Task.FromResult**, the UI thread has been released and the code is running on the new async thread. When returning from **GetUserAsync** to its caller, **GetUserInfoAsync**, the call automatically marshals back to the calling thread, which could be the UI thread. Again, this program eats up CPU cycles on the UI thread, making the application less responsive. Fortunately, there's a way to fix this problem.

## Fulfilling the Async Contract

The previous section explained how the code runs on the calling thread by default, which could be the UI thread. Whenever you call an async method in the FCL, that code will release the calling thread and continue on a new thread, which is proper behavior of the async contract that developers expect. You should do the same in your code.

To do this, use the **Task.ConfigureAwait** method, passing **false** as the parameter. The following is an example that fixes the problem in **GetUserInfoAsync**.

```
    public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
    {
        var userName =
            (from name in names
             where name.StartsWith(term)
             select name)
            .FirstOrDefault();

        var user = new UserInfo();
        user.Info = await UserService.GetUserAsync(userName).ConfigureAwait(false);
        user.Address = await AddressService.GetAddressAsync(userName);

        return user;
    }
```

*Code Listing 116*

The **GetUserInfoAsync** method appends **ConfigureAwait(false)** to the call to **GetUserAsync**. **GetUserAsync** returns a **Task<string>** and **ConfigureAwait(false)** operates on that return value, releasing the calling thread and running the rest of the method on a new async thread. That's it; that's all you have to do.

You still have the issue of synchronous processing before the first call to **`ConfigureAwait`**. Sometimes, you can't do anything about it because it's necessary to execute that code before the first **`await`**. However, if it's possible to rearrange the code to do any processing after the first **`await`**, you should do so.

## A Few More Notes on Async

I made this point previously, but I feel it bears repeating. Especially for library code, you should prefer async methods that return **`Task`** or **`Task<T>`**. In the UI you don't have a choice if you're writing an event handler. If you're using a Model View ViewModel (MVVM) architecture, you'll also need **`void Command`** handlers. You shouldn't have these issues in reusable library code and in this scenario, async **`void`** methods are dangerous.

Async void methods that throw exceptions will crash your application.

Much of the discussion in this chapter is around how async improves the user experience by releasing the UI thread. In addition to that, async also improves application performance by not blocking threads. These scenarios usually involve some type of out-of-process operation such as network communication, file I/O, or REST service calls. These operations can use Windows operating system services such as I/O completion ports to free threads while the long-running out-of-process operation executes; they can then reallocate those threads when the operation completes and needs to return to your code. In addition to performance increases through efficient thread management, you can also improve the scalability of a server application by using async to avoid blocking threads more than you have to.

For all the seeming complexity that this chapter introduces, attempting to perform many of the operations associated with managing threads for application responsiveness, performance, and scalability is made much easier through the use of async.

## Summary

Async is a useful capability that allows your application to be responsive and perform well. The user experience of async is a method with an **`async`** modifier and the ability to await an async method's **`Task`**. In addition to the user experience, there are additional considerations for writing async libraries. You should be aware of the threading behavior and how an **`async`** method runs on the caller's thread by default. Remember that you should minimize synchronous code before the first **`await`** and that you should call **`ConfigureAwait(false)`** at the earliest opportunity, releasing the UI thread and running the remaining algorithm on the new async thread.

# Chapter 9  Moving Forward and More Things to Know

To keep subject matter succinct, I've passed up features that could evolve into deeper discussions. This chapter is about some of those features, if only to highlight that they are part of the C# language and that you are likely to encounter them regularly.

## Decorating Code with Attributes

An attribute is a feature of C# that lets you decorate code with meta-information for various tools. I use the term "tool" loosely, but it could be the C# compiler, a testing framework, or a UI technology. Essentially, these tools read the attributes to make some decision on how to work with your code. I'll show you a few examples so you can be familiar with attribute syntax when you encounter it in code.

The **`Obsolete`** attribute lets you indicate that some code has been deprecated. It's a C# compiler attribute and the compiler will emit a message regarding its use. The following code shows an example.

```
using System;

public class ShoppingCart
{
    [Obsolete("Method planned for deprecation on date – use … instead.")]
    public void Add(string item) { }

    [Obsolete("Method is obsolete and can no longer be used", error: true)]
    public decimal CalculateTax(decimal[] prices) { return 0; }
}
```

*Code Listing 117*

When the C# compiler sees the **`Obsolete`** attribute decorating **`Add`**, it will show a warning with the message argument matching the parameter to the **`Obsolete`** attribute. In the second example, the compiler shows the message as an error and you won't be able to compile because the second parameter, **`true`**, indicates that the compiler should treat usage of that method as an error.

The next example uses attributes for a unit test with MSTest, Microsoft's unit testing software.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

*Code Listing 118*

As with most unit testing frameworks, MSTest has a test runner that loads the unit test code, looks for classes decorated with the **`TestClass`** attribute, and executes methods with the **`TestMethod`** attribute. It does this with another capability of C# called reflection.

## Using Reflection

Reflection gives you the ability to examine compiled .NET code. With reflection, you can build useful tools like MSTest, dynamically instantiate types and execute their code, and more. The following code uses reflection to dynamically instantiate a type and execute one of its methods.

```
using System.Linq;
using System;
using System.Reflection;

public class FinancialCalculator
{
    public decimal Sum(decimal[] numbers)
    {
        return numbers.Sum();
    }
}

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        FinancialCalculator calc =
            (FinancialCalculator)Activator.CreateInstance(calcType);
        decimal sum = (decimal)sumMethod.Invoke(calc, new object[] { prices });

        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
        Console.ReadKey();
    }
}
```

*Code Listing 119*

**FinancialCalculator.Sum** uses the LINQ **Sum** method, so add a **using** clause for **System.Linq**. Add a **using** declaration for **System.Reflection** to support reflection too.

With reflection, a **Type** instance gives you access to all of the information about a type. The **Main** method calls **GetMethod** to obtain a **MethodInfo** reference to the **Add** method, but there are many more methods that let you look at various parts of a type. As an example of a subset of capabilities available, you can call **GetMethods**, **GetProperties**, or **GetFields** to get an array of **MethodInfo**, **PropertyInfo**, or **FieldInfo** respectively. There are many more methods in the **Type** class you can use, and it's a fun exercise to write code to practice with this.

**Activator.CreateInstance** creates a new instance of the **Type** it's passed. Calling **Invoke** lets you run a method and get the results. Much of the previous reflection code is hard-coded for simplicity, but it's very useful for when you need to write code that examines the capabilities of another piece of code and optionally work with the member of a type.

## Working with Code Dynamically

C# also has a type called dynamic. Its purpose is to allow you to interoperate with dynamic languages, like IronPython and IronRuby, and makes reflection easier. Microsoft also has a technology called Silverlight. Dynamic could make working with the HTML DOM easier, but Silverlight has been largely replaced by HTML 5 as a dynamic web application technology.

The **dynamic** type lets you assign any value to a **dynamic** variable and use any typed members on that variable. Rather than the C# compiler emitting errors, any errors are handled by the CLR at runtime. You might see where this has a lot of power through coding flexibility, yet a drawback of dynamic typing is that it offers no indication of type-related errors until runtime. Using the **FinancialCalculator** class from the previous example, the following is an example of some dynamic code.

```
using System;
using System.Reflection;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        dynamic calc = Activator.CreateInstance(calcType);
        dynamic sum = calc.Sum(prices);
        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
        Console.ReadKey();
    }
}
```

*Code Listing 120*

You might notice that this code is simpler than the full reflection implementation. You don't have an interface and have no guarantee that the **`calc`** instance has a member named **`Sum`**, but since the alternative is to use reflection, you're still in the same situation of runtime evaluation. Therefore, this might be a reasonable approach for this particular scenario.

Pulling together what you learned about generics, it might be useful to improve the algorithm even further, as shown in the following listing.

```
using System;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        decimal sum = GetSum<FinancialCalculator, decimal>(prices);

        Console.WriteLine("Sum: {0}\nPress any key to continue.", sum);
        Console.ReadKey();
    }

    public static TValue GetSum<TCalc, TValue>(TValue[] prices)
        where TCalc : new()
    {
        dynamic calc = new TCalc();
        TValue sum = calc.Sum(prices);
        return sum;
    }
}
```

*Code Listing 121*

The previous example totally eliminates the need for reflection, reduces code, makes the algorithm strongly typed where it needed to be, and makes it dynamic where it helps. It would have been possible to use an interface constraint to make **`GetSum`** more strongly typed, but I used this as an exercise to help you think about where dynamic might be useful.

## Summary

Attributes are C# features that tell a tool something about your code. Reflection helps you write meta-code that can evaluate and execute other code. There is a dynamic type that lets you make assumptions about the code you're writing, interfacing with dynamic languages, and making it easy to perform reflection.

This completes *C# Succinctly*. I hope it has been useful for you. I wish you the best in your further studies.