The C# Player's Guide

Sources

- [Textbook] The C# Player's Guide (2nd Edition) (Link)

Chapter 1 – The C# Programming Language

What is the .NET Framework?

- The C# compiler turns your source code into Common Intermediate Language (CIL or IL for short)
    - When it is time to run your program, the IL code in your .exe or .dll will be handed off to the Common Language Runtime (CLR) to run it
    - As your program is running, the CLR will look at the IL code and compile it into binary executable code that the computer it is running on understands
        - For each block of code you have, this will only happen once each time you run your program – the first time it needs it. This is called Just-in-Time compiling, or JIT compiling

Chapter 3 – Hello World: Your First C# Program

Building Blocks: Projects, Solutions and Assemblies

- A project is simply a collection of source code and resource files that will all eventually get built into the same executable program
- When compiled, a project becomes an assembly in the form of either an EXE or DLL file
- A solution will combine multiple projects together to accomplish a complete task or form a complete program

A Closer Look at Your Program

- Namespaces, classes, and methods are ways of grouping related code together at various levels
- Methods are a way of consolidating a single task together in a reusable block of code
    - In other programming languages, methods are sometimes called functions, procedures, or subroutines
- When one thing is contained in another, it is said to be a member of it
- Classes are a way of grouping together a set of data and methods that work on that data into a single reusable package
    - Classes are the fundamental building blocks of object-oriented programming

Chapter 4 – Comments

How to Make Good Comments

- A few rules of thumbs for making comments are:
    - Write the comments for a chunk of code as soon as you've got the piece more or less complete
    - Write comments that add value to the code
    - You don't need a comment for every single line of code, but it is helpful to have one for every section of related code
- When you write comments, take the time to put in anything that you or another programmer may want to know if they come back and look at the code later

Chapter 5 – Variables

What is a Variable?

- A variable is a place in memory where you can store information, and the process of creating a variable is called declaring a variable

Good Variable Names

- The purpose of a variable name is to give a human-readable label for the box that you intend to put stuff in, and whoever stumbles across the variable should instantly be able to ascertain what information it is supposed to contain
    - Rule #1: Meet C#'s requirements
    - Rule #2: Variable names should describe the stuff you intend on putting in it
    - Rule #3: Don't abbreviate or remove letters
    - Rule #4: A good name will usually be kind of long
    - Rule #5: If your variables end with a number, you probably need a better name
    - Rule #6: "data", "text", "number", and "item" are usually not descriptive enough
    - Rule #7: Make the words of the variable name stand out from each other
        - As an example, using camel-case as in "playerScore"

Chapter 7 – Basic Math

Operations and Operators

- An expression is something that evaluates to a single value

Compound Assignment Operators

- Compound assignment operators use their own variables to update themselves

```
int a = 5;
a += 3;            // This is the same as a = a + 3;
```

Chapter 8 – User Input

String Interpolation

- C# 6.0 introduces a new feature called string interpolation, which lets you write a line of text and code in a much more concise and readable way than using string concatenation

```
Console.WriteLine($"The cylinder's volume is: {volume} cubic units.");
```

    - The curly braces can contain any valid C# expression (math, methods, etc.), and is not limited to just a single variable

Chapter 9 – More Math

Working with Different Types and Casting

- Implicit casting happens for you whenever you go from a narrower type to a wider type
    - This is called widening conversion, and it is the kind of conversion that doesn't result in any loss of information
- Explicit casts usually turn a value from a wider type into a narrower type, and because of that, could mean a loss of data

```
long a = 3;
int b = (int)a;
```

Chapter 10 – Decision Making

The Conditional Operator ?:

- The condition operator works a bit like an if-else statement, and is a shorthand way of implementing logic into your code

```
Console.WriteLine((score > 70) ? "You passed!" : "You failed.");
```

Chapter 11 – Switch Statements

- When it comes to the decision of using a switch vs if-else statement, switch statements are usually considered faster (depending on the situation), but it is in your best interest to pick whichever version makes the code most readable

Chapter 12 – Looping

- Three kinds of loops that C# offers are the While, Do-While, and For Loops

- Often, one type of loop is cleaner and easier to understand than the others, but they can all basically do the same stuff

Breaking Out of Loops

- Keep in mind that the break keyword can be used to exit out of the loop at any point

```
while(true)
{
    Console.Write("What is thy bidding, my master? ");
    string input = Console.ReadLine();

    if(input == "quit" || input == "exit")
        break;
}
```

Continuing to the Next Iteration of the Loop

- The continue command used inside of a loop will jump back to the start of the loop and check the conditions again, without finishing the current loop

Chapter 13 – Creating Arrays

Creating Arrays

- To create a new array and place it in the variable, you will use the new keyword and specify the number of elements that you'll have in the array
    - Once a size is set for the array, it cannot be changed

```
int[] scores = new int[10];
```

Getting and Setting Values in Arrays

- To access a specific spot in the array, we use the square brackets again, along with what is called a subscript or an index, which is a number that tells the computer which of the elements (things in an array) to access

Arrays of Arrays and Multi-Dimensional Arrays

- When each array within a larger array has a different length, it is called a jagged array
    - If they are all the same length, it is often called a square array or rectangular array

The 'foreach' Loop

- To use a foreach loop, you use the foreach keyword with an array, specifying the name of the variable to use inside of the loop
    - If you need to know what index you're at, your best bet is to use a for loop instead

```
int[] scores = new int[10];

foreach (int score in scores)
    Console.WriteLine("Someone had this score: " + score);
```

- Foreach loops enhance readability, but are somewhat slower as a result of internal mechanics

Chapter 14 – Enumerations

- Defining enumerations are a way to define your own type of variable

The Basics of Enumerations

- When we create an enumeration, we're defining a new type, which will be placed directly inside of the namespace (rather than within a class)
    - To define an enumeration, we use the enum keyword, give the enumeration a name, and list the values that your enumeration can have inside of curly braces

```
namespace Enumerations
{
    // You define enumerations directly in the namespace.
    enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

- To an enum variable a value, we will use the '.' (dot) operator, since it is used for member access

Why Enumerations Are Useful

- Enumerations force the computer (and programmers) to use only specific values, which have been previously defined
    - This prevents errors, and makes for more readable code
- Keep in mind that, under the surface, C# is simply wrapping our enumerations around an underlying type, and are just numbers (int) at their core

Chapter 15 – Methods

- Code reuse through the divide and conquer technique allows us to break our program into small manageable pieces
    - These small, reusable building blocks of code are called methods
    - In other programming languages, they are commonly called functions, subroutines, or procedures

Creating a Method

- Because a method belongs to the class, it is called a member of that class
    - Whenever we create new methods, they must be added to a specific type, and can't be floating around on their own (ex. outside of a class type)
- Note that while it is not required, the standard convention is to have method names start with an upper case letter, while variables usually start with a lower case letter

Returning Stuff from a Method

- While the return statement is often the last line of the method, it doesn't have to be
    - For instance, you can have an if statement right in the middle of a method that returns a value from a method only when a certain condition is met
    - Note that as soon as a return statement is hit, the flow of execution immediately goes back to where the method was called from – nothing more gets executed in the method

Method Overloading

- Creating multiple methods with the same name is called overloading
    - While two methods can have the same name, they can't have the same signature
    - Only use overloading when trying to do the exact same thing with different kinds of data
- A method's signature is defined as the combination of the method name and the types and order of the parameters that get passed in
    - Note that this does not include the parameter's names, just their types

```
static int Multiply(int a, int b)
{
    return a * b;
}

static double Multiply(double a, double b)
{
    return a * b;
}
```

XML Documentation Comments

- It is a good idea to add a comment by a method to describe what it is
    - This way, when you or somebody else wants to use it, they can easily figure out what it does, and how to make it work
- To add XML documentation comments, go just above the method, and type three forward slashes
    - Once populated, the information will now appear in IntelliSense

```
/// <summary>
/// Takes two numbers and multiplies them together, returning the result.
/// </summary>
/// <param name="a">The first number to multiply</param>
/// <param name="b">The second number to multiply</param>
/// <returns>The product of the two input numbers</returns>
```

Recursion

- When a method is called from inside of itself, this is called recursion

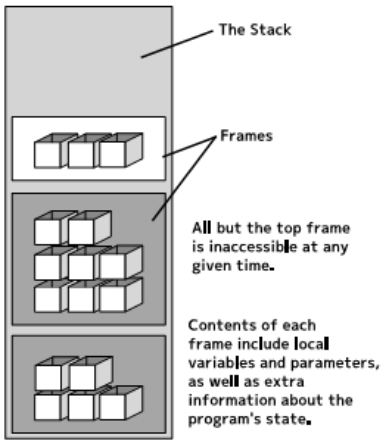
```
static int Factorial(int number)
{
    // We establish our "base case" here. When we get to this point, we're done.
    if(number == 1)
        return 1;

    return number * Factorial(number - 1);
}
```

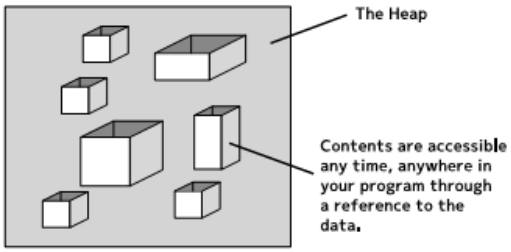
Chapter 16 – Value and Reference types

The Stack and the Heap

- When your program first starts, the operating system gives it a pile of memory for it to work with
    - The program splits this memory up mainly into two sections: the stack and the heap
- The stack is used to keep track of the current state of the program, including all the local variables that you have
    - The stack behaves like a stack of containers, where frames of information (such as the variables from a single method) are placed one on top of the other
    - A frame will also include information to keep track of the program's current state of execution, such as what line of code the program was at before entering the method
    - Like a stack of containers, you can readily access the contents of the frame on the top of the stack, but you can't really get to the containers lower down
        - When we return from a method, the top frame is taken off the stack and discarded, and we go back down to the frame beneath it
    - The debugger that is built into Visual Studio can inspect the entire stack, including the buried frames, and show them to you, through what is called a stack trace



- The heap, on the other hand, does not deal with tracking the program's state, and only focuses on storing data
    - Data within the heap can be accessed at any time, and order is not important
    - Memory within the heap is automatically managed through the .NET Framework's garbage collector
- In a multi-threaded application, each thread will have its own stack, but all threads in the program will share the same heap



References

- Since the CLR manages memory on the heap, we don't need to reference specific locations in memory using pointers, as in C++
    - Rather, C# uses a reference, which allows memory to be moved around in the heap by the CLR without losing the information that you are interested in
    - References can be thought of as a unique identifier that tells the computer where to find the rest of the data in the heap

Value Types and Reference Types

- In C#, all types are divided up into two broad categories: value types and reference types

- When you have a variable that is a value type, the actual contents of that variable live where the variable lives (on the stack)
- With a reference type, the contents of the variable live on the heap, and the variable itself only contains a reference to the actual content
  - Strings and arrays are both examples of reference types

Null: References to Nothing

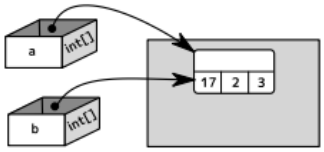
- Reference types can have a reference to nothing at all, which is called a null reference
  - Value types cannot be assigned a value of null
- Checking for null is incredibly common, and it should be noted that C# 6.0 introduced a new pair of operators that allow you to check for null using much more concise syntax

Value and Reference Semantics

- Reference types have what's called reference semantics, while value types have value semantics or copy semantics
  - With the following code, for example, the integer value would be copied independently to variable 'b', and only 'b' would be incremented

```
int a = 3;
int b = a;
b++;
```

- When an element of a reference type is changed, the change in the heap is available for all references of that element, no matter whether the reference is outside of the current method or class




```
int[] a = new int[] { 1, 2, 3 };
int[] b = a;
b[0] = 17;
Console.WriteLine(a[0]); // This will print out 17.
```

- Surprisingly, there are a lot of talented programmers who claim to know C# that haven't figured this heap vs. stack and value vs. reference type stuff yet
  - If you don't understand these differences, you will be constantly running into strange problems that just don't seem to make sense

Try It Out!
Value and Reference Types Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Anything on the stack can be accessed at any time.
2. **True/False.** The stack keeps track of local variables.
3. **True/False.** The contents of a value type can be placed in the heap.
4. **True/False.** The contents of a value type are *always* placed in the heap.
5. **True/False.** The contents of reference types are *always* placed in the heap.
6. **True/False.** The garbage collector cleans up old, unused space on both the heap and stack.
7. **True/False.** You can assign **null** to value types.
8. **True/False.** If **a** and **b** are reference types, and both reference the same object, modifying **a** will affect **b** as well.
9. **True/False.** If **a** and **b** are value types with the same value, modifying **a** will affect **b** as well.

Answers: **(1)** False. **(2)** True. **(3)** True. **(4)** False. **(5)** True. **(6)** False. **(7)** False. **(8)** True. **(9)** False.

Chapter 17 – Classes and Objects

Modeling the Real World

- Mimicking the real world with code is one of the best ways to construct your program, since it breaks down into manageable pieces, and presents an intuitive way to work with the code
  - If we wanted to describe cars in class form, the 'car' category would be the class, a 'Ford Pinto' could be the object (or instance) created from the class, the 'speed' and 'range' could be properties, and the 'left turn' or 'accelerate' could be methods

- Because a class combines a collection of variables and methods, you can also think of a class as simply a way of packaging related data and methods together into one single new cohesive piece
- A constructor is a method that is called when creating an instance of a class, which describes how to create or build a new object that has that type
    - Constructors can either have no parameters, or include several them, just like any other method

Using an Object

- We can access our object's behaviors through the methods it has defined in it
    - The dot operator is used to access members that belongs to a type

```
Random random = new Random();
int nextRandomNumber = random.Next();
```

The Power of Objects

- Classes and the objects that they create are the fundamental building block of programming in object-oriented programming languages
    - In C#, nearly all your code will belong to a particular class (or another custom-made type)
- The fact that classes and objects are modeled after real world objects means that you can build your code in a way that will intuitively make sense to anyone trying to use it, and that's worth a lot
    - They also provide us with reusable pieces of code that break a complex program into small pieces that are easily managed

Classes are Reference Types

- It is critical to point out that all classes are reference types
    - For example, objects derived from classes can have null assigned to them, and two variables that are assigned the same reference will affect one another, due to reference semantics
- Keep in mind that if an instance of a class is passed into a method, whatever is done to the class object in the method will also affect the object outside of the method, due to the both objects referencing the same location in the heap

```
public static void Main(string[] args)
{
    Random random = new Random();
    DoSomething(random);
}

public static void DoSomething(Random r)
{
    // The variable 'r' references the same exact object as 'random' in Main.
    // The two variables are two different references that refer to the same object.
    Console.WriteLine(r.Next());
}
```

Chapter 18 – Making Your Own Classes

Creating a New Class

- It is best practice to place each class in its own file, rather than having multiple classes per file
    - This helps to keep your files to a manageable size, and makes this easier to find
- The very top-level item in the Solution Explorer is the solution, which is simply a collection of related projects
- Anything that is part of a class is called a member of the class, and these members include:
    - Variables (the data)
    - Methods (things it can do)
    - Constructors (ways to create or setup new instances of the class)

Adding Instance Variables

- Since each object or instance of the class will have its own set of data, these variables are called instances variables
    - If you have two different books, and each one is represented by its own Book object, they can each keep track of their own titles separately, each in its own set of instance variables

Access Modifiers: private and public

- Everything in a class has an accessibility level, which indicates where it can be accessed from
    - The private keyword indicates that this variable (or method/constructor) can only be accessed (used or changed) from inside of the class
    - The public keyword (or access modifier) allows that variable (or method/constructor) to be accessible from anywhere, including outside of the class
- By making this private, we keep them safe from outside influence, which is almost always what we want
    - If there is a need for things outside of the class to access or modify the data within a class, we can provide methods that are public which will ensure that the data's integrity is maintained
    - This idea is called encapsulation
- The default accessibility level is private for members within a class

Adding Constructors

- Constructors allow us to set the instance variables at the time that a new instance of the object is created, to ensure that the new instance is in a valid state
    - For instance, it probably doesn't make sense to create a book without a title, so we would want a constructor that takes the title as a parameter

```
public Book(string title)
{
    this.title = title;
}
```

- Using the name of the class, and providing no return type tells the compiler that this is a constructor, and not a normal method
- The default constructor will be used if no other constructor is defined within the class

Variable Scope

- If a variable is created within a method, for instance, the variable has method scope
    - A variable that is created inside of a loop could have block scope, while a variable created inside of a class will have class scope
- The key to scope is to let the curly braces be your guide, as typically anything created inside of a set of curly braces can be used anywhere inside of those curly braces

Name Hiding

- One interesting thing that you can do in C# is name a parameter or local variable (which has method or block scope) the exact same thing as something that has class scope
    - This is called name hiding, which can help to avoid the situation where two variables mean the same thing but have different names
- Often, people will use the m_ (which means 'member') convention for naming the instance variables, which avoids name hiding
    - While common, this approach kills readability since you have to mentally remove the m_ to know what the variable does
- Probably the best way to deal with this situation is to use name hiding, along with this keyword to access the instance variables, as the keyword references the current object that you are inside of
    - Note that in many cases, this isn't needed, because C# already assumes you are referring to the stuff inside of the class if there's not anything by that name in the method scope or block scope

```
public Book(string title)
{
    this.title = title;
}
```

Adding Methods

- Methods defined inside of a class can be either public or private, depending on how restrictive you want to be with it
- Methods that both get and set the instance variables are extremely common

- The set methods allow for validation to take place on data

The 'static' Keyword

- Any class member that is marked with the static keyword belongs to the class, rather than any instance
  - Anything that is static doesn't belong to a specific object, but rather it is shared among all the objects of that class
- Static Variables will have the same value shared amongst all the instances
  - If the static class variable is changed in one instance, then that variable's value is changed in all class instances
- Static Methods do not need an instance of a class to be called, since they belong to the class as a whole
  - A method marked with the static keyword is accessed by using the class name instead of an instance name (ex. using the static WriteLine method within the Console object)
- Static Classes cannot be instantiated, and the class cannot have instance variables or methods
  - For instance, the static Console class cannot be instantiated, but there's also no need to do so, since all the methods are static
  - This is great for classes that are used simply to group together a collection of utility methods
- Static Constructors allow you to set up any static class variables that you may have, so that they're ready to go whenever anyone wants to use them
  - As soon as a program first attempts to use your class, the CLR will stop for a second and check for any static constructors that it may have, and if one exists, it will run it

```
public class Something
{
    public static int sharedNumber;

    static Something()
    {
        sharedNumber = 3;
    }
}
```

The 'internal' Access Modifier

- If something is marked as internal, this means that it can be used anywhere inside of the current project or assembly
  - While there are some exceptions to this, you can generally think of an assembly as a single project in compiled form, resulting in a single EXE or DLL
  - Note that classes are internal by default, and will restrict access to even variables and methods that are public
- In contrast, things that are public are available to anyone who has access to the assembly, even if it lives in a different assembly or project
  - Public is sometimes too accessible, since anything that is public to the outside world will have to be maintained and supported, and any changes will directly affect the people using them
- It is good advice to limit the things you want to make public to as few things as possible

Final Thoughts on Classes

- No matter how much you know and how experienced you are, you'll get it wrong sometimes
  - This is true especially as you develop a program and try to add features that you had never thought of, but fortunately, software can easily be refactored and rearranged from the wrong organization to the right organization
- There's no such thing as getting it right all the time, so don't beat yourself up when you do
  - Just change it to make it work like you think it should and keep moving forward

Try It Out!

Classes Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Classes are reference types.
2. (Classes/Objects) define what a particular type of thing can do and store, while a/an (class/object) is a specific instance that contains its own set of data.
3. Name three types of members that can be a part of a class.
4. **True/False.** If something is static, it is shared by all instances of a particular type.
5. What is a special type of method that sets up a new instance is called?
6. Where can something **private** be accessed from?
7. Where can something **public** be accessed from?
8. Where can something **internal** be accessed from?

Answers: (1) True. **(2)** Classes, object. **(3)** Instance variables, methods, constructors. **(4)** True. **(5)** Constructor. **(6)** Only within the class. **(7)** Anywhere. **(8)** Anywhere inside of the project it is contained in.

Chapter 19 – Properties

The Motivation for Properties

- C# has a feature to clean up the getting and setting of instance variables, in the form of properties

Creating Properties

- A property is simply an easy way to create get and set methods, while still maintaining a high level of readability
    - Rather than using parameters as in a normal method, the keyword value is used to take on the value that we are trying to set

```
public int Score
{
    get
    {
        return score;
    }
    set
    {
        score = value;
        if (score < 0)
            score = 0;
    }
}
```

- Properties allow us to get and set the value of instance variables, including doing the extra work to validate what the user is trying to set, without making our code harder to read
    - The instance variable that we don't see behind the property is called the backing field of the property, although it's worth noting that not all properties are required to have a backing field
    - Properties do not need to necessarily have both the get and set blocks for a property, and can have one or the other
- Using lower case names for instance variables, and upper-case names for the matching property is very common

Different Accessibility Levels

- It is possible to have different accessibility levels for the getter and setter

```
public int Score
{
    get  // This is public by default, because the property is marked 'public'.
    {
        return score;
    }
    private set // This, however, is now private.
    {
```

Auto-Implemented Properties

- An even more readable way of using getting and setters is to create auto-implemented properties
    - This creates a backing field behind the scenes, and simple get and set code

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

- To make a property read-only, you should only have a getter with no setter
    - Read-only data is called immutable
- For an auto-implemented property, you can also assign a default value to the property which will get assigned before the constructor runs

```
public class Vector2
{
    public double X { get; set; } = 0;
    public double Y { get; set; } = 0;
}
```

Object Initializer Syntax

- When you create a new object, you can use the following syntax to assign values to the properties, right when you create the object

```
Bookbook = new Book() { Title = "Frankenstein", Author = "Mary Shelley" };
```

  - This prevents us from needing to do this:

```
Book book = new Book();
book.Title = "Frankenstein";
book.Author = "Mary Shelley";
```

- This setup is called object initializer syntax, and can be a convenient way to set up objects without needing to define tons of different constructors

Chapter 20 – Structs

Creating a Struct

- Creating a struct is very similar to creating a class, where the only difference is the use of the struct keyword

```
struct TimeStruct
{
    private int seconds;

    public int Seconds
    {
        get { return seconds; }
        set { seconds = value; }
    }

    public int CalculateMinutes()
    {
        return seconds / 60;
    }
}

class TimeClass
{
    private int seconds;

    public int Seconds
    {
        get { return seconds; }
        set { seconds = value; }
    }

    public int CalculateMinutes()
    {
        return seconds / 60;
    }
}
```

Structs vs. Classes

- The difference between a class and a struct is that structs are value types, where classes are reference types
  - When you assign the value of a struct from one variable to another, the entire struct is copied; this same thing applies to passing one to a method as a parameter
- Note that structs can't be assigned a value of null, and they could potentially slow down your program if you are passing them around or copying them a lot
- With classes, you can create your own parameterless constructor, allowing you to replace the default one with your own custom logic, which cannot be done with structs

Deciding Between a Struct and a Class

- Having a need for reference or value semantics plays an important role in choosing to use a class or a struct
  - As well, if your type is not much more than a compound collection of a small handful of primitives (built-in types), a struct may be the way to go
  - If your going to have a lot of methods (or events/delegates), or want to use inheritance, then a class is a better option

Prefer Immutable Value Types

- In programming, we often talk about types that are immutable, which means that once you've set them up, you can no longer modify them (read-only)
    - Making a value type immutable will save you a great deal of trouble in the long run, since it is far too easy to think that the value you have, which might be a copy, could be used to modify the original

The Built-In Types are Aliases

- The built-in types are not only value types, but they are also struct types

Chapter 21 – Inheritance

- Inheritance allows us to create a Polygon class (for instance) and a Square class that is based on the Polygon
    - This makes it inherit (or reuse) everything from the Polygon class that the Square class is based on, and any changes to the Polygon class carry through automatically to the Square class
- Classes support inheritance, but structs and other types do not

Base Classes

- A base class (or superclass or parent class) is any normal class that happens to be used by another for inheritance

Derived Classes

- A derived class (or subclass) is one that is based on, and expands upon another class
    - In our example above, the Square class is derived from the Polygon class
- The class is created in the same way as always, with the addition of a colon (':') and the name of the base class
    - You can have as many layers of inheritance as you want

```
class Square : Polygon  // Inheritance!
{
    //...
}
```

Using Derived Classes

- Derived classes are just like any other class, and can be used as a type just like any other class
    - But, because the computer knows that a Square is just a special type of Polygon, you can use the Square type anywhere that you could use a Polygon type, since a Square is a Polygon

```
Polygon polygon = new Square(4.5f);
```

    - Because our variable uses the Polygon type, we can only work with the things that the Polygon type defines
- The is keyword or casting allows us to determine if an object is a specific type
- Objects can be converted from one object type to another by using the as keyword

```
Polygon polygon = new Square(4);
Square square = polygon as Square;
```

- As well, you can create an array of the base class and put any derived class inside of it
    - Of general note, a derived class can be used anywhere where a base class can be used

Constructors and Inheritance

- While properties, instance variables and methods are inherited, constructors are not
    - You can't use the constructors in the base class to create a derived class, since a constructor in the base class doesn't have any clue how to set up the new parts of the derived class
- By default, a constructor in the derived class will use the base class's parameterless constructor

The 'protected' Access Modifier

- If a member of the class uses the protected accessibility level, then anything inside of the class can use it, as well as any derived class
    - It's a little broader than private, but still more restrictive than public

The Base Class of Everything: object

- Without specifying any class to inherit from, any class you make still is derived from a special base class that is the base class of everything; the object class
    - If you go up the inheritance hierarchy, everything always gets back to the object class eventually
- Since the object type is the base class for all classes, any and every type of object can be stored in it

```
object anyOldObject = new Square(4.5f);
```

Sealed Classes

- In order to ensure that no one derives anything from a specific class, the class may be sealed to prevent inheritance

Partial Classes

- If class become large, and cannot be broken down into smaller pieces, they can be split into multiple files or sections by using the partial keyword
    - Even if the parts are in different files, they'll be combined into one at compilation
    - The partial keyword can be used on structs and interfaces

C# Does Not Support Multiple Inheritance

- C# does not allow for multiple inheritance, and only one base class can be used
    - This does not prevent a class from being both a derived and base class, however

Try It Out!

Inheritance Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** A derived class adds functionality on to a base class.
2. **True/False.** Everything that is in a base class can be accessed by the derived class.
3. **True/False.** Structs support inheritance.
4. **True/False.** All classes are derived from the **object** class.
5. **True/False.** A class can be a derived class and a base class at the same time.
6. **True/False.** You can prevent a class can't be derived from (can't be used as a base class).

Answers: **(1)** True. **(2)** False. Not private members. **(3)** False. **(4)** True. **(5)** True. **(6)** True.

Chapter 22 – Polymorphism, Virtual Methods, and Abstract Classes

Polymorphism

- Polymorphism is a feature that allows derived classes to override or change the way the base class does things, and do them in their own way
    - The fact that classes can all do the same task (by calling the same method) differently is what gives us "many forms" (Greek translation) and polymorphism
    - Where inheritance allows us to add things to a type, polymorphism allows us to alter how the derived class will handle the things the base class supplies
- The virtual keyword gives derived classes permission to change the way that the method works
    - This can be used for properties, as well as indexers and events

```
class GoogleSearch : SearchEngine
{
    public override string[] Search(string findThis)
    {
        // Google Search is, of course, way better than 3 hard-coded results like this…
        return new string[] {
            "Here are some great results.",
            "Aren't they neat?",
            "I found 1.2 billion things, but you will only look at the first 10." };
    }
}
```

- To change the implementation for the derived class, we simply create a new copy of the method, along with our custom-made implementation, and add the override keyword
    - The virtual keyword is the base class's way of saying "I give you permission to do something different with this method," while the override keyword is the derived class's way of saying "You gave me permission, so I'm going to use it and take things in a different direction"
- For example, because our derived classes RBsSearchEngine and GoogleSearch are inherited from the SearchEngine class, we can use them anywhere a SearchEngine object is required

```
SearchEngine searchEngine1 = new SearchEngine(); // The plain old original one.
SearchEngine searchEngine2 = new GoogleSearch();
SearchEngine searchEngine3 = new RBsSearchEngine();
```

    - Now when we use the Search method, and depending on the actual type, the various implementations will be called

```
string[] defaultResults = searchEngine1.Search("hello");
string[] googleResults = searchEngine2.Search("hello");
string[] rbsResults = searchEngine3.Search("hello");
```

    - We get different results for each of these methods, because each of these three types define the method differently

Revisiting the 'base' Keyword

- We can use the base keyword to access non-private things from the base class, including its methods and properties
    - While most of the time, you can directly access things in the base class without actually needing to use the base keyword, when you override a method, you'll be able to use the base keyword to still get access to the original implementation of the method

```
public override string[] Search(string findThis)
{
    // Calls the version in the base class.
    string[] originalResults = base.Search(findThis);

    if(originalResults.Length == 0)
        Console.WriteLine("There are no results.");

    return originalResults;
}
```

Abstract Base Classes

- With the abstract keyword, we can define classes and methods without providing any implementation or body

```
public abstract class SearchEngine
{
    public abstract string[] Search(string findThis);
}
```

    - Note that abstract methods cannot go inside of regular, non-abstract classes
- When a class is marked as abstract, you can't create any instances of that class directly
    - Instead, you need to create an instance of a derived class that isn't abstract
- Abstract classes can have as many virtual or normal methods as you want
    - In a derived class, the override keyword is used in the exact same way as with virtual methods

The 'new' Keyword with Methods

- When you use the new keyword, you are saying, "I'm making a brand-new method in the derived class that has absolutely nothing to do with any methods by the same name in the base class"
    - This is usually a bad idea because it means that you now have two unrelated methods with the same name and signature
    - This name conflict can be resolved, but it is best to avoid this altogether

- People often get confused and use the new keyword where the override keyword should have been used instead
    - It's best to just go with a different name for a method instead

Try It Out!

Polymorphism Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Polymorphism allows derived classes to provide different implementations of the same method.
2. **True/False.** The **override** keyword is used to indicate that a method in a derived class is providing its own implementation of a method.
3. **True/False.** The **new** keyword is used to indicate that a method in a derived class is providing its own implementation of a method.
4. **True/False.** Abstract methods can be used in a normal (non-abstract) class.
5. **True/False.** Normal (non-abstract) methods can be used in an abstract class.
6. **True/False.** Derived classes can override methods that were **virtual** in the base class.
7. **True/False.** Derived classes can override methods that were **abstract** in the base class.
8. **True/False.** In a derived class, you can override a method that was neither **virtual** nor **abstract** in the base class.

Answers: (1) True. **(2)**True. **(3)** False. **(4)** False. **(5)** True. **(6)** True. **(7)** True. **(8)** False.

Chapter 23 – Interfaces

What is an Interface?

- At a conceptual level, an interface is basically just the boundary it shares with the outside world
    - Interface are everywhere in the real world, and out to users of the object how it can be used or controlled, and also provides feedback on its current status
- Interfaces (including C# implementations) provide these key principles:
    - Interfaces define how outside users make requests to the system
    - Interfaces define the ways in which feedback is presented to the user
- For example, an interface defines how other objects can interact with the car, without prescribing the details of how it works behind the scenes
    - When something presents a interface to the user, the user assumes that it's capable of doing the things that the interface promises
- The broader lesson is that interfaces define a specific set of functionalities, and as long as the interface remains intact, the details on the inside could be swapped out and the user wouldn't have to change anything

Creating an Interface

- Interfaces are structures in a way that looks very much like a class or struct, but instead of the class or struct keyword, you will use the interface keyword

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

namespace AwesomeProgram
{
    public interface IFileWriter
    {
        string Extension { get; }

        void Write(string filename);
    }
}
```

- All the members of an interface have no actual implementation
    - All members of an interface are public and virtual by default, and cannot be altered into anything else
    - You cannot add a method body, nor can you use a different accessibility level
- Interfaces are commonly named by using the letter 'I' as a prefix, as it is easy to pick out an interface versus a class

Using Interfaces

- To make a class use or implement an interface, we use the same notation that we used for deriving a class from a base class (':')

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

namespace AwesomeProgram
{
    public class TextFileWriter : IFileWriter
    {
        public string Extension
        {
            get { return ".txt"; }
        }

        public void Write(string filename)
        {
            // Do your file writing here...
        }
    }
}
```

- If a class implements an interface, it is required to include an implementation for all the members that the interface specifies
    - The class can also include a lot more than that, but at a minimum, it will need to implement the things in the interface
- In most ways, an interface will work in the same way that a base class does
    - For instance, we can have an array of an IFileWriter interface, and put different types of objects that implement the IFileWriter interface in it

```
IFileWriter[] fileWriters = new IFileWriter[3];
fileWriters[0] = new TextFileWriter();
fileWriters[1] = new RtfFileWriter();
fileWriters[2] = new DocxFileWriter();

foreach(IFileWriter fileWriter in fileWriters)
{
    fileWriter.Write("path/to/file" + fileWriter.Extension);
}
```

Multiple Interfaces and Inheritance

- While C# does not allow inheritance of more than one base class, you can implement more than one interface
    - Having your type implement multiple interfaces just means that your type will need to include all the methods for all of the interfaces that you are tying to implement
- In addition to implementing multiple interfaces, you can also still derive from a base class (and only one)
    - While multiple inheritance could be useful in some cases, it can usually be taken care of by either implementing multiple interfaces, or by using a technique called composition, where one class or object simply has another object inside of it (as an instance variable)

```
public class AnyOldClass : RandomBaseClass, IInterface1, IInterface2
{
    // ...
}
```

Try It Out!

Interfaces Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. What keyword is used to define an interface?
2. What accessibility level are members of an interface?
3. **True/False.** A class that implements an interface does not have to provide an implementation for all of the members of the interface.
4. **True/False.** A class that implements an interface is allowed to have other members that aren't defined in the interface.
5. **True/False.** A class can have more than one base class.
6. **True/False.** A class can implement more than one interface.

Answers: (1) interface. **(2)** Public. **(3)** False. **(4)** True. **(5)** False. **(6)** True.

Chapter 24 – Using Generics

The Motivation for Generics

- To explain why we need generics, let's say you wanted to make a class to store a list of numbers
    - An array could do this, but perhaps we make a class that wraps an array and creates a new larger one when we run out of space

    - If we make a class that wraps an array, we can only specify one type (ex. int) that can be used, and would have to make an entirely new class to handle strings
- We could potentially use the object type for our list class, but then we would have to be casting all the time, and could never be certain that our array is type safe
    - Type safety is where you always know what type of object you are working with
- So, we've got two bad choices, where we either make many classes of lists, or we make a single class of objects that aren't type safe
    - This is where generics come into play

What are Generics?

- Generics are a clever way for you to define special generic types (classes or structs) which save a spot to place a specific type in later
    - You can choose what type you want to use in that specific class instance
    - So, one time, you'll use the generic class and say, "this time it is a list of ints," and another time you'll say, "Now I want a list of Hamburger objects"
- In short, generics provide a way to create type safe classes, without having to commit to any type when you create the class

The List Class

- To create an instance of the List class, you need to specify the type of stuff you're putting into the list, which makes it type safe
    - To do this, you put the type you want to use inside of angel brackets ('<' and '>')

```
List<string> listOfStrings = new List<string>();
```

- The List class contains numerous methods to manipulate the items and contents of the list
- Just like arrays, we can use the collection initializer syntax when creating a list

```
List<int> someNumbersInAList = new List<int>() { 14, 24, 37 };
```

The IEnumerable<T> Interface

- The IEnumerable<t> interface is encountered all the time, and allows a collection to give others the ability to look at each item contained in it, one at a time
    - Nearly every class in the .NET framework that contains multiple items implements this interface
    - In this sense, it serves as the base or lowest level of defining a collection
        - Both the List and Dictionary classes implement this interface
- All sorts of things implement IEnumerable<T>, and if all you care about is the ability to do something with each item in a collection, you can get away with treating it simply as an IEnumerable<t>

```
IEnumerable<int> numbers = new int[3] { 1, 2, 3 };
```

    - You'll see some methods returning IEnumerable<T> if they don't want you to know the actual concrete type being used (it could be an array or a List or something else) and if they want you to have the ability to loop through the items (not add or remove items)

The Dictionary Class

- The Dictionary class uses one piece of information (the key) to store and look up another piece of information (the value)
    - It is a mapping of key/value pairs, and because the class is generic, we're not limited to strings, and can use any type that we want
- The Dictionary class is generic, just like the List class, but it has two different types that are generic: the key and the value

```
Dictionary<string, int> phoneBook = new Dictionary<string, int>();
```

    - The Dictionary class allows us to use the indexing operator ('[' and ']') to get or set values in it

```
phoneBook["Gates, Bill"] = 5550100;
phoneBook["Zuckerberg, Mark"] = 5551438;

int billsNumber = phoneBook["Gates, Bill"];
```

Chapter 25 – Making Generic Types

Creating Your Own Generic Types

- Creating a generic type is very similar to creating plain old classes or stucts
    - Structs and classes can both use generics
- Generics are specified by using <T>, which is called a generic type parameter
    - We can use this type parameter anywhere we want to throughout our class to represent the type that will eventually replace it
- We could put multiple types in angle brackets, separated by commas, to have multiple generic types in our class
    - For example, we could use <K, V> for a key/value pair, just like the Dictionary class has
- Note that the 'T' name does not have to be used, it is just a very common name for a generic type

Using Your Generic Type in Your Class

- An example of a completed class that uses generics would look like:

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

namespace Generics
{
    public class PracticeList<T>
    {
        private T[] items;

        public PracticeList()
        {
            items = new T[0];
        }

        public T GetItem(int index)
        {
            return items[index];
        }

        public void Add(T newItem)
        {
            T[] newItems = new T[items.Length + 1];

            for (int index = 0; index < items.Length; index++)
                newItems[index] = items[index];

            newItems[newItems.Length - 1] = newItem;

            items = newItems;
        }
    }
}
```

Constraints for Type Parameters

- For any generic type parameter, you can place constraints or limitations on what types you can actually use for it
    - For instance, you can state that a generic type must be derived from a particular base class, or that it must implement a particular interface
    - If the C# compiler can be sure that the generic type will be derived from a particular base class, it knows that you can call the methods that belong to that base class
- In short, while constraints limit the types that you can use, it gives you a lot more power in what you can actually do with the type parameter
- To specify a constraint, you use the where keyword and the colon (':') operator

```
public class PracticeList<T> where T : IComparable
{
   //...
}
```

    - Multiple constraints for a single type can be added by separating them with commas, and if you have multiple generic types, you can specify type constraints for each of them using a new where keyword

```
public class PracticeDictionary<K, V> where K : SomeRandomInterface, SomeBaseClass
                                       where V : SomeOtherInterface
{
   //...
}
```

- You can do things like specify a parameterless constructor constraint, which requires that the type used has a public parameterless constructor, by using new() as a constraint
    - We could also specify that a type must be a value or reference type with the struct or class constraint
    - Or we could indicate that one type parameter must be derived from another type parameter

Generic Methods

- In addition to generic classes, you can have individual methods that use generics as well
    - These can be a part of any class or struct, generic or regular

```
public T LoadObject<T>(string fileName) { ... }
```

Try It Out!

Generics Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. Describe the problem generics address.
2. How would you create a list of **string**s, using the generic **List** class?
3. How do you indicate that a class has a generic type parameter?
4. **True/False.** Generic classes can only have one generic type parameter.
5. **True/False.** Generic type constraints limit what can be used for the generic type.
6. **True/False.** Constraints let you use the methods of the thing you are constraining to.

Answers: (1) Type safety without creating a lot of types that differ only by the types they use. **(2)** List<string> strings = new List<string>(); **(3)** In angle brackets by the class name. **(4)** False. **(5)** True. **(6)** True.

Chapter 26 – Namespaces and Using Directives

Namespaces

- A namespace is simply a grouping of names
    - Usually, people will put related code in the same namespace, so you can also think of a namespace as a module or package for related code
    - Namespaces separate one type with a certain name from other types with the same name

Fully Qualified Names

- When combined, the namespace name and the class name is called a fully qualified name
    - System.Drawing.Point is a fully qualified name, as is System.Windows.Point

Using Directives

- Using fully qualified names typically makes your code less readable
    - In any file, we have the ability to point out a namespace and say, "I'm using the stuff in that namespace, so if I don't use a fully qualified name, that's where you'll find it"
- This is done by putting in a using directive at the top of the file

The Error 'The type or namespace X could not be found'

- If you try to use the unqualified name for a type, leaving out the namespace, and you don't have an appropriate using directive, you'll run into an error

```
The type or namespace 'X' could not be found (are you missing a using directive or an assembly
reference?)
```

    - This could mean that either a using directive is missing, or an assembly reference is missing, but it's usually the first
- When this happens, you can hover the mouse over the underlined word, and have the namespace added automatically to your file

Name Collisions

- On a few rare occasions, you'll add using directives for two namespaces that both contain a type with the same name
    - This is called a name collision because the unqualified name is ambiguous

- One solution is to use a fully qualified name, which is the author's preferred solution, as it keeps things clear and unambiguous
    - You can also use an alias to solve the program, which can be thought of as a nickname
- An alias can be defined by using the using keyword, through which a namespace can be assigned the alias
    - The drawback to this approach is that you now have an additional name floating around that doesn't exist anywhere

```
using CFromN1 = N1.C;
```

Static Using Directives

- When working with static classes, you can use a special type of using directive called a using static directive
    - Recall that a static class is one that is marked with the static keyword, which forces all members to be static
- For these static classes, you can add a using directive with the static keyword, and you'll have access to its methods without even needs to name the class it belongs to

```
using static System.Math;
using static System.Console;

namespace StaticUsingDirectives
{
    public static class Program
    {
        public static void Main(string[] args)
        {
            double x = PI;
            WriteLine(Sin(x));
            ReadKey();
        }
    }
}
```

- This can be a bit of a double-edged sword, since it makes the code more concise, but strips out meaningful information about which type each static method, property, or variable came from
    - Use it when the extra simplicity makes things clearer, and avoid it when it makes it more ambiguous

Try It Out!

Namespaces Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False. using** directives make previously inaccessible code accessible.
2. **True/False. using** directives make it so you do not need to use fully qualified names.
3. **True/False.** Two types are allowed to have the same name.
4. **True/False.** A name collision is when two types have the same name.
5. Name two ways to resolve a name collision.
6. What **using** directive would need to be added so that you can use **System.Math**'s **Abs** (absolute value) method as **Abs(x)** instead of **Math.Abs(x)**?

Answers: (1) False. **(2)** True. **(3)** True. **(4)** True. **(5)** Use fully qualified names or aliases. **(6)** using static System.Math;

Chapter 27 – Methods Revisited

Optional Parameters

- C# provides a way to specify a default value for a parameter, making it optional if you're ok with the default

```
public int RollDie(int sides = 6)
{
    return random.Next(sides) + 1;
}
```

    - With this code, you can now call RollDie() and the value of 6 will be used, or you can fall back to putting a value in if you would like

```
RollDie();   // Uses the default value of 6.
RollDie(20); // Uses 20 instead of the default.
```

- While you're allowed to have as many optional parameters as you want, and you can mix and match them with non-optional parameters, all optional parameters must come at the end, after all the "normal" parameters

Named Parameters

- To avoid the ambiguity of not knowing what order the parameters of a method are in, C# allows you to supply names for parameters

```
Clamp(min: 50, max: 100, value: 20);
```

- With named parameters, it is very clear what value belongs to which parameter
    - Furthermore, as long as the compiler can figure out where everything goes, this allows you to put the parameters in any order, making it so that you don't need to worry about the official ordering, as long as they all have a value
- When you call a method, you can use regular unnamed parameters, or named parameters, but all regular unnamed parameters must come first, and each variable must be assigned only once
    - You can also combine optional parameters with named parameters
- One side effect of this feature is that parameter names are now a part of your program's public interface
    - Changing a parameter's name can break code that calls the method using named parameters, so use these carefully

Variable Number of Parameters

- C# provides a way to supply a variable number of parameters to a method, by using the params keyword with an array variable

```
public static double Average(params int[] numbers)
{
    double total = 0;

    foreach (int number in numbers)
        total += number;

    return total / numbers.Length;
}
```

    - To the outside world, the params keyword makes it look like you can supply any number of parameters, but behind the scenes, the C# compiler will turn these into arrays to use in the method call

```
Average(2, 3);
Average(2, 5, 8);
Average(41, 49, 29, 2, -7, 18);
```

    - You can also directly pass an array into this type of method
- You can combine a params argument with other parameters, but the params argument must be the last one, and there can only be one of them

The 'out' and 'ref' Keywords

- Recall that for value types being pass into a method as a parameter, we get a cop of the original data, while for reference types, we get a copy of the reference to an object in the heap
- Methods have the option of handing off the actual variable, rather than copying its contents, by using the ref or out keyword
    - Doing this means that the called method is working with the exact same variable that existed in the calling method
    - This sort of creates the illusion that value types have been turned into reference types, and sort of takes reference types to the next level, where it feels like you've got a reference to a reference

```
public void MessWithVariables(out int x, ref int y)
{
    x = 3;
    y = 17;
}
```

We can then call this code like this:

```
int banana = 2;
int turkey = 5;

MessWithVariables(out banana, ref turkey);
```

    - At the end, banana will contain a value of 3, while turkey will be 17
- The out and ref keywords will have to be used to call the method as well, since handing off the actual variables from one method to another is a risky game to play
    - By requiring these keywords in both the called method and the calling method, we can ensure that the variables were handed off intentionally
- With the ref keyword, the calling method needs to initialize the variable before the method call, which allows the called method to assume that it is already initialized
    - These are sometimes called reference parameters
- With the out keyword, the compiler ensures that the called method initializes the variable before returning from the method
    - These are sometimes called output parameters

- Since we're passing the actual variable, rather than just the contents, this means that the value type won't have to be completely copied, which can speed things up
    - You can also use this to get back multiple values from a method, but there are better ways to do this (building a class that can store all data of interest, or using the Tuple class)
- Be warned that sending the actual variable to a method, instead of just the contents of the variable is a dangerous thing to do, since we are giving another method control over our variables

Try It Out!

Methods Revisited Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

For questions 1-3, consider the following code: **void DoSomething(int x, int y = 3, int z = 4) { … }**

1. Which parameters are optional?
2. What values do **x**, **y**, and **z** have if called with **DoSomething(1, 2);**
3. What values do **x**, **y**, and **z** have if called with the following: **DoSomething(x : 2, z : 9);**
4. **True/False.** Optional parameters must be after all required parameters.
5. **True/False.** A parameter that has the **params** keyword must be the last parameter.
6. Given the method **void DoSomething(int x, params int[] numbers) { … }** which of the following are allowed?
    a. **DoSomething();**
    b. **DoSomething(1);**
    c. **DoSomething(1, 2, 3, 4, 5);**
    d. **DoSomething(1, new int[] { 2, 3, 4, 5 });**
7. **True/False.** Parameters that are marked with **out** result in handing off the actual variable passed in, instead of just copying the contents of the variable.
8. **True/False.** Parameters that are marked with **ref** result in handing off the actual variable passed in, instead of just copying the contents of the variable.
9. **True/False.** Parameters that are marked with **out** must be initialized inside the method.
10. **True/False.** Parameters that are marked with **ref** must be initialized inside the method.

Answers: (1) y and z. **(2)** x=1,y=2,z=4. **(3)** x=2,y=3,z=9. **(4)** True. **(5)** True. **(6)** b, c, d. **(7)** True. **(8)** True. **(9)**True. **(10)** False.

Chapter 28 – Reading and Writing Files

All at Once

- To write a bunch of text to a file, we can use the File class's static WriteAllText method

```
string informationToWrite = "Hello persistent file storage world!";
File.WriteAllText("C:/Place/Full/Path/Here.txt", informationToWrite);
```

- To read a file back, we would use the ReadAllLines method

```
string fileContents = File.ReadAllText("C:/Place/Full/Path/Here.txt");
string[] fileContentsByLine = File.ReadAllLines("C:/Place/Full/Path/Here2.txt");
```

- If we wanted to read a file back in to reassemble a high score list, for instance, we may start off by reading the entire text of the file, then parse (the process of breaking up a block of text into smaller, more meaningful components) the text and turn it back into our list of high scores

```
Arwen,2778
Gimli,140
Bilbo,129
Aragorn,88
Sam,36
```

```
public class HighScore
{
    public string Name { get; set; }
    public int Score { get; set; }
}
```

```
public HighScore[] LoadHighScores(string fileName)
{
    string[] highScoresText = File.ReadAllLines(fileName);

    HighScore[] highScores = new HighScore[highScoresText.Length];

    for (int index = 0; index < highScoresText.Length; index++)
    {
        string[] tokens = highScoresText[index].Split(',');

        string name = tokens[0];
        int score = Convert.ToInt32(tokens[1]);

        highScores[index] =new HighScore() { Name = name, Score = score };
    }

    return highScores;
}
```

Text-Based Files

- There's another way to handle text-based file I/O that is quite a bit more complicated, but allows us to do the writing or reading a little at a time instead of all at once
- Rather than writing out an entire file all at once, we can instead use the FileStream object, which we'll wrap in a TextWriter object to simplify the process, and use that to write stuff out as needed

```
FileStream fileStream = File.OpenWrite("C:/Place/Full/Path/Here3.txt");
StreamWriter writer = new StreamWriter(fileStream);

writer.Write(3);
writer.Write("Hello");

writer.Close();
```

- Reading the file this way is more troublesome than writing it was, since we don't know the structure ahead of time to allow us to read it back in

```
FileStream fileStream = File.OpenRead("C:/Place/Full/Path/Here3.txt");
StreamReader reader = new StreamReader(fileStream);

// Read a single character at a time.
char nextCharacter = (char)reader.Read();

// Read multiple characters at a time.
char[] bufferToPutStuffIn = new char[2];
reader.Read(bufferToPutStuffIn, 0, 2);
string whatWasReadIn = new string(bufferToPutStuffIn);

// Read a full line at a time.
string restOfLine = reader.ReadLine();

reader.Close();
```

  - When we're done, we call the Close method to make sure that we are no longer connected to the file

Binary Files

- Instead of writing a text-based file, the other choice is to write a binary file, which can't be simply opened and read in a text-editor (not encrypted, though)
  - Note that binary files usually take up less space than text-based files

Chapter 29 – Error Handling and Exceptions

- C# provides a powerful way to trap or catch errors and recover from them gracefully, through an approach called exception handling
  - It is like many other languages such as C++, Java, and VB.NET

How Exception Handling Works

- When an error occurs, the code that discovered the problem will package up all the information it knows about the problem into a special object called an exception
  - These exceptions are either an instance of the Exception class or a class derived from the Exception class

- Since the method that discovered the error does not know how to recover from it, it takes the Exception object and throws it up to the method that called it, hoping that it knows how to resolve the issue
    - Once this happens, the rest of the method will not get executed
- If the exception makes it all the way back to the top of the call stack without being caught, your program will have a fatal error
    - If in release mode, this is called an unhandled exception, whereas in debug mode, Visual Studio will handle the exception at the last minute and allow you to dig around to see what went wrong

Catching Exceptions

- To catch exceptions, we will take a potentially dangerous block of code and place it in a try block, followed by a catch block that identifies the exceptions that we can handle and defines how to resolve them

```
static int GetNumberFromUser()
{
    int usersNumber = 0;

    while(usersNumber < 1 || usersNumber > 10)
    {
        try
        {
            Console.Write("Enter a number between 1 and 10: ");
            string usersResponse = Console.ReadLine();
            usersNumber = Convert.ToInt32(usersResponse);
        }
        catch(Exception e)
        {
            Console.WriteLine("That is not a valid number. Try again.");
        }
    }

    return usersNumber;
}
```

Not Giving the Exception Variable a Name

- While the example above has a name given to the exception (“e”), we don’t have to give a name
    - Without a name, it is impossible to use the exception variable, but it also frees up the name that you would have used for something else, and won’t give you a compiler warning about a variable being declared but not used

Handling Different Exceptions in Different Ways

- In most cases, you’ll want to handle different exception types in different ways

```
try
{
    int number = Convert.ToInt32(userInput);
}
catch (FormatException e)
{
    Console.WriteLine("You must enter a number.");
}
catch (OverflowException e)
{
    Console.WriteLine("Enter a smaller number.");
}
catch (Exception e)
{
    Console.WriteLine("An unknown error occured.");
}
```

    - When you do this, only one of the catch blocks will be executed, and order is important here (specific types first)
- Not all exceptions that come up need to be caught, and you can build your catch blocks in a way that some are handled while others are allowed to propagate further up the call stack, possibly to a catch block in another method

Throwing Exceptions

- You can create and throw exceptions yourself, but don’t get carried away with throwing exceptions
    - As a rule, the close you handle an error to the place it was detected, the better

```
public void CauseTrouble()  // Always up to no good...
{
    throw new Exception("Just doing my job!");
}
```

- Note that you are not limited to using the Exception class (and it's preferable not to), since there are other options already defined (ex. NotImplementedException, IndexOutOfRangeException, etc.)
- If you can't find an existing exception type that fits, you can create your own
    - All you need to do is create a class that is derived from the Exception class or from another exception class

```
// For all of you on the other side of the pond, if it makes you feel any better, you
// can still call them "beef burgers".
public class AteTooManyHamburgersException : Exception
{
    public int HamburgersEaten { get; set; }

    public AteTooManyHamburgersException(int hamburgersEaten)
    {
        HamburgersEaten = hamburgersEaten;
    }
}
```

With this class, you can now say:

```
throw new AteTooManyHamburgersException(125);
```

And:

```
try
{
    EatSomeHamburgers(32);
}
catch(AteTooManyHamburgersException e)
{
    Console.WriteLine($"{e.HamburgersEaten} is too many hamburgers.");
}
```

- As a rule, you should be throwing different exception types whenever the calling code would want to handle the error in different ways
    - If, for example, your method could fail in two different ways, you should be throwing two different exception types, which would allow you to use different catch blocks to handle them differently
    - If there's no good pre-made exception type for you to use, make your own
- It gets easy to get lazy and start always throwing just the plain Exception type
    - Don't get lazy, and use the right exception types, even if that means creating your own

The 'finally' Keyword

- The finally block can be placed after the try or catch block, and the code inside gets executed no matter what
    - Be careful to not use the finally block to handle exceptions

Exception Filters

- The reality is, you don't always want to handle all exceptions of the same type in the exact same way or at the exact same time
    - One option in this case is to catch the exception, check for certain condition and handle the things that you want, and then rethrow anything left over

```
try
{
    throw new Exception("Message");
}
catch(Exception e)
{
    if (e.Message == "Message") { Console.WriteLine("I caught an exception."); }
    else { throw; }
}
```

    - This is called rethrowing the exception, and allows you to throw the exact same exception that was caught, but with the limitation that the stack trace has now moved away from the original place that it was thrown (can be frustrating for debugging)
- In C# 6.0, exception filters provide a better option, since they allow you to add a bit of code to the catch block that can be used to filter whether an exception is caught or not

```
try
{
    throw new Exception("Message");
}
catch(Exception e) if(e.Message == "Message")
{
    Console.WriteLine("I caught an exception.");
}
```

- This approach is preferable to filtering by placing an if-statement inside of the catch block, since it is clearer and avoids the problem where the stack trace is changed

Some Rules about Throwing Exceptions

- You should not leave resources open that were opened before the exception was thrown
  - They should be closed first, which can usually be handled in a finally clause, but we need to ensure that the program is still in a safe state
- You should revert to the way things were before the problem occurred, so that once the error has been handled, people know that the system is still in a valid state
  - A finally block can do this, as well
- If you can avoid throwing an exception in the first place, that is what you should do

Chapter 30 – Delegates

Delegates: Treating Methods like Objects

- Delegates in C# allow us to assign a method to a variable and pass it around

Creating a Delegate

- When we create a delegate, we're going to say, "This type of delegate can store any method that has these parameters, and this return type"
- Delegates are typically created directly inside of a namespace, just like we've done before with classes, structs, and enums, and it is also typical to put them in their own file
  - To create a delegate, we would create a new file with a name that matches our delegate (ex. a delegate called MathDelegate would have a file called MathDelegate.cs)

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

namespace Delegates
{
    public delegate int MathDelegate(int a, int b);
}
```

Using Delegates

- The first thing that we need to use delegates is some methods that match the delegate's requirements (ex. two int parameters, with an int as the return type)

```
public static int Add(int a, int b)
{
    return a + b;
}

public static int Subtract(int a, int b)
{
    return a - b;
}
```

  - Like with a normal variable, we can assign it a value, but in this case, that value will be the name of a method that matches the delegate

```
MathDelegate mathOperation = Add;

int a = 5;
int b = 7;

int result = mathOperation(a, b);
```

  - Instead of being a normal method call, the delegate hands off execution to whatever method happens to be stored in the delegate variable at that time
- Delegates are useful in that they allow you to dynamically change which method is called at runtime, and can be used as part of events, lambda expressions, and query expressions

- This provides a foundation for some powerful features in C#

The Delegate and MulticastDelegate Classes

- When a delegate is defined, the C# compiler will turn it into a class that derives from MulticastDelegate, which derives from the Delegate class
  - Note that you are not allowed to create a class that derives from either of these classes
- Note that you can declare a new delegate type from anywhere that you can declare a new class, and that the delegate can contain null, which should be checked for each time
- The Invoke method can also be used to call a method, which some C# programmers prefer to use

```
int result = mathOperation.Invoke(3, 4);
```

Delegate Chaining

- Delegates can work with multiple methods through the MulticastDelegate class
- A second (or third, etc.) method can be added to a delegate by using the += operator

```
LogEventHandler logHandlers = LogToConsole;
logHandlers += LogToFile;
```

  - Now when we invoke the delegate, both methods in the delegate will be called

```
logHandlers(new LogEvent("Message"));
```

  - Methods can be removed from a delegate by using the -= operator
- Note that when multiple methods are called through a multicast delegate, only the final method to be invoked returns the value, and all other results are completely ignored
  - In practice, multicast delegates are generally only used for signature with a void return type
- As well, if an exception is thrown when a method is invoked, and further handlers down the chain will not be called, so it is critical to avoid throwing exceptions in methods that are attached to delegates

The Action and Func Delegates

- You can make your own delegates in C# whenever you want to, but it is likely that you won't need to create your own delegates very often
  - Included in the .NET Framework is a collection of pre-defined delegates that use generics, and are incredibly flexible
- The Action delegates all have a void return type, while the Func delegates have a generic return type
  - The Action delegate is pre-defined for up to 16 parameters

```
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
public delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2, T3 arg3, T4 arg4);
```

  - The Func delegate is also pre-defined for up to 16 parameters

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, Result>();
public delegate TResult Func<T1, T2, Result>();
```

- Calling an Action or Func delegate is simple

```
Func<int, int, int> mathOperation = Add;
int a = 5;
int b = 7;
int result = mathOperation(a, b);
```

Chapter 31 – Events

- Events allow classes to notify others when something specific happens
  - This is extremely common in GUI-based applications, where there are things like buttons and checkboxes
- Outside of user interfaces, events can also allow "client" or listener classes to attach themselves to specific events generated by other classes when they're interested, and detach themselves when they don't care anymore
  - The advantage of this kind of a model is that the object that is raising events doesn't need to know or care about the details of who is attached or listening to any event, nor is it responsible for getting listeners registered or unregistered

Defining an Event

- To define an event inside of a class, we'll need to add a single line of code as a member of the class

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

namespace Events
{
    public class Point
    {
        private double x;
        private double y;

        public double X
        {
            get { return x; }
            set
            {
                x = value;
                OnPointChanged();
            }
        }

        public double Y
        {
            get { return y; }
            set
            {
                y = value;
                OnPointChanged();
            }
        }

        public event EventHandler PointChanged;

        public void OnPointChanged()
        {
            if(PointChanged != null)
                PointChanged(this, EventArgs.Empty);
        }
    }
}
```

- The event keyword is what makes this member an event that others can attach to
    - This member can be defined as public, internal, protected, or private, but is typically public or internal
- We also specify the delegate type of methods that can be attached to the event
    - Remembering how delegates work, this means that all listener methods will be required to have a specific set of parameters, as well as a specific return type (though with events, it is almost invariably void)
    - In this case, we use the EventHandler delegate, which is a pre-defined delegate specifically made for simple events
        - This delegate has a return type of void and has two parameters; an object, which is the "sender" of the event, and an EventArgs object
        - The EventArgs object is simple, but is designed to store some information about the event itself
- Note that any delegate type can be used, and we are not just limited to EventHandler

Raising an Event

- Now that we have an event, we need to add in code to raise the event in the right circumstances (see sample code above)
    - We check to see if the event is null, which would mean that no event handlers are attached, and would result in a NullReferenceException
- Once we know that the event has event handlers attached to it, we can raise the event by calling the event with the parameters required by the delegate – in this case, a reference to the sender (this) and an EventArgs object
    - Note that it is very common to put the word "On" at the beginning of the method name, which is named the same as the event

Attaching and Detaching Event Handlers

- The way we attach something to an event is giving the event a method to call when the event occurs
    - The method we attach is sometimes call an event handler
    - Because events are based on delegates, we'll need a method that has the same return type and parameter types as the delegate we're using – EventHandler in this case

```
public void HandlePointChanged(object sender, EventArgs eventArgs)
{
    // Do something intelligent when the point changes. Perhaps redraw the GUI,
    // or update another data structure, or anything else you can think of.
}
```

- To attach an event handler to the event, we would use:

```
Point point = new Point();

point.PointChanged += HandlePointChanged;

// Now if we change the point, the PointChanged event will get raised,
// and HandlePointChanged will get called.
point.X = 3;
```

  - The += operator can be used to add the HandlePointChanges method to our event, while the -= operator can be used to remove the method
  - You can attach as many event handlers to an event as you want, and all will be executed, provided that an exception is not thrown along the way

Common Delegate Types Used with Events

- When using events, the event can use any delegate type imaginable, although there are a few that are most common
  - If you want an event that simply calls a parameterless void method when the event occurs (a simple notification without any additional data or information), then the Action delegate can be used

```
public event Action PointChanged;
```

You could subscribe the method below to that event:

```
private void HandlePointChanged()
{
    // Do something in response to the point changing.
}
```

- If you need to get some sort of information to the subscriber, point data for example, you could use one of the other variants of Action that has generic type parameters

```
public event Action<Point> PointChanged;
```

  - Then we subscribe to the event with a method like this:

```
private void HandlePointChanged(Point pointThatChanged)
{
    // Do something in response to the point changing.
}
```

- We could also use the EventHandler<TEventArgs> event handler, which we could use for example to keep track of a change to a number

```
public class NumberChangedEventArgs : EventArgs
{
    public int Original { get; }
    public int New { get; }
    public NumberChangedEventArgs(int originalValue, int newValue)
    {
        Original = originalValue;
        New = newValue;
    }
}
```

  - We'd define our event like this:

```
public event EventHandler<NumberChangedEventArgs> NumberChanged;
```

  - This event would be raised like we saw earlier, usually in a method called OnNumberChanged:

```
public void OnNumberChanged(int oldValue, int newValue)
{
    if(NumberChanged != null)
        NumberChanged(this, new NumberChangedEventArgs(oldValue, newValue));
}
```

  - Methods that want to handle this event would then look like this:

```
public void HandleNumberChanged(object sender, NumberChangedEventArgs args)
{
    Console.WriteLine($"The original value of {args.Original} is now {args.New}.");
}
```

The Relationship between Delegates and Events

- It's a common misconception among C# programmers that events are just delegates that can be attached to multiple methods, or that an event is just an array of delegates, both of which are wrong
    - In most cases, delegates are not used to call more than one method at a time, while events quite often are, and hence the C# programmers assume that delegates can only handle a single method
- A more accurate way to think of it is more along the lines of an event being a property-like wrapper around a delegate, with what we've seen up to now being like an auto-implemented property

Try It Out!

Delegates and Events Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Delegates allow you to assign methods to variables.
2. **True/False.** You can call and run the method currently assigned to a delegate variable.
3. **True/False.** Events allow one object to notify another object when something occurs.
4. **True/False.** Any method can be attached to a specific event.
5. **True/False.** Once attached to an event, a method cannot be detached from an event.

Answers: (1) True. **(2)** True. **(3)** True. **(4)** False. **(5)** False.

Chapter 32 – Operator Overloading

- When you create your own types, you can also define how some of these operators should work for them
    - For example, if you create a class called Cheese, you can also define what the '+' operator should do, allowing you to add two Cheese objects together
    - This is called operator overloading, and it is a powerful feature of C#
- The following operators can be overloaded: +, -, *, /, %, ++, --, ==, !=, >=, <=, >, and <
    - Additionally, the operators +=, -=, /=, and %= are not technically overloadable, but when we overload the + operator, the += operator is automatically overloaded for us, and so on
    - The logical operators && and ||, the assignment operator (=), the dot operator (.), and the new operator cannot be overloaded
- As well, you cannot create entirely new operators using operator overloading

Overloading Operators

- As an example, to overload the '+ operator, we simply add the following code as a member of the Vector class

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

namespace OperatorOverloading
{
    public class Vector
    {
        public double X { get; set; }
        public double Y { get; set; }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }

        public static Vector operator +(Vector v1, Vector v2)
        {
            return new Vector (v1.X + v2.X, v1.Y + v2.Y);
        }
    }
}
```

- All operator overloads must be public and static
    - This should make sense, since we want to have access to the operator throughout the program, and since it belongs to the type, rather than a specific instance of the type
- The relational operators can be overloaded in the exact same way, only they must return a bool

```
public static bool operator ==(Vector v1, Vector v2)
{
    return ((v1.X == v2.X) && (v1.Y == v2.Y));
}

public static bool operator !=(Vector v1, Vector v2)
{
    return !(v1 == v2); // Just return the opposite of the == operator.
}
```

- Recall that relational operators must be overloaded in pairs, as shown in the sample

- Overloading operators is done primarily to make our code look cleaner, and is known as syntactic sugar
  - Anything that you can do with an operator, you could have done with a method
- Just because you can overload operators, does not mean that you should
  - The rule to follow is to only overload operators that have a single, clear, intuitive, and widely accepted use

Try It Out!

Operator Overloading Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Operator overloading means providing a definition for what the built-in operators do for your own types.
2. **True/False.** You can define your own, brand new operator using operator overloading.
3. **True/False.** All operators in C# can be overloaded.
4. **True/False.** Operator overloads must be **public**.
5. **True/False.** Operator overloads must be **static**.

Answers: (1) True. **(2)** False. **(3)** False. **(4)** True. **(5)** True.

Chapter 33 – Indexers

- Indexers are a way to overload the indexing operator ([and]), although just because we can use it doesn't mean that it's always the right solution

How to Make an Indexer

- Defining an indexer is almost like a cross between overloading an operator and a property, and is defined within a class

```
public double this[int index]
{
    get
    {
        if(index == 0) { return X; }
        else if(index == 1) { returnY; }
        else { throw new IndexOutOfRangeException(); }
    }
    set
    {
        if (index == 0) { X = value; }
        else if (index == 1) { Y = value; }
        else { throw new IndexOutOfRangeException(); }
    }
}
```

  - Note that we don't have to use public and static, unlike when overloading other operators
  - Inside the brackets, we list the type and name of the indexing variable that we'll use inside of this indexer
- With this is place, we should be able to use the code below, which is clearer than if we had been forced to do it with methods (xComponent = v.GetIndex(0);)

```
Vector v = new Vector(5, 2);
double xComponent = v[0]; // Use indexing to set the x variable.
double yComponent = v[1]; // Use indexing to set the y variable.
```

Using Other Types as an Index

- We're not stuck with just using integers for an index, and can use any type that we want

```
public double this[string component]
{
    get
    {
        if (component == "x") { return X; }
        if (component == "y") { return Y; }
        throw new IndexOutOfRangeException();
    }

    set
    {
        if (component == "x") { X = value; }
        if (component == "y") { Y = value; }
        throw new IndexOutOfRangeException();
    }
}
```

- This allows us to get and set like this

```
Vector v = new Vector(5, 2);
double xComponent = v["x"]; // Indexing operator with strings.
double yComponent = v["y"];
```

- Note that we can do indexing with multiple indices, such as using both a string and integer

- Indexers can be very powerful, and allows us to make indexing or data access look more natural when working with our own custom-made types
  - Like operator overloading, we should take advantage of this when it makes sense, but be cautious of overusing it

Index Initializer Syntax

- If a class defines an indexer, you can use index initializer syntax (or an index initializer)

```
Dictionary dictionary = new Dictionary()
{
    ["apple"] = "A particularly delicious pomaceous fruit of the genus Malus.",
    ["broccoli"] = "The 7th most flavorless vegetable on the planet."
    // ...
};
```

- Because index initializer syntax can be used any time the type defines an indexer, and because lots of classes define indexers, there are quite a few places that this can be used
  - You should let the readability of the code dictate whether it's the right choice to use or not

Chapter 34 – Extension Methods

- When using a class that has been made by somebody else (the string type, for example), you can extend the class' functionality by adding in your own methods (ex. myString.ToRandomCase(), which doesn't exist in the .NET Framework)
  - These are called extension methods, which are static methods in a static class that makes a method appear as though it were a member of the original class

Creating an Extension Method

- To create an extension method, start by adding a new class file to your project, which can be called something like StringExtensions, or PointExtensions, for example
  - The first parameter must be the type of object that we're creating the extension method for, marked with 'this' keyword

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

namespace ExtensionMethods
{
    public static class StringExtensions
    {
        private static Random random = new Random();

        public static string ToRandomCase(this string text)
        {
            string result = "";

            for (int index = 0; index < text.Length; index++)
            {
                if (random.Next(2) == 0)
                    result += text.Substring(index, 1).ToUpper();
                else
                    result += text.Substring(index, 1).ToLower();
            }

            return result;
        }
    }
}
```

- Like with operator overloading and indexers, this is basically just syntactic sugar, as the C# compiler will rework any place that we call our extension method

```
string title = "Hello World!"
string randomCaseTitle = title.ToRandomCase();
```

  - The extension method looks nicer and it feels like a real method of the string class, which is a nice thing, but be aware of the limitations of using an extension method (need additional using statements, can't assume that methods will work from program to program, etc.)

Chapter 35 – Lambda Expressions

The Motivation for Lambda Expressions

- Rather than creating new methods or using delegates to filter out values from a list, we could use a lambda expression instead
  - This could reduce clutter in your code, and reduce the separation of the method from the calling code, hence making the code more readable
- Note that lambda expressions more or less replace anonymous methods (C# 2.0)

Lambda Expressions

- Basically, a lambda expression is simply an anonymous method that is written in a form that theoretically makes it more readable (C# 3.0)
- As an example, if we wanted to create a lambda expression to determine if a variable is even or odd, we would write the following

```
x => x % 2 == 0
```

  - The lambda operator (=>) is read as "goes to" or "arrow"
  - The type of the value on the right becomes the return type of the expression, where this expression takes the input value (X), and mods it with '2', checking the result against '0'

Multiple and Zero Parameters

- Lambda expressions can have more than one parameter

```
(x, y) => x * x + y * y
```

  - This could have been written in method form, as shown below

```
public int HypoteneuseSquared(int x, int y)
{
    return x * x + y * y;
}
```

- Along the same lines, you can also have a lambda expression that has no parameters

```
() => Console.WriteLine("Hello World!")
```

Type Inference and Explicit Types

- The C# compiler is smart enough to look at most lambda expressions and figure out what variable types and return type you are working with
    - This is called type inference
- If the compiler can't figure it out, an error message will be displayed, and you may have to explicitly write out the variable type

```
(int x) => x % 2 == 0;
```

Statement Lambdas

- In addition to the expression lambda, we can also use the statement lambda, which allows more complicated statements that include a return statement

```
(int x) => { bool isEven = x % 2 == 0; return isEven; }
```

    - As a statement lambda gets longer, you should probably consider pulling it out into its own method

Scope in Lambda Expressions

- Inside of a lambda expression, you can access the variables that were in scope at the location of the lambda expression

```
int cutoffPoint = 5;
List<int> numbers = new List<int>(){ 1, 7, 4, 2, 5, 3, 9, 8, 6 };

IEnumerable<int> numbersLessThanCutoff = numbers.Where(x => x < cutoffPoint);
```

    - If our lambda expression had been turned into a method, we wouldn't have access to that cutoffPoint variable (unless we supplied it as a parameter)

Expression-Bodied Members

- C# allows you to use the same expression syntax to define normal non-lambda methods within a class

```
public int ComputeSquare(int value) => value * value;
```

    - This only works if the method can be turned into a single expression, meaning that we can't use the statement lambda syntax
    - If we need a statement lambda, we would just write a normal method
- This syntax is not limited to methods, and can apply to indexers, operator overloads, and properties (but not for the setter)

```
public class SomeSortOfClass
{
    // These two private instance variables used by the methods below.
    private int x;
    private int[] internalNumbers = new int[] { 1, 2, 3 };

    // Property (read-only, no setter allowed)
    public int X => x;

    // Operator overload
    public static int operator +(SomeSortOfClass a, SomeSortOfClass b) => a.X + b.X;

    // Indexer
    public int this[int index] => internalNumbers[index];

    // Normal method
    public int ComputeSquare(int value) => value * value;
}
```

- It is important to reiterate that just because we can use lambda statements does not mean that we necessarily should, and it always comes back to readability and understandability

Try It Out!

Lambda Expressions Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** Lambda expressions are a special type of method.
2. **True/False.** A lambda expression can be given a name.
3. What operator is used in lambda expressions?
4. Convert the following to a lambda expression: **bool IsNegative(int x) { return x < 0; }**
5. **True/False.** Lambda expressions can only have one parameter.
6. **True/False.** Lambda expressions have access to the local variables in the method they appear in.

Answers: (1) True. **(2)** False. **(3)** Lambda operator (=>). **(4)** x => x < 0. **(5)** False. **(6)** True.

Chapter 36 – Query Expressions

- Query language constructs in C# are allowed in the form of query expressions

What is a Query Expression?

- One of the most common tasks that you will do in programming is to dig through data sets
    - A query is simply a way of choosing a specific chunk of the complete set of data, including how the results should be organized
- Most things that we've done up until now have been procedural, meaning that we've said step-by-step how to do it
    - Query expressions are declarative, which means that we don't care how it is done, but rather declare what we want and allow the computer to figure it out
    - Note that anything that we can do with query expressions can be done with procedural stuff like loops and if statements, but query expressions promote readability
- Query expressions are excellent for working with SQL databases, as well

LINQ and Query Expressions

- Query expressions are done using Language Integrated Query (LINQ)

Creating a Simple Query Expression

- The purpose of a query expression is to take a set of data and retrieve a subset of that data in a new collection, with the possibility of retrieving the results in a specific order or grouping
- As an example, if we have a Person class, and a List of allPeople, we can query it as shown below

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public int Height { get; set; }
    public int Weight { get; set; }
}
```

```
List<Person> allPeople = new List<Person>();
// Add a bunch of people here...
```

```
IEnumerable<Person> adults =
        from person in allPeople
        where person.Age >= 18
        select person;
```

    - One item to keep in mind is to not modify the original list during a query statement, as this will cause the program to crash

More Complicated where Clauses

- You are allowed to combine multiple things into one, using the standard && and|| operators

```
// In the state of California, kids need a child restraint
// in a car, if under 5 or under 60 pounds.
IEnumerable<Person> kidsNeedingChildRestraints =
    from person in allPeople
    where person.Age <= 5 || person.Weight < 60
    select person;
```

- Multiple where clauses can also be used, which all must be true for it to count (essentially and implicit &&)

```
// In the state of Utah, you need a booster seat if they are 8 or younger and also
// less than 57 inches tall. (We'll assume the height variable is in inches for now.)
IEnumerable<Person> kidsNeedingChildRestraints =
    from person in allPeople
    where person.Age <= 8
    where person.Height < 57
    select person;
```

Multiple from Clauses

- Multiple from clauses can be brought into one query, which allows data to be pulled from multiple places, or to dig further into the results from the first from clause

Ordering Results

- The orderby clause can be used to dictate how the results should be given back to you
    - You also can sort in ascending or descending order

```
IEnumerable<Person> tallestToShortest =
    from person in allPeople
    orderby person.Height descending
    select person;
```

Retrieving a Different Type of Data

- Instead of returning a subset of the input list (Person object, in this case), we could return a list of a different type (ex. return the first names in string format)

Method Call Syntax

- Everything that we've seen with query syntax could alternatively be done with a variation called method call syntax
    - This is often less readable, but allows you to do a few other things that query syntax does not

```
IEnumerable<Person> allAdults = allPeople.
    Where(person => person.Age > 18).
    OrderByDescending(person => person.Age);
```

Chapter 37 – Threads

- Most modern computers have more than one processor, generally contained on one processor chip
    - Unless your code is intentionally structured to run on multiple processors, it will only ever use one
- The basic process of threading is that we can take chunks of code that are independent of each other and make them run in separate threads
    - A thread is almost like its own program, in that the computer will run multiple threads all at the same time on different processors (although memory is still shared)
- When many threads are running, the computer will let one thread run on each processor for a little while, and then it will suddenly switch it out for a different one
- Context switching occurs when the computer lets each processor run for a little while, before switching it out for a different one
    - Your code must be able to handle this processor switching, or there will be strange errors that will appear in your program over time

Threading Code Basics

- To run a separate thread, you will need to create a new thread, tell it what method to run, and then start it

```
public static void CountTo100()
{
    for (int index = 0; index < 100; index++)
        Console.WriteLine(index + 1);
}
```

```
Thread thread = new Thread(CountTo100);
thread.Start();
```

  - You'll also need to add a new using directive to get access to the Thread class
  - Note that the Thread object is a delegate, which requires a method to be passed in (return type void)
- To have the original thread wait at some point for this new thread to terminate, we would use the Join method

```
thread.Join();
```

  - When a thread runs into this statement, it freezes and waits there until the other thread finishes up, effectively joining the execution of the two threads
- You can create as many threads as you want, and they'll all get their fair share of processing time, but note that too many threads take time to create and could cause context switching bottlenecks
- You'll also never get the exact same output two times in a row, due to the operating system performing context switching as it sees fit; hence the timing and ordering is unpredictable
  - This should be a consideration before deciding to run things on separate threads
- The thread can also be put to Sleep for a certain amount of time, which is useful when you need one thread to wait for another thread to do something

```
Thread.Sleep(1000);
```

Using Parameterized Thread Start

- There is an alternative ThreadStart delegate that allows us to take a single parameter, called ParameterizedThreadStart
  - This delegate has a void return type, but takes a single parameter of type object which means that we can cast that object to anything
  - While it may seem limiting to only have the option of passing in an object, but this is incredibly versatile as it can be casted to whatever you need
  - Since the ParameterizedThreadStart doesn't allow you to return information, you could easily construct an object that has a property that can store what would have been returned

```
namespace Threading
{
    public class DivisionProblem
    {
        public double Dividend { get; set; } // the top
        public double Divisor { get; set; }  // the bottom
        public double Quotient { get; set; } // the result (normally would be returned)
    }

    class Program
    {
        static void Main(string[] args)
        {
            Thread thread = new Thread(Divide);

            DivisionProblem problem = new DivisionProblem();
            problem.Dividend = 8;
            problem.Divisor = 2;

            thread.Start(problem);
            thread.Join();

            Console.WriteLine("Result: " + problem.Quotient);

            Console.ReadKey();
        }

        public static void Divide(object input)
        {
            DivisionProblem problem = (DivisionProblem)input;
            problem.Quotient = problem.Dividend / problem.Divisor;
        }
    }
}
```

Thread Safety

- In order to ensure that the data integrity is upheld when threading, certain sections of code that we know might be problematic must be thread safe, meaning that the code can prevent bad things from happening when multiple threads want to use it
- If we have a certain block of code that modifies data that could be accessed by other threads, we call that section a critical section
    - The principle of only allowing one thread in at a time is called mutual exclusion and the mechanism that is used to enforce this is often called a mutex
- To enter a critical section, we require that the thread that is entering the section acquire the mutual exclusion lock for a specific object
    - When the thread has this lock, it can enter the critical section and execute the code, releasing the lock once complete
    - Threads that try to grab a lock that is already taken will be suspended until the lock is released
- The first step is to create some sort of object that is accessible to anything that needs access to the critical section, which can act as the lock
    - Often, this is best done with a private instance variable in the same object or class as the data that is being modified

```
private object threadLock = new object(); // Nothing special needed. Any object will do.
```

    - You could alternatively use a static variable instead of an instance variable if you need thread safety across all instances of the class
- To make a block of code thread safe, you simply add a block surrounding the critical section with the lock keyword

```
lock(threadLock)
{
    // Code in here is now thread safe. Only one thread at a time can be in here.
    // Everyone else will have to wait at the "lock" statement.
}
```

Chapter 38 – Asynchronous Programming

- Asynchronous programming is difficult, and this chapter only briefly introduces us to the concepts

What is Asynchronous Programming?

- Asynchronous programming means taking a piece of code and running it on a separate thread, freeing up the original thread to continue and do other things while the task completes
    - A typical use case is when you start something that you know is going to take a while to happen, while a while could mean a few milliseconds, or much longer
    - Some common scenarios include requests to a server or website, making requests to the file system, or performing algorithm or processing work
- Modern computers have a lot of computational power, and users expect a responsive user interface
    - These two points mean that asynchronous programming is becoming more and more necessary and important to know

Approaches from the Early Days

- Asynchronous programming means taking a piece of code and running it on a separate thread, freeing up the original
- Up to now, we've been performing synchronous requests, which are clean, simple, and ideal

```
Score[] highScores = highScoreManager.LookupScores();
```

- Some other early options for performing asynchronous work included:
    - Using a ThreadPool to queue up work, which is somewhat simpler than using threads
    - Implementing an event-based asynchronous approach, which raises an event once the method has been completed, although shortcomings include splitting up your code and dealing with events
    - Using the AsyncResult pattern, which essentially provides a 'ticket' to the request, but adds significant complexity to the process
    - Run a callback method, which uses delegates that will be called upon completion

The Task-based Asynchronous Pattern

- Asynchronous programming has been a mess in the past not only with C#, but with every language
- C# 4.0 introduced the Task-based Asynchronous Pattern (TAP), which lays the foundation for the current solution that was introduced in C# 5.0
- TAP introduces two new classes while are Task and Task<TResult>
    - Task represents a long running asynchronous chunk of work, while Task<TResult> is a generic variant that additionally serves as a promise of a resulting value when the task is completed
    - Task is used when a return value is not needed, while the generic version can be used when a result is required

```
public Task<Score[]> LookupScores()
{
    return Task.Run(() => {
        // Do the real work here, in a synchronous fashion.
        return new Score[10];
    });
}
```

    - Instead of directly returning the score array, we return a Task that contains it, or more accurately, we return a task that promises to give us an array of scores when it completes

The 'async' and 'await' Keywords

- Building off the TAP pattern, C# 5.0 made asynchronous programming far simpler and easier to read by integrating it into the language itself, done by using the async and await keywords
    - The Task and Task<TResult> classes still function as the core building block of asynchronous programming with these new keywords, and you'll still create a new Task or Task<TResult> using Task.Run, like in the previous section
    - The real difference happens when you get a Task returned to you from a method

```
Score[] highScores = await highScoreManager.LookupScores();
```

- While previous approaches to asynchronous programming have been convoluted, using Task or Task<TResult>, combined with the new await
- keyword, allows everything to fall into place to get a clean solution

```
public async void GrabHighScores()
{
    // Preliminary work...
    Console.WriteLine("Initializing asynchronous lookup of high scores.");

    // Start something asynchronously.
    Score[] highScores = await highScoreManager.LookupScores();

    // Work to be done after completion of the asynchronous task.
    Console.WriteLine("Completed asynchronous lookup of high scores.");
}
```

    - The LookupScores method returns a Task<Score[]>, which is an uncompleted task with the promise of having a Score[] when finished
        - Any time that a method returns a task, you can optionally "await" it with the await keyword
    - The preliminary code before the await happens immediately, by the thread that called the method, but once the await is reached, the thread that called the method jumps out of the method and continues merrily on its way to something else
        - The asynchronous code will either happen on a separate thread (if you use Task.Run) or perhaps on no thread at all (if it's waiting for the network or file system to complete some work)
    - Everything after the await keyword is scheduled to happen when the task finishes
- The await keyword also extracts the value out of the Task<Score[]> upon completion
    - Note that if the non-generic Task is returned, which doesn't have a promised value, you can still await it, but it must be structured as a statement rather than an assignment

```
Console.WriteLine("Before await.");
await SomeMethodThatReturnsTask();
Console.WriteLine("After await.");
```

- The await keyword can only be called inside of a method that is marked with the async keyword

Chapter 39 – Other Features in C#

Iterators and the Yield Keyword

- In Chapter 24, the IEnumerable<T> interface was introduced, which stated that anything that implements IEnumerable<T> (or the non-generic variant IEnumerable) can be used in a foreach loop
    - Recall that an IEnumerable is simply anything where you can examine multiple values in a container or collection, one at a time
        - This capability makes types that implement interface iterators
- In addition to a straight up simple IEnumerable implementation, you can additionally define an iterator using the yield keyword

```
class IteratorExample : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        for (int index = 0; index < 10; index++)
            yield return index;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

    - The yield return will return the next item that is "yielded"
- Unlike a normal return statement, when we use yield return, we're saying, "pause what you're doing here and return this value, but when you come back, start up again here"
    - Note that the compiler isn't turning all the yielded values into a list, but rather returning one at a time
- The yield return syntax can only be used inside of methods that return IEnumerable<T> or their non-generic counterparts

Constants

- Marking a variable as const tells the compiler that we're assigning it a value that will never change, and will actually provide an compiler error if a change is attempted
    - Anything marked with const is automatically treated as though it were static, and hence does not need to be marked as such
    - This is a compile-time constant that must be assigned a value when created
- The readonly keyword is like this, but the value can either be assigned at declaration or within a constructor
    - This is a runtime constant, which can be different in every instance of a class
    - This is an excellent way of building immutable types, as discussed earlier in the book

Unsafe Code

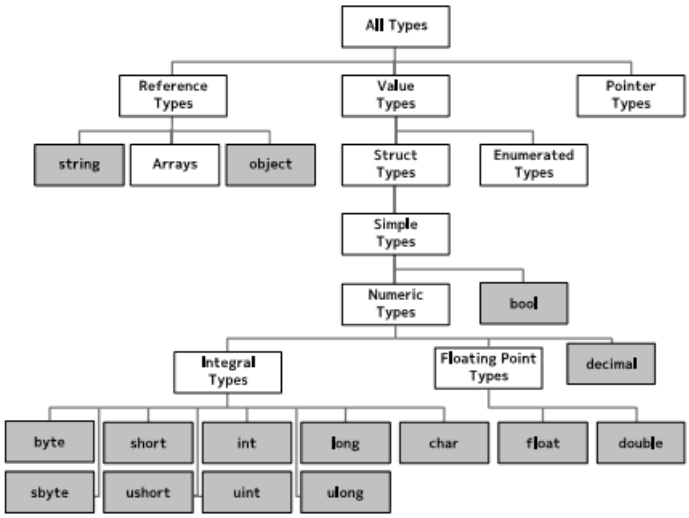
- As we have previously learned, all memory in C# is managed for us
    - This is unlike C and C++, which did not have this feature
- If we need to, we can free ourselves of managed code by using the unsafe keyword either within a method, or as part of the method declaration

```
public void DoSomethingUnsafe()
{
    unsafe
    {
        // You can now do unsafe stuff here.
    }
}
```

    - Unsafe code doesn't mean that the code is dangerous, rather that the code is unverifiable, and the compiler can't guarantee safety
- In these contexts, pointer types can be created in addition to value and reference types
    - Pointer types store the actual memory address for a variable, and aren't managed references or garbage collected
    - These exist solely for interoperability with code that is running outside of the .NET Framework, and can be declared by placing a * by the type

```
int* p; // A pointer to an integer.
```

- It will be pretty rare to need or even want unsafe code and pointer types, but it is worth knowing that it's an option



Attributes

- Attributes in C# provide a way for you to add metadata to an element of your code
    - They can be added to the types that you create, their members, and parameters/return types within a method declaration
    - These attributes can be detected by the compiler, external code tools like unit testing frameworks, or even detected while your code is running by using reflection
- There are hundreds of types of attributes that are built into the .NET Framework
    - As an example, the Obsolete attribute can be used to mark classes and methods that are out of date and should no longer be used

```
[Obsolete]
private void OldDeadMethod()
{
}
```

    - The compiler uses this attribute to detect calls to obsolete methods and warns the programmer about it by displaying a warning in the Error List
- Attributes are simply classes that are derived from the Attribute class, and you can make your own, if needed
- In most cases that an attribute is used, people put 'Attribute' at the end of the name for their attribute (ex. ObsoleteAttribute)

The 'nameof' Operator

- It's quite common to want to do something with the name of a property, method, type, or other bit of code
    - Among other things, this is useful in debugging and when you're doing data binding in a UI framework like WPF or Windows Forms, where property names are frequently used
- The nameof operator allows you to refer to a variable, type, or type member (like a method or property) and have the compiler change it into a string that matches the name
    - The syntax is special because you normally can't access a class' properties unless they're static properties (which they aren't)

```
public override string ToString()
{
    return $"[{nameof(Book)} {nameof(Title)}={Title} {nameof(Author)}={Author} " +
                $"{nameof(Pages)}={Pages} {nameof(WordCount)}={WordCount}]";
}
```

Bit Fields

- There are a few cases where we want to fall back down to a low level, and work with individual bits and bytes
    - For example, if we want to store permissions for a large online discussion forum, we may want to use a single bit to manage each permission, rather than assigning a bool that amounts to eight bits per permission, where '1' would indicate true and '0' false

0	0	1	0	0	0	1	1
(Not used)	Account Suspended	Create threads	Delete others' posts	Edit others' posts	Vote on posts	Edit own posts	Create posts

  - This trick is actually very widespread, especially for things like large data sets, operating systems, and sending things across networks
- Bit shift operators can be used to work with bits, and allows bits to be moved over a certain number of slots
  - The left bit shift operator (<<) moves the bits left a certain number of spaces, while the right operator does the opposite

```
int result = 5 << 2; // 00000101 turns into 00010100, or 20.
```

- Bitwise logical operators, such as the 'and' (&) and 'or' (|) go down the bits, one at a time, and do an 'and' or an 'or' operation on each bit

```
int a = 5; // 00000101
int b = 3; // 00000011
int combinedWithAnd = a & b; // results in 00000001, or 1.
int combinedWithOr = a | b;  // results in 00000111, or 7.
```

  - A third bitwise operator is the exclusive or operator (^), called xor, which is only true if exactly one is true

```
int combinedWithXor = a ^ b; // results in 00000110, or 6.
```

  - The last is the bitwise complement operator (~), which is a little like the ! operator, and takes each bit and changes it to the opposite

```
int number = 200; // 11001000
int bitwiseComplement = ~number; // 00110111, or 55.
```

- Note that all these operators can be used in a compound manner (ex. <<=, &=, etc.)
- Enumeration Flags are used to make this more readable

```
[Flags]                         // Don't forget to add the Flags attribute.
public enum ForumPrivileges
{
    CreatePosts =      1 << 0, //  1 or 00000001
    EditOwnPosts =     1 << 1, //  2 or 00000010
    VoteOnPosts =      1 << 2, //  4 or 00000100
    EditOthersPosts =  1 << 3, //  8 or 00001000
    DeletePosts =      1 << 4, // 16 or 00010000
    CreateThreads =    1 << 5, // 32 or 00100000
    Suspended =        1 << 6, // 64 or 01000000

    // Note we can also add in "shortcuts" here:
    None = 0,
    BasicUser = CreatePosts | EditOwnPosts | VoteOnPosts,
    Administrator = BasicUser | EditOthersPosts | DeletePosts | CreateThreads
}
```

- To use this in practice, some combinations of these ideas can be used
  - The | operator can be used to "turn on" a bit in the bit field

```
ForumPrivileges privileges = ForumPrivileges.BasicUser;
privileges |= ForumPrivileges.Suspended; // Turn on the 'suspended' field
```

  - The & operator can be used to check if a flag is set, using some simple trickery

```
bool isSuspended = (privileges & ForumPrivileges.Suspended) == ForumPrivileges.Suspended;
```

    - The trick here is that if we do a bitwise and operation with something that only has one bit set to true, it will either become all 0's, which indicates that the field was not set, or if the field was set, we'll get back the value of the field we were checking for, turning all other fields to 0
  - Using a combination of the & and the ~ operators, we can turn off a field

```
privileges &= ~ForumPrivileges.Suspended;
```

  - You can also toggle a field on using the ^ operator

```
privileges ^= ForumPrivileges.DeletePosts;
```

Reflection

- The ability to have code analyze the structure of other code is called reflection
  - The biggest use for reflection is when you want to look at an unknown assembly or object, although doing this is always slower than directly accessing the code

- Uses such as unit testing an assembly to find methods that end with the word “Test”, or to find all classes in a DLL that implement a certain interface are a couple examples
- The core class used in reflection is the Type class, which represents a compiled type on the .NET Framework, including classes, structs, or enumerations
    - For any given type, you can use the typeof keyword to get the Type object that represents it

```
Type type = typeof(int);
Type typeOfClass = typeof(MyClass);
```

- With a given type, you can figure out what constructors, methods, properties, or other things the type has defined

```
ConstructorInfo[] contructors = type.GetConstructors();
MethodInfo[] methods = type.GetMethods();
```

Using Statements and the IDisposable Interface

- Earlier on in the book, we used a Close method when we were finished reading a file, which freed up any unmanaged memory or resources that were in use
    - There is a better way to deal with these unmanaged resources
- Typically, types that access unmanaged memory will implement the IDisposable interface
    - These classes have a Dispose method which cleans up any unmanaged memory the object is using
    - We could directly call this method (like how we call the Close method), but there are issues that may come up if an exception is thrown while the file is open, for instance
- A better way of handling unmanaged code is to use a using statement (not to be confused with a using directive)

```
using (FileStream fileStream = File.OpenWrite("filename"))
{
    // Do work here...
}
```

    - When the ending curly brace is reached, the object inside of the parentheses in the using statement is disposed (the Dispose method is called), and this holds true even if an exception is thrown in the middle of the block
- A using statement is a very readable and simple solution for dealing with unmanaged memory

Preprocessor Directives

- Many languages, including C#, give you the ability to include instructions for the compiler within your code
    - These instructions are called preprocessor directives (even though the C# compiler does not have a preprocessor, like some languages)
- These preprocessor directives all start with the # symbol, which tips you off to the fact that something is a preprocessor directive
- Two simple preprocessor directives are the #warning and #error directives, which give you the ability to make the compiler emit a warning or error with a specific message
    - One example of when you may want to do this is if you intend to change some section of code, you could use a #warning to remind you to do so

```
#warning Enter whatever message you want after.
#error This text will show up in the Errors list if you try to compile.
```

- The #region and #endregion directives allow you to add little regions to your code, which Visual Studio then allows you to collapse and expand

```
#region Any region name you want here
public class AwesomeClass
{
    // ...
}
#endregion
```

- Compilation symbols are special names that can be defined (turned on) or undefined (turned off – the default)
    - For example, there’s a DEBUG symbol that is defined when you compile in debug mode, but not when you compile in release mode
    - Symbols can be set either at the top of a file, or within the entire solution (added via the solution explorer)
- The #if, #else, #elif, and #endif directives work very much like an if statement, expect that they’re instructions for the compiler to follow, and don’t end up in your final program

```
public static void Main(string[] args)
{
#if DEBUG
    Console.WriteLine("Running in debug mode.");
#else
    Console.WriteLine("Running in release mode.");
#endif
}
```

- Note that it's always a little dangerous to have things run differently in the final release version than in the test version, because you may not discover all the bugs it has until the program has been released

Implicitly Typed Local Variables and Anonymous Types

- C# provides a way to infer the type (implicitly typed) of a variable at compile time using the var keyword

```
var a = 3;
```

  - It is very important to understand that the type in use is determined at compile time, based on what is assigned to it, and this is not the same as having no type (or any type)
  - There are some languages that are weakly typed, where a variable can contain any type of information, but the var keyword does not do this since the compiler will turn it into a type using type inference at compile time
- Type inference only applies for local variables, as instance variables or static class variables must always have an explicit type
- It is generally advised to not use implicitly typed variables since forcing a human to infer a type reduces the readability of the code
  - Note that anonymous types can be created, which require the use of the var keyword

```
var anonymous = new { Name = "Steve", Age = 34 };
Console.WriteLine($"{anonymous.Name} is {anonymous.Age} years old.");
```

  - When using anonymous types like this, we can't use an explicit type, because we've never created a class for it

Nullable Types

- Typically, bool (being a value type) contains only true and false
  - The concept of nullable types allows the use of null for value types, as it uses the Nullable<T> generic type
- C# provides a shorthand way of using the nullable type

```
bool? nullableBool = true;
```

Simple Null Checks: Null Propagation Operators

- C# 6.0 introduced a new feature called null propagation that is quite powerful
  - One of the greatest annoyances of programming in many languages is null checking
- For an example like the one below, any one of five parts of the statement could turn out to be null, which would bring up a null reference exception and crash the program

```
private string GetTopPlayerName()
{
    return highScoreManager.Scores[0].Player.Name;
}
```

  - Null checking is a very good idea, and is good defensive coding practice
- The null propagator operator (either ?. or ?[]) makes it easy to do these null checks without causing the code to become ugly
  - If any of the objects involved turn out to be null, the expression will short circuit and evaluate to null

```
return highScoreManager?.Scores?[0]?.Player?.Name;
```

Command Line Arguments

- A very popular alternative to manually typing in commands is to use command line arguments
  - This allows you to put your program in scripts, and run it as a part of a larger automated task

```
C:\Users\RB\Documents>Add.exe 3 5
```

- These numbers on the end get pulled into your program as command line arguments

```
static void Main(string[] args)
{
    int a = Convert.ToInt32(args[0]);
    int b = Convert.ToInt32(args[1]);

    Console.WriteLine(a + b);
}
```

User-Defined Conversions

- C# allows you to define your own implicit or explicit casts

```
public static implicit operator MagicNumber(int value)
{
    return new MagicNumber() { Number = value, IsMagic = false };
}
```

  - Like with all other operators, this must be static and public, and we can choose whether the cast will be implicit or explicit

```
int aNumber = 3;
MagicNumber magicNumber = aNumber;
```

- Explicit casts are used whenever your losing information in the conversion process

```
static public explicit operator int(MagicNumber magicNumber)
{
    return magicNumber.Number;
}
```

  - This time, we'll have to explicitly state that we want to convert from one type to another

```
MagicNumber magicNumber = new MagicNumber() { Number = 3, IsMagic = true };
int aNumber = (int)magicNumber;
```

- Just because we can create a user-defined conversion does not mean that we necessarily should
  - It can be too easy, especially with an implicit cast, to assume that one object references another object in the heap, when, a new object has been created that does not contain a reference
  - Instead, consider adding in a new constructor, or a ToWhatever or FromWhatever method to your type, as it will be much more obvious what is happening

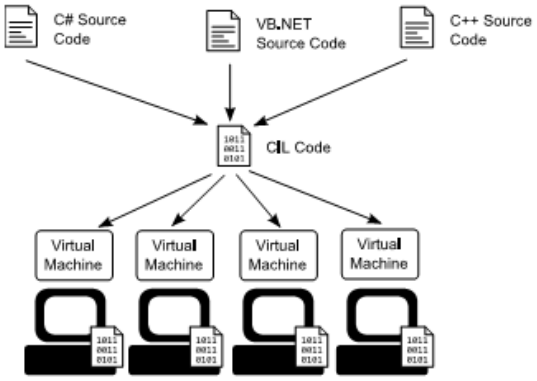
Chapter 40 – C# and the .NET Framework

Binary, Assembly, Languages, and Compiling

- Instead of working with raw 1's and 0's, there's a low-level language called assembly language which is a human-readable form of binary
  - For the most part, assembly directly mirrors binary, and each line of assembly will result in a single instruction in the binary code
- Compiling code is simply translating from a higher-level language to a lower level language
- Programming languages are built in a way that mimics a human language, but has a formal structure to it that allows for a compiler to figure out, step-by-step, what binary instructions it needs to use to make it happen

Virtual Machines and the CLR

- Without a virtual machine, you'd need a different compiler (or at least different configuration settings) for each hardware architecture that you want to have your code run on, as well as a different compiler for each programming language
  - To deal with this, instead of trying to make compilers for each language and computer architecture, one very special architecture, called a virtual machine, is used
  - A virtual machine is a software program that mimics a full computer, and provides controlled access to the hardware of the computer that the VM is running on
- The Common Language Runtime, or CLR, is the .NET Framework's virtual machine
  - Rather than compiling directly to binary instructions, languages that run on the CLR emit code in a special type of assembly language called the Common Intermediate Language (CIL or IL), which the CLR reads and ultimately compiles to binary instructions



- When it comes time to load an assembly, like an EXE or DLL, the CLR figures out what types it includes and what methods each one has, and builds a table for them
    - When it comes time to run a method, if it has not been run before, the CLR will load the IL code for that method and compile it to binary code, which is called Just-in-Time (JIT) Compiling
    - The next time this method runs, it sees that is was already compiled and just re-runs it, and hence JIT compiling only happens once for a given method

Advantages and Drawbacks of the CLR

- Some advantages of the CLR include memory management, security, and support of multiple languages
    - Drawbacks may include slower performance (but not always), the inability to write low-level code that operating systems or drivers require, and the potential for somebody to be able to reverse-engineer your code

The BCL and FCL Class Libraries

- In addition to the CLR, the .NET Framework also includes a very large collection of previously built code and types
    - The collection is massive, and contains more than the "standard library" of most languages, including C++
- This large collection is called the Base Class Library (BCL)
    - The BCL contains all the built-in types, arrays, exceptions, math libraries, basic file I/O, security, collections, reflection, networking, string manipulation, threading, etc.
    - Typically, anything that includes the System namespace is part of the BCL
- In addition to the BCL, the .NET Framework covers broad functional areas such as database access and graphical user interfaces (Windows Forms or WPF)
    - This entire collection, including BCL, is called the Framework Class Library (FCL)
    - In casual discussion, sometimes people use FCL and BCL interchangeably, which isn't strictly correct, but it is perhaps good enough for most things
- After going through this book, you'll know nearly everything there is to know about C# itself

Try It Out!

.NET Framework Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** C# is compiled to binary that is immediately executable by the computer.
2. Is a virtual machine software or hardware?
3. What is the name of the .NET Framework's virtual machine?
4. Name the two key parts of the .NET Framework.
5. **True/False.** A slightly different version of the CLR is required for every platform you want to run C# code on.
6. **True/False.** The CLR understands C# code.
7. Name two other languages that can be used to write code for the .NET Framework.
8. **True/False.** Virtual machines will always be slower than compiling directly to binary.
9. **True/False.** IL code is easier to reverse engineer than compiled code.

Answers: (1) False. **(2)** Software. **(3)** Common Language Runtime (CLR). **(4)** the CLR and the BCL/FCL. **(5)** True. **(6)** False. **(7)** C++, VB.NET, IronPython, IronRuby, ... **(8)** False. **(9)** True.

Chapter 41 – Getting the Most from Visual Studio

Windows

- The Main Code Window, where you have been typing all your code for all your projects, is the most important window
    - Note that there are two drop-down boxes that you can use to quickly jump to a specific type or member of a type in the current file
    - If you press F12, you'll automatically jump to where that thing is defined, and pressing Shift + F12 will find all the places in the code where the current selection is used
- Your code within the Solution Explorer is typically organized into namespaces, which are usually placed in separate folders under your project, and each new type that you create is usually in its own .cs file

IntelliSense

- Besides providing auto-completion, IntelliSense also shows you the comments that are provided for any type or its members, including your own XML comments and documentation

Basic Refactoring

- Refactoring is the process of changing the way your code is organized without changing how it functions
    - The idea with refactoring is to make it so that your code is better organized and cleaner, making it easier to add new features in the future
- ReSharper is an expensive Visual Studio add-on that provides a massive amount of refactoring support among many other useful features
- Visual Studio also has a few basic refactoring tools that are worth pointing out
    - Choosing the Rename tool (F2) allows you to change a name everywhere in your code
    - If you have a block of code that you want to pull out into its own method, you can select the code, right-click, and choose Quick Actions, then Extract Method

Keyboard Shortcuts

- **F5:** Compile and run your program in debug mode.
- **Ctrl + F5:** Compile and run your program in release mode.
- **Ctrl + Shift + B:** Compile your project without attempting to run it.
- **Ctrl + Space:** Bring up IntelliSense.
- **Ctrl + . :** Show Quick Actions.
- **Ctrl + G:** Go to a specific line number.
- **Ctrl +]:** Go to the matching other curly brace (**{** or **}**).
- **F12:** Go to the declaration of a variable, method or class.
- **Shift + F12:** Locates all places where something is referenced, throughout the project.
- **Ctrl + R then Ctrl + R:** Rename an element of code.
- **Ctrl + R then Ctrl + M:** Extract selected code into its own method.
- **Ctrl + -:** Move back to the last place you were at.
- **Ctrl + Shift + -:** Move forward to the place you were at before moving back.
- **Ctrl + F:** Find in the current document.
- **Ctrl + Shift + F:** Find in the entire solution.
- **Ctrl + H:** Find and replace in the current document.
- **Ctrl + Shift + H:** Find and replace in the entire solution.

Chapter 42 – Referencing Other Projects

- There are piles of code out there that have been written by smart people, are well tested, and do really great things
    - You can take advantage of other libraries and DLL files to get a quicker start on your own project, or even reuse code that you made in previous projects

Referencing Existing Projects

- The simplest approach to reusing another external collection of code is by referencing another project, either one you wrote yourself or one that you have source code for (ex. open source)
    - In order to add the project as a reference, add the existing project to your solution, add a reference to the project (right-click on references, and Add ➔ Reference), and add the appropriate using directives for the simpler, unqualified names

Adding a Reference to .NET Framework Assemblies

- The .NET Framework is massive, and contains many assemblies that are not are not made immediately accessible, in order to keep your project light
    - If the compiler is unable to find the type you're looking for after adding a using directive, then a reference to an extra .NET assembly might be required
- These references are added in the same way as adding a reference to an existing project

Adding a DLL

- It is common to get executable code, packaged in a DLL, as they are versatile and widespread
    - Add it in the same way as previously described, although this time you will browse to the file to get the reference

Chapter 43 – Handling Common Compiler Errors

Compiler Warnings

- Instead of an error, sometimes you'll get a more minor problem called a warning
    - Warnings can sometimes be more dangerous than an error, however
- It is best to try to treat compiler warnings as errors, and eliminate them as soon as you can
    - Don't let dozens of warnings pile up, since fixing them early will save trouble down the road

Chapter 44 – Debugging Your Code

- Once you get through any compiler errors, you can run your program
    - After compilation, errors could still pop up, which are now called runtime errors
- Visual Studio gives you the ability to dig into the program while it is running to see what is going on, which is called debugging

Launching Your Program in Debug Mode

- The first step is to make sure that your program is in debug mode
    - When the program is compiled in debug mode (as opposed to release mode), it includes a whole lot of extra information in the EXE file to allow Visual Studio to keep track of what is being executed
    - This extra information slows down your program and makes the EXE file significantly larger, so ensure that once the bugs are corrected, that the program is deployed in release mode

Viewing Exceptions

- When your program is running in debug mode, unhandled exceptions will pause the program and your computer will switch back to Visual Studio for you to look at the problem
    - Local variables can now be looked at by hovering over them to see the current value
    - The call stack window will tell you what the current method is, and what methods had been called, which is one of the most useful tools that you must figure out what happened

Editing Your Code While Debugging

- In some cases, you can edit your code while it is running and then continue running the program, exactly where you left off, with the edited code

Breakpoints

- Whenever you want, you can set a breakpoint on a line of code in Visual Studio, which will cause the program to halt at that point
    - Breakpoints can be added and removed even while the program is running

Stepping Through Your Program

- Sometimes a program will be executing, but taking a long time at some location
    - This can mean that it is stuck in a loop, and by hitting the Pause button in the debug toolbar, the execution will be halted where it is currently running

- The restart button will close the current execution and start over from the beginning

- The Step Into, Step Over, and Step Out buttons allow you to advance one line at a time, but behave differently when encountering methods

- The Run to Cursor button is useful if you want to skip ahead to the location of your cursor

Try It Out!

Debugging Quiz. Answer the following questions to check your understanding. When you're done, check your answers against the ones below. If you missed something, go back and review the section that talks about it.

1. **True/False.** You can easily debug a program that was compiled in release mode.
2. **True/False.** Debug mode creates executables that are larger and slower than release mode.
3. **True/False.** If an exception occurs in debug mode, your program will be suspended, allowing you to debug it, even if you haven't set a breakpoint.
4. **True/False.** In release mode, Visual Studio will stop at breakpoints.
5. **True/False.** In some cases, you can edit your source code while execution is paused and continue with the changes.

Answers: (1) False. **(2)** True. **(3)** True. **(4)** False. **(5)** True.

Chapter 45 – How Your Project Files are Organized

- The general rule for what goes into version control is that user-specific project files don't belong in version control (everyone should have their own copy), and anything that can be rebuilt from other things (like compiled executable files) should be skipped as well

The Solution Directory

- The top-level folder corresponds to the solution, which can be thought of as a collection of projects that are all related and used to accomplish a specific task
- The .sln file is your solution file, which contains information about what projects are contained in the solution, as well as solution-specific settings like build configurations
    - This should be included in a version control system
- The .suo file is the solution user option file, and contains various settings fro the project that Visual Studio uses
    - This contains things like what files you had open when you last used the project, so that they can get reopened when the project is reopened
    - This should not be included in a version control system
- Note that in your solution directory (folder), each project will have its own folder
    - This explains why there are two seemingly duplicate folders that are created when a new project is started

The Project Directory

- The .csproj file defines your project, and identifies what files are included in the project, along with listing any other assemblies, libraries, or projects that the project uses
    - This should be included in a version control system
- The .csproj.user file contains user-based settings, but for your project rather than the solution
    - This should not be included in a version control system
- The bin and obj folders will appear after the project is compiled for the first time, and are used for building and storing executable versions of your projects and solutions (EXE and DLL files)
    - The obj directory can be thought of as a staging area for the final executable code
    - The bin folder is where the final executable code will go

- In either of these folders, there will be a Debug or Release folder, depending on how the project was built
    - Both the bin and obj directories should not be included in a version control system

The Properties Folder

- The Properties folder contains properties and resources that you've added to your file, and can be quite large for a large project
    - At a minimum, you'll probably see an AssemblyInfo.cs file in the Properties folder, which contains information about your project, including versioning information
    - This should be included in a version control system