C# IN SIMPLE TERMS

The easiest way to learn C#

Matthew P Jones

Thank You Dear Reader!

Before we go any farther, I must take some time to say the following: thank you very much, dear reader, for becoming a paid subscriber and downloading this eBook! I appreciate your support for my projects.

This book is a collection of all of my C# in Simple Terms posts that were originally published on Exception Not Found from September to December of 2020. Each post roughly corresponds to a chapter in this eBook, and the chapters are in the original publishing order.

Each chapter has been updated and reformatted for easier reading in an eBook style, including links to other chapters, formatted code blocks, and more. I am aware that reading an eBook is a much different experience than reading a website and have made many changes to the original text in order to ensure that the reading experience is just as good here as on the main blog.

How to Read This Book

There are a few things I would like my dear readers to know before they get started reading this book.

The Sample Project

First, you should know that there is a sample project, hosted on GitHub, which contains all the code samples in this book. You can find it here.

The sample project is designed to be used independently of this book, excepting the bonus chapters. You can use either that project or this book to learn about C# and its features, though my personal recommendation is to use this book, because it contains much more reasons as to why we do things in a certain way.

Glossary Words

Throughout this eBook, you will find words written *in this style*. Those are glossary words and are defined in the Glossary at the end of this eBook.

Found an Error? Tell me about it!

If you find an error in this eBook, please tell me about it by emailing me at exceptionnotfound1@gmail.com or contacting me on Twitter.

Get the Bonus Chapter!

This version of this eBook does not have the bonus chapter, Chapter 23: Asynchronous Programming. To get this chapter, you must be a paid subscriber on my blog Exception Not Found. You can sign up to be a paid subscriber here.

Table of Contents

Chapter 1: The Type System

To kick off the first chapter this book, let's learn about the most basic feature C# has: the type system.

Strongly Typed and Type-Safe

The first and most important thing to know about C# as a programming language is this: C# is a *strongly typed* language. This means that every variable, every constant, every class, every single object ever created using C# has a *type*. It is impossible for an object to exist in C# without it having a type.

A type in C# (and .NET) is a set of properties about a specific kind of object. These might include the storage space it needs, the maximum or minimum size of the object, and others.

```
int four = 4; //Max 2147483647 (2^31 - 1), Min 2147483648 (-2^32)
double twopointfive = 2.5; //Size: 8 bytes
char a = 'a'; //Size: 16 bit
```

The lines above demonstrate how to create a variable in C#. The first word in each line above is the type (e.g. `int`, `double`, `char`), the second word is the variable name (e.g. `four`, `twopointfive`, and `a`) and the number or character on the right side of the equal sign (`=`) is the variable's value.

Because C# is a strongly-typed language, C# is also a *type-safe* language. That means that objects which are *instantiated* as a given type (number, character, date, class, etc.) have rules in place to ensure that said *instance* behaves as that type. Therefore the C# compiler will allow us to add two number types together, but trying to add a number to a word will cause an error.

```
int five = 5;
int ten = five + five; //No problem!
int invalidTen = five + "five"; //Compilation error!
```

Error: Cannot implicitly convert type 'string' to 'int'

System.Object

Because no object can exist in C# without a type, there exists a "base" type that every kind of object inherits from.

In .NET, that "base" type is called `System.Object`.

```
System.Object newObject = new System.Object();
```

We can also instantiate this object using a simplified syntax:

```
object newObject = new object();
```

Type Inheritance

C# and .NET support the concept of *type inheritance*. Types can "derive" (meaning they inherit attributes, properties, and constraints) from other types. For example, all types in C# derive from the base class `System.Object`.

We will discuss inheritance, particularly inheritance involving classes, more thoroughly in Chapter 9 of this series, which covers Inheritance and Polymorphism; for now, just know that types can inherit behavior from other types.

Value, Reference, Implicit, and Null

C# supports two distinct kinds of types: *value types* and *reference types*.

Value Types

Value types are objects whose value is contained by the object. In C#, value types include almost all "primitive types", which we will discuss in Chapter 2. These are often "simpler" types, like numbers or characters.

In C#, all value types inherit from a base class `System.ValueType`, which in turn inherits from `System.Object`, as all types must.

```
int myNumber; //Default value 0
bool myBool; //Default value false
double myDouble; //Default value 0
```

If we do not specify a value when creating a value type, they get a default value.

Reference Types

Reference types, unlike value types, are objects which exist on the memory heap. The variables we create contain a "pointer" or a "reference" to that value on the memory heap; the variable does not contain the value itself. Generally, reference types are more complex types, such as custom classes, queries, and collections (like arrays).

```
Array[] myArray; //Default null
MyClass myClass; //Default null
MyClass otherClass = new myClass();
```

A reference type, if it is initialized without a value, will have the value *null*. We will discuss null later in this chapter.

Because reference types are merely a reference (or pointer) to the object's value, it is possible for objects on the heap to no longer have any references to them. C# includes a feature called *garbage collection*, where automatic memory management will "reclaim" memory from reference types and other sources that are no longer being used. In most cases, you don't need to worry about the garbage collection process; it just happens behind the scenes and doesn't interfere with anything.

Implicit Types (var)

If you've read any C# code, you have probably encountered something like this:

```
var myValue = 5;
```

`var` is a special keyword in C#. It is a "placeholder" for a type which will be determined by the value of the variable. For example, in the line above "myValue" is initialized as type `int`, because the value assigned to it is a simple integer (5).

It is important to note that `var` is not a type unto itself. It is only a placeholder.

Using `var`, we can make our code easier to read. For example, imagine we have a set of variable declarations:

```
double myDouble = 5.6;
char myChar = 'a';
MyClass myClass = new MyClass();
int myInt = 7;
```

Using the `var` keyword, we can simplify our code a bit:

```
var myDouble = 5.6;
var myChar = 'a';
var myClass = new MyClass();
var myInt = 7;
```

In this way, each of these variables has their type implicitly assigned from their value.

Null

We must also talk about a special type in C#: `null`.

Null is a literal type that represents a null reference; that is, one which does not point to an object on the memory heap. It is also the default value of reference types when they are created.

```
//The below two lines will be treated as identical.
MyClass myClass = null;
MyClass myClass;
```

We use `null` when a reference type object does not yet have a value. Instances of reference types, by default, have the value `null`.

When writing code, we often must do *null checking*, which is when we confirm that a particular object is or is not currently `null`.

```
if (myObject == null)
{
    //Do something
}
```

However, we cannot use the `var` keyword and assign the variable to `null`; that will throw a compilation error. This is because the C# compiler cannot determine the type of a variable if the value is `null`.

```
var myClass = null;
```

Error: Cannot assign null to an implicitly-typed variable

New Keywords

- `null` – A special value that represents a missing reference.
- `object` – A shortcut reference to the root type `System.Object`.
- `var` – A placeholder for a type. The actual type is determined by the C# compiler using the value of the instance.

Summary

The type system in C# is robust, able to support value and reference types as well as the special type `null`, though sometimes we need to use null-checking before we manipulate objects that can be `null`.

All types support type inheritance, which is where types may adopt the properties, attributes, and constraints of other types, and because C# is a strongly-typed language, every object ever created must have a type. We can create implicit variables using the `var` keyword, and every single object in C# will inherit from the base class `System.Object`.

Chapter 2: Primitive Types, Literals, & Nullables

As we learned in the previous chapter, C# supports a robust type system. Part of that system is a group of "basic" types. These types, also called *primitive types*, form the foundation of many C# programs.

Number Types

The most basic of the "primitive" types in C# are the number types. These include *integral numeric types* (which represent whole numbers, like 1, 67, 1957321, 8, and so on) and *floating-point numeric types* (which represent non-whole numbers such as 1.2, 6.99, 8234.66, and so on).

Int

Of the integral numeric types, the type `int` is the default and most common. The type `int` represents a 32-bit integer, with a positive or negative value.

```
int five = 5;
int thirteenHundred = 1300;
int negativeForty = -40;
int intMaxValue = int.MaxValue; //(2^31 - 1)
```

The type `int` is used for many kinds of variables, including math, counters, and iterators.

Short, Long, and Byte

The types `short`, `long`, and `byte` are all integral numeric types, like `int`. However, they represent different ranges of values.

A `short` represents a 16-bit integer:

```
short three = 3;
short negativeOneHundred = -100;
short shortMaxValue = short.MaxValue; //(2^15 - 1)
```

A `long` represents a 64-bit integer:

```
long fifty = 50;
long longMaxValue = long.MaxValue; //(2^63 - 1)
```

Finally, a `byte` is a 8-bit integer that only represents positive values.

```
byte four = 4;
byte byteMaxValue = byte.MaxValue; //(2^7 - 1)
```

Signed and Unsigned

Integral types in C# are normally *signed,* meaning they can represent positive or negative values (the exception to this is `byte`, which is *unsigned* and therefore can only represent positive values).

We can use the types `ushort`, `uint`, and `ulong` to represent unsigned integers, and `sbyte` to represent signed bytes.

```
ushort unsignedShortMax = 65535;
uint unsignedIntMax = 4294967295;
ulong unsignedLongMax = 18446744073709551615;
sbyte signedByteMin = -127;
```

Floating-Point Numeric Types

In C#, floating-point numeric types represent non-whole numbers. They are used to do complex math calculations, represent money and currency transactions, and in other situations where we need more than simple integers.

Double and Float

A `double` is an 8-byte number used when we need quick calculations but don't care about *precision* (see "A Note About Precision" below).

```
double fortytwo = 42.0;
double pi = 3.14159;
```

A `float` is used in the same situation as a `double`, but it has less range, since it is a 4-byte number. Consequently, it can perform calculations within its range even more quickly than `double`.

```
float negativeThirty = -30.0F;
float eighteenHundredAndAHalf = 1800.5F;
```

Note that we need the F literal here; there will be more about literals later in this chapter.

A Note About Precision

When using types `double` or `float`, *precision* is lost when doing complex calculations. For example, in C# we can do this:

```
double sum = 0.1 + 0.2;
```

However, we will get a strange result: 0.30000000000000004

When doing arithmetic with `double` or `float`, precision (which is the degree of accuracy of numbers on the right-hand side of the decimal point) is sacrificed to gain speed. Calculations involving floating-point numbers are computationally expensive, meaning it takes a long time (comparatively) to get a result.

Programming language compilers, including C#'s compiler, take shortcuts when doing these kinds of calculations; these shortcuts dramatically speed up the calculations while not reducing precision too much, except in certain circumstances. Most applications will not care that 0.1 + 0.2 = 0.30000000000000004, because precision is not an absolute requirement for this calculation.

For most applications that do not deal with money or currency, we as developers probably don't care about the loss of precision that comes from doing arithmetic using `double` or `float`; it is most likely small enough to be negligible.

However, there are times when precision cannot be lost, and for those times, we use the `decimal` type.

Decimal

The `decimal` type is used when we need to keep precision, but don't mind that calculations are more computationally expensive to do. `decimal` is primarily used for currency or money calculations, since loss of precision would be harmful there.

```
decimal dollars = 1.45M;
decimal billionaire = 1000000000.01M;
```

Note that we need the M literal.

Mixing Number Types

It is possible to use `decimal`, `double`, and `float` in calculations with the integral types, though there are certain rules that the C# compiler will enforce when doing so

For example, using any integral type and an instance of `double` in a calculation results in a value of type `double`.

```
int five = 5;
double fivePointFive = 5.5;
double sum = five + fivePointFive; //Result is type double, value 10.5
```

This works similarly for `float`, though if the resulting value is too large, the type of the result is automatically converted to `double`.

As you might have guessed, mixing integral types and `decimal` gives a result of type `decimal`:

```
short three = 3;
decimal sixPointSevenTwo = 6.72;
decimal sum = three + sixPointSevenTwo; //Result type decimal, value 9.72
```

In general, mixing integral numeric types and floating-point numeric types in math calculations will result in objects which have the floating-point numeric type.

Non-Number Types

Besides the integral numeric types and the floating-point numeric types, there are also several non-number types that C# provides.

Bool

C# includes the type `bool` to represent *boolean* values (values that must be either `true` or `false`).

```
bool isTrue = true;
bool isFalse = false;
```

Boolean values are often utilized in *boolean logic*, which we will demonstrate more of in Chapter 4 of this book.

Char

The type `char` represents a single text character.

```
char a = 'a';
char ampersand = '&';
char x = 'x';
char comma = ',';
char semicolon = ';';
```

String

The type `string` represents a collection of characters, and because of this, we think of `string` as representing text.

```
string sentence = "This is a sentence.";
string otherSentence = "The quick brown fox jumped over the lazy dog.";
```

Please note that by Microsoft's own definition of a "primitive" type, the type `string` is NOT considered a primitive.

The term "string" comes from the idea of this type being a "string" of characters. In fact, the type `string` is implemented as collection of characters, and can be used as though it is an array. We will discuss arrays and collections in Chapter 12.

Also, unlike all the other primitive types in this article, `string` is a reference type, not a value type, meaning (among other things) that its default value is `null`. In a later

article in this series, we will see many ways of manipulating strings using a variety of C# operators.

DateTime

The type `DateTime`, like `string`, is not truly a "primitive" type but it is so commonly used in C# applications that I felt it was worthy of inclusion in this chapter.

An instance of `DateTime` represents a point in time. Typically, this is expressed as a date and a time.

```
DateTime date1 = new DateTime();
DateTime date2 = new DateTime(2020, 3, 15); //15 March 2020
DateTime date3 = new DateTime(2020, 3, 15, 10, 30, 00); //15 March 2020,
10:30
```

There are many ways to create instances of this object; the above is just a few of them. We will see many other ways to manipulate `DateTime` objects Chapter 20: Dates and Times.

Literal Values

The C# compiler makes assumptions as to what type a variable has if we do not directly tell it what that type should be. In C#, if we write the following code:

```
var myValue = 7.8;
```

The type of `myValue` will be `double`, because `double` is the default type for any number with a decimal point.

If we want to instantiate `myValue` as type `decimal`, we need to declare it with the *literal marker* M.

```
var myValue = 7.8M;
```

There are many types of literal markers, including:

```
var myDouble = 5.6D; //double
var myFloat = 2.88F; //float
var myLong = 568373L; //Type long or ulong if the value is too large
var myUnsignedInt = 98765U; //Type uint or ulong if the value is too large
```

Nullable Types

C# allows for the use of *nullable types*, where a primitive value type can be either one of its "normal" values or `null`. Nullable types are identified with the operator `?`.

Nullable types get the value `null` as their default value.

```
char? a = null;
double? myDouble; //Value will be null
decimal? myMoney = 45.61M;
bool? trueFalseOrNotFound = false;
DateTime? myDate = null;
int? myNumber = null;
float? myFloat = 6.3F;
```

Summary

Basic types in C# include integral numeric types `int`, `long`, `short`, and `byte`; floating-point numeric types `double`, `float` and `decimal`; and non-numeric types `bool`, `char`, `string`, and `DateTime`, plus others; each has a distinct purpose.

Combining objects of different numeric types in math calculations generally results in an object having the more-general type (e.g. combining an `int` and a `double` will result in a `double`, combining a `short` and a `decimal` results in an object of type `decimal`, etc.)

To specify which type a given value should be, we can use literal markers, such as `M` for `decimal` or `F` for `float`. We can also make some objects nullable using the `?` operator to allow them to have the value `null` in addition to the normal values those types can have.

Chapter 3: Casting, Conversion, Parsing, and Type Checking

So far, we have discussed the basics of the type system in Chapter 1: The Type System and primitive types in Chapter 2: Primitive Types, Literals, & Nullables We will need to know both of these things in this chapter, because now we will start to see ways that we can change a value from one type to another.

Sometimes we want to take an object and change its type; for example, take a value that was of type `int` and change it to a `double`, or start with an instance of `float` and turn it into a `long`. We can do this in two ways: *casting* and *conversion*.

Instances of type `string` must be treated a bit differently. We can take objects of type `string` and attempt to change their value into a different type through *parsing*.

Casting

Casting is a method by which we take an instance of an object and attempt to "force" its value into a new type. When a cast is attempted, if the value of the object is allowable in the new type, the object will be *casted* into an object of the specified type.

We cast a value by placing the targeted type in parentheses `()` next to the value we want to cast.

C#'s compiler allows many kinds of casting. For example, we can cast an `int` to a `double`, a `char` to an `int`, or a `float` to a `decimal`.

```
int five = 5;
var doubleFive = (double)five;

char a = 'a';
var valueA = (int)a;

float myFloat = 4.56F;
 decimal myMoney = (decimal)myFloat;
```

The examples above are all different forms of *implicit casting*, which lets the compiler decide exactly how to change a value from one type to another.

For each of these casts (and many others) the C# compiler will "force" the value into a new variable of the specified type. This works if the range of the new type includes the value. However, some casts will fail because the types are not compatible, such as:

```
string myString = "This is a sentence";
byte myByte = (byte)myString;
```

Error: cannot convert 'string' to 'byte'

In this example, there is no way to determine if the value of `myString` can be converted to an instance of `byte`, so the C# compiler will throw an error. Strings must be parsed, not casted.

Further, casting from a more-precise type to a less-precise type will result in a loss of precision:

```
decimal myMoney = 5.87M;
int intMoney = (int)myMoney; //Value is now 5; the .87 was lost
```

Because of this, we need to be careful when converting more-precise types (e.g. the floating-point numeric types such as `double` and `decimal`) to less-precise types like `int`, `char`, or `long`.

Conversion

A conversion is like a cast in that it takes a value of an instance and changes said value into a value of another type. However, conversions are more forgiving than casts, generally do not lose precision, and take computationally longer to execute.

The .NET Framework provides us with a class called `Convert`. This class can take values from all the primitive types and attempt to convert them to all other primitive types.

A sample code block that uses the `Convert` class is on the next page.

```
int five = 5;
decimal decFive = Convert.ToDecimal(five);
decimal myMoney = 5.67M;
int intMoney = Convert.ToInt32(myMoney); //Value is now 6;
                                         //the decimal value was rounded
```

When casting a floating-point numeric type to an integral numeric type (such as casting an instance of `double` to `int`), the numbers after the decimal point are lost. When converting, the value is instead rounded to the nearest whole number using a methodology known as "banker's rounding": if the number is exactly halfway between two whole numbers the even number is returned (e.g. if the number is 4.5, return 4; if the number is 5.5, return 6); otherwise, round to the nearest whole number.

The `Convert` class can also handle numeric to non-numeric and vice-versa conversions, such as:

```
string five = "5.0";
decimal decFive = Convert.ToDecimal(five); //Value is 5.0

double myValue = 5.33;
string stringValue = Convert.ToString(myValue); //Value is "5.33"

int intTrue = 1;
bool isTrue = Convert.ToBoolean(intTrue); //Value is true because number
is not 0
```

In general, casting is faster but more prone to errors, and conversion is slower but more likely to succeed. Which one you use in any given situation is a decision that is left up to the individual developers.

Parsing

As mentioned earlier, the `string` type has a unique place among the C# primitive types. Because it is a reference type, it needs special handling when converting from it to other types. We call this parsing.

The .NET Framework provides us with `Parse()` and `TryParse()` methods on each primitive type to handle converting from a `string` to that type.

Parse()

For example, if we needed to parse an instance of type `string` to a `decimal`, we could use the `Parse()` method:

```
string decString = "5.632";
decimal decValue = decimal.Parse(decString); //Value is 5.632M
```

However, if the `string` cannot be parsed to an acceptable value for the target type, the `Parse()` method will throw an exception:

```
string testString = "10.22.2000";
double decValue = double.Parse(testString); //Exception thrown here!

string intTest = "This is a test string";
int intValue = int.Parse(intTest); //Exception thrown here!
```

TryParse()

For situations where we don't know if the `string` value can be parsed to the desired type, we use the method `TryParse()`:

```
string value = "5.0";
decimal result;
bool isValid = decimal.TryParse(value, out result);
```

If `isValid` is `true`, then the string value was successfully parsed and is now the value of the variable `result`.

The usage of the `out` keyword is explained

Chapter 6: Methods, Parameters, and Arguments.

Checking for Type (is, as, GetType(), typeof)

There are times, when writing a C# program, that we might want to check if a given instance of an object is of a given type. C# provides us with several ways to do this.

is Keyword

There are occasions when we do not know the specific type of a given object. Very often this happens if the code retrieved the object from another source, such as an external database, API, or service. For this situation, C# provides us with the `is` keyword which tests if an object is of a specified type:

```
var myValue = 6.5M; //M literal means type will be decimal
if(myValue is decimal) { /*...*/ }
```

The `is` keyword returns `true` if the object is of the specified type, and `false` otherwise.

as Keyword

For reference types, C# provides us with the `as` keyword to convert one reference type to another.

```
string testString = "This is a test"; //string is a reference type
object objString = (object)testString; //Cast the string to an object
string test2 = objString as string; //Will convert to string successfully
```

Note that this only works on valid conversions; types which do not have a defined conversion will throw an exception:

```
public class ClassA { /*...*/ }
public class ClassB { /*...*/ }

var myClass = new ClassA();
var newClass = myClass as ClassB; //Exception thrown here!
```

Chapter 9: Inheritance and Polymorphism discusses inheritance more thoroughly.

GetType() and typeof

For any given object in C#, we can get its type as an object by calling the `GetType()` method. An example of this is on the next page.

```
var sentence = "This is a sentence.";
var type = sentence.GetType();
```

We can then check if the given type is a known type, such as a primitive, a class, or others by using the `typeof` keyword.

```
var sentence = "This is a sentence.";
var type = sentence.GetType();
if(type == typeof(string)) { /*...*/ }
else if (type == typeof(int)) { /*...*/ }
```

New Keywords

- `is` – Used to check if a value is of a given type.
- `as` – Used to convert a reference type instance from one type to another.
- `typeof` – Returns the type of a given object.

Summary

Casting and converting are ways in which we can change a value from one type to another; casting is faster but more prone to errors, while conversion is more computationally expensive but also more forgiving.

Parsing is a special form of conversion that deals with getting a value from an object of type `string` and changing that value to another type.

Using the keywords `is` and `as`, we can check for type equality between two objects and determine if one object can be changed to a different type respectively.

Finally, the special method `GetType()` and the `typeof` keyword can be used to check if objects are of a particular type.

Chapter 4: Operators

One of the most basic parts of any application are the *operators*. In C#, operators are often symbols or groups of symbols that perform some kind of operation between two or more *operands*. The operands most commonly appear on either side of the operator.

```
int total = 5 * 5;
```

The line above has two operators: the multiplication operator `*` which multiplies two operands (in this case, each operand has a value of 5), and the assignment operator `=`, which takes the result of the multiplication operation and assigns it to the variable `total`.

This chapter does not cover all possible operators in C#; some were already talked about in Chapter 3: Casting, Conversion, Parsing (`is` and `as`), some are left out of this series due to being more advanced topics (such as bitwise and shift and pointer-related operators) and some will be discussed in future articles.

Operator Structure

For the most part, operators follow this structure:

```
var value = operand1 operator operand2;
```

Let's look at some of the various operators that C# provides us with.

Assignment and Equality Operators

The assignment and equality operators are the most basic of the C# operators, and the most common; they are used to assign values to variables and to check if two objects have the same value.

Assignment (=)

The assignment operator `=` assigns a value to an object, which might be a variable instance, a property, or something else.

```
int year = 2020;
Console.WriteLine(year); //2020
```

Equality (==)

The equality operator results in a boolean value (e.g. `true` or `false`) that represents whether the two operands are equal.

```
int five = 5;
int otherFive = 5;
bool areEqual = five == otherFive;
Console.WriteLine(areEqual);
```

This operator works differently for reference types. The equality operator only returns `true` if both instances of a reference type point at the same object.

```
public class MyClass() { /*...*/ }
var myClass = new MyClass();
var myOtherClass = new MyClass();
var myThirdClass = myClass;
Console.WriteLine(myClass == myOtherClass); //false
Console.WriteLine(myOtherClass == myThirdClass); //true
```

Inequality (!=)

We can check if two values are *not* equal using the inequality operator `!=`.

```
var myMoney = 6.54M;
var theirMoney = 4.65M;

Console.WriteLine(myMoney != theirMoney); //true
```

This has the same limitations as the Equality (`==`) operator; it behaves slightly differently for reference types.

Increment Assignment Operators (+= and -=)

When dealing with number types (both integral and floating-point) we can increment or decrement a value by a specified amount using the `+=` and `-=` operators (example below):

```
var i = 16;
i += 5; //21

var j = 61;
j -= 15; //46
```

Null-Coalescing Assignment Operator (??=)

The null-coalescing assignment operator `??=` assigns the value of the right-side operand to the left-side operand if and only if the left-side operand is null.

```
int? startNum = 5;
startNum ??= 3; //startNum is 5

startNum = null;
startNum ??= 3; //startNum is 3
```

Math Operators

Also called "arithmetic operators", the math operators do math operations on their operands.

Basic Math (+, -, *, /)

The basic math operators perform simple math operations (addition, subtraction, multiplication, and division):

```
var sum = 5 + 9; //14
var difference = 56 - 14; //42
var product = 6 * 6; //36
var quotient = 42 / 7; //6
```

Note that these operators, when used on integer types, will round the result toward zero and produce a result that is an integer.

Remainder (%)

The remainder operator (also called the *modulus* operator) gives the remainder from a division operation.

```
var remainder = 43 % 5; //3
```

Increment (++) and Decrement (--)

These operators increase or decrease the value of a variable. Note that we do not need to assign the new value to a new variable; the value of the current variable is directly changed.

```
var value = 3;
value++; //4
value--; //3
```

The behavior of these operators can be changed on whether they are *postfixed* or *prefixed*. Postfixed increment and decrement operators show the value *before* the operation has occurred.

```
var value = 3;

Console.WriteLine(value); //3
Console.WriteLine(value++); //3
Console.WriteLine(value); //4
```

Whereas prefixed increment and decrement operators show the value *after* the operation occurs.

```
var value = 3;

Console.WriteLine(value); //3
Console.WriteLine(++value); //4
Console.WriteLine(value); //4
```

Order of Operations

The math operators in C# obey an order of operations. This means these operations are evaluated in this order:

- First, do increment (++) and decrement (--).
- Then do multiply (*), divide (/), and remainder (%).
- Finally, do addition (+) and subtraction (-).

Operators at the same level in the order of operations and for which there is not a clearly defined order are evaluated from left to right.

```
var output = 5 + 2 * 9; //Evaluated as 5 + (2 * 9) = 23
var output2 = (5 + 2) * 9; //63

var output3 = 15 / 5 * 3; //Evaluated as (15 / 5) * 3 = 9
var output4 = 15 / (5 * 3); //1
```

Boolean Logic Operators

When dealing with boolean values, we have a special set of operators designed to evaluate true or false statements.

A Boolean Logic Primer

Boolean operations always work on two operands. The most basic Boolean operations are as follow:

- AND: The result is `true` if and only if both operands are `true` and is `false` otherwise.
- OR: The result is `false` if and only if both operands are `false` and is `true` otherwise.
- XOR (Exclusive-OR): The result is `true` if and only if the operands have different values (e.g. one operand is `true` and the other is `false`)

There is also a basic Boolean operation that takes only one operand:

- NOT: Returns the opposite of the current value (e.g. if the value is currently `true`, NOT will return `false`).

Most Boolean logic that you will see in any given C# application flows from the usage of these three operators.

Logical Negation (!)

The negation operator `!` "flips" the boolean to its opposite value. This is, in fact, the NOT operation.

```
var isTrue = true;
Console.WriteLine(!isTrue); //false

var isFalse = false;
if(!isFalse)
{
    //These lines will execute
}
```

Conditional Logical AND (&&)

The conditional logical AND operator && evaluates boolean operands. If both operands are `true`, this operation returns `true`. If either operand is `false`, the operator returns `false`. In fact, if the first operand is `false`, the second operand is not even evaluated; this operator is called "conditional" because it only evaluates the second operand on the condition that the first operand is `true`.

```
var hasName = true;
var hasAddress = true;

var isValidCustomer = hasName && hasAddress; //true

hasName = false;
var isStillValidCustomer = hasName && hasAddress; //false
```

Conditional Logical OR (||)

The conditional logical OR operator `||` returns `true` if at least one of its operands is `true`. Similarly to the conditional logical AND operator, this operator is also conditional: if the first operand is `true`, the second is not evaluated.

```
var hasPhone = false;
var hasEmail = true;

var hasValidContactInfo = hasPhone || hasEmail; //true
var areYouSure = hasEmail || hasPhone; //true, hasPhone not evaluated
```

Logical AND (&)

The logical AND operator `&` is like the conditional logical AND but will always evaluate both operands.

```
var hasLastName = true;
var hasFirstName = false;

var hasCompleteName = hasFirstName & hasLastName; //false
```

Logical OR (|)

The logical OR operator `|` is like the conditional logical OR, but just like the logical AND, will always evaluate both operands.

```
var hasLastName = true;
var hasFirstName = false;
var hasAnyName = hasFirstName | hasLastName; //true
```

Logical Exclusive OR (^)

The logical exclusive OR operator `^` evaluates to `true` if one of its operands is `true` and the other is `false`. If both operands have the same value, the result of using exclusive-OR will be `false` (see the code block on the next page).

```
var isXOR = true ^ true; //false
isXOR = true ^ false; //true
isXOR = false ^ true; //true
isXOR = false ^ false; //false
```

Order of Operations

The Boolean logic operators execute in a defined order:

1. Logical negation (!)
2. Logical AND (&)
3. Logical exclusive OR (^)
4. Logical OR (|)
5. Conditional logical AND (&&)
6. Conditional logical OR (||)

This means that we need to be careful when using multiple Boolean logic operators so that we end up with what we expect.

```
var isTest = true ^ false & true; //true ^ (false & true), result true
var isOtherTest = false || (false ^ true && true); //true
```

Comparison Operators

The comparison operators check if their operands match certain conditions. Each returns `true` if the condition is true, and `false` otherwise.

Less Than (<) and Less Than Or Equal (<=)

```
bool isLessThan = 7 < 9; //true
isLessThan = 9 < 7; //false

int otherValue = 6;
otherValue++;
bool isLessThanOrEqual = otherValue <= 7; //true
```

Greater Than (>) and Greater Than or Equal (>=)

```
bool isGreaterThan = 18 > 15; //true
isGreaterThan = -7 > -10; //true

int testValue = 88;
bool isLarge = testValue >= 50; //true
```

Other Operators

C# includes many more operators, but we're going to focus on two that will most likely see widespread usage in your C# applications: the condition operator (`?:`) and the null-coalescing operator (`??`).

Conditional Operator (?:)

The conditional operator evaluates a boolean target expression and returns the result of one of two other expressions.

```
var isTrue = true;
string message = isTrue ? "Yay!" : "Boo..."; //message is "Yay!"

var isFalse = false;
string message = isFalse ? "Yay!" : "Boo..."; //message is "Boo..."
```

We will discuss the exact meaning of the word "expression" in Chapter 18: Expressions, Lambdas, and Delegates.

It is possible to combine this operator multiple times with itself, though this often results in difficult-to-read code, such as the code block on the next page.

```
var hasName = false;
var hasAddress = false;
var hasPhone = true;

var message = hasName ? "Welcome!"
              : hasAddress ? "Hiya!"
              : hasPhone ? "Thanks for calling!" : "Whoops.";
```

Null-Coalescing Operator (??)

The null-coalescing operator returns the value of the left-hand operand if that value is not `null`; otherwise, it evaluates the right-hand operand and returns that.

```
int? val = null;
int counter = val ?? 1; //counter is now 1, because val is null

int? otherVal = 5;
counter = otherVal ?? 1; //counter is now 5, because otherVal is not null
```

Note that the left-hand operand (`val` and `otherVal` in the example code) cannot be a non-nullable value type.

Summary

C# includes a wide variety of operators, such as:

- Assignment operators (=, !=, +=, -=, ??=)
- Math operators (+, -, *, /, %, ++, --)
- Boolean logic operators (!, &&, ||, &, |, ^)
- Comparison operators (<, <=, >, >=)
- Other operators (?:, ??)

These operators perform a wide variety of functions, enabling us to combine them to solve complicated problems with just a few operators.

Combining operators, particularly the math operators, means we must be aware of the order of operations for these operators so that we produce the result we desire.

Chapter 5: Code Blocks, Basic Statements, and Loops

As we continue learning about C# and how we can write applications with it, we have now come to a point where, with our foundation knowledge down, we can start building simple programs.

Next, we need to discuss the flow of control in a C# program, and specifically how we as developers can write the building blocks of almost any C# application. This means we will answer the question, "how does the code know what to do next?"

Our C# programs know what lines of code to execute next using two sets of keywords: the selection statement keywords, and the loop keywords. Each of these uses a construct called a “code block” to specify what lines of code need to run.

Code Blocks

In C#, a *code block* is a group of lines of code between curly braces { }.

```
{
    //Everything between { and } is part of this code block.
}
```

Both selection statement keywords and loops work with code blocks, though in different ways.

Selection Statements (if, else, switch, case)

These keywords cause "decision points" in C# programs, where the program may or may not execute a code block based on whether a condition or set of conditions is `true`.

In C#, this set of keywords consists of `if`, `else`, `switch`, `case`, and `break`. Let's take a closer look at each of these keywords.

If, Else, If Else

The most basic of the selection statement keywords are `if` and `else`. We use these keywords to evaluate boolean expressions and direct the program to execute specified lines of code if certain expressions are `true`.

```
if (expression) //If this expression evaluates to true...
{
    //...Execute this
}
else
{
    //Otherwise, execute this
}
```

We can also use an `else if` clause to add more conditions to our evaluation.

```
decimal money;
decimal orderTotal;
if (money > orderTotal)
{
    Console.WriteLine("Thanks for your purchase!");
}
else if (money == orderTotal)
{
    Console.WriteLine("Wow! Thanks for having exact change!");
}
else
{
    Console.WriteLine("Sorry, you don't have enough money.");
}
```

An `if` statement can be implemented alone. `if else` statements and `else` statements must follow an `if` statement. We can have as many `if else` statements as we like in each set.

We can also nest `if` and `else` statements to form more complex decisions (see the code block on the next page):

```
if(expression1)
{
    if(expression2)
    {
        if(expression3)
        {
            /*...*/
        }
        else if(expression4)
        {
            /*...*/
        }
        else
        {
            /*...*/
        }
    }
    else
    {
        if(expression5)
        {
            /*...*/
        }
        /*...*/ //These lines of code will be executed
                 //whenever the containing else statement is executed.
    }
}
```

Switch, Case, Break

Sometimes we want to evaluate a given object against a large set of possible values. For these situations, we use the keyword `switch` and its related keywords `case` and `break`.

A `switch` statement is used when a single object needs to be evaluated against several potential values. Each of these possible values is identified by using a `case` statement. The `case` statement can consist of multiple lines of code and ends when the system encounters the `break` keyword. A sample using this syntax is on the next page.

```
switch(variable)
{
    case value1:
        executeThis();
        executeThat();
         break;
    case value2:
        executeThisOtherThing();
        break;
    default:
        break;
}
```

For example, let's imagine we have a company that is sponsoring us for a tournament of some kind. We want to display the company a message, but we need it to be a different message for each level of sponsorship that we offer. We might implement a `switch` statement to output the correct message for each sponsorship level:

```
var sponsorLevel = 2;
switch (sponsorLevel)
{
    case 1: //Gold
        Console.WriteLine("Thanks for being a gold sponsor!");
        break;
    case 2: //Silver
        Console.WriteLine("Thanks for being a silver sponsor!");
        break;
    case 3: //Bronze Level 2
    case 4: //Bronze Level 1
        Console.WriteLine("Thank you for being a bronze sponsor!");
        break;
    default: //All others
        Console.WriteLine("Thank you for sponsoring us!");
        Console.WriteLine("Would you like to upgrade?");
        break;
}
```

Note that we can have multiple `case` statements use the same block of code by "stacking" them, like case 3 and case 4 in the example above.

Switch Expression

When we need a switch statement to only produce a concrete value, we can use a switch expression.

For example, let's say we have a set of cards. Each card has a resource type, and a color. The cards with a given resource type always have the same color.

We can build enumerations, one for `CardColor`, and one for `CardType`.

```
public enum CardColor
{
    Brown,
    Grey,
    Blue,
    Green,
    Yellow,
    Red,
    Purple
}

public enum CardType
{
    NaturalResource,
    ManufacturedResource,
    Cultural,
    Science,
    Economic,
    Military,
    Guilds
}
```

Because the color and resource type are enumerations (which we will discuss in Enumerations), we can use a switch expression to give us the color if we have the resource type. The switch expression is on the following page.

```
public static void Main()
{
    var cardType = GetCardType(cardID); //Method not defined here
    var cardColor = cardType switch //Switch expression
    {
        CardType.NaturalResource => CardColor.Brown,
        CardType.ManufacturedResource => CardColor.Grey,
        CardType.Cultural => CardColor.Blue,
        CardType.Science => CardColor.Green,
        CardType.Economic => CardColor.Yellow,
        CardType.Military => CardColor.Red,
        CardType.Guilds => CardColor.Purple
    };
}
```

Extra special bonus points to whoever knows what this is from.

In short, "decision point" statements like `if`, `else`, and `switch` help our applications execute different code blocks depending on specified conditions.

Loops (for, foreach, while, do while)

In C#, *loops* are code blocks that are executed multiple times. The exact number of times they are executed can differ, or be dependent on a variable, or on a collection of objects.

There are four ways to implement a loop in C#, and they each have a distinct use.

For Loop

A `for` loop is a loop that executes once for each value in a given range. The loop must define a variable (commonly named `i` or `j`, termed the *initializer*); a *condition* where the loop will execute again so long as the condition is `true`; and an *iterator* which defines by how much the initializer will change after every loop.

A common kind of `for` loop uses integers and a simple increment. The sample for a `for` loop is on the following page.

```
for(int i = 0; //Initializer
    i < 10; //Condition
    i++) //Iterator
{
    //This code will execute ten times,
    //one each for i = 0, i = 1, up to and including i = 9.
}
```

We can also use increments of values other than 1:

```
for (int i = 0; i < 10; i += 2) //+= is Increment Assignment Operator
{
    //This code will execute five times.
    //i = 0, i = 2, i = 4, i = 6, and i = 8
}
```

`for` loops are particularly useful if we know the range of values that we want to loop against in advance.

Foreach Loop

When dealing with collections of objects (such as an array or a `List<T>`; both of which are discussed in Chapter 13: Arrays and Collections), we can use a `foreach` loop to iterate over every object in the collection. In this case, the iterator object is of the same type as the objects in the collection.

```
var items = new int[] {4, 5, 6, 7, 8};

foreach(int item in items)
{
    Console.WriteLine(item);
}

//Example class
public class Drawing
{
    public string Name { get; set; }
}
```

```
//Make a collection of Drawings
List<Drawing> drawings = new List<Drawing>()
{
    new Drawing()
    {
        Name = "Test Drawing 1"
    },
    new Drawing()
    {
        Name = "Test Drawing 2"
    }
};

//Iterate over each drawing in the collection
foreach (Drawing iterator in drawings) //iterator is of type Drawing
{
    Console.WriteLine(iterator.Name);
}
```

Using a `foreach` loop to iterate over a collection is the most common scenario for this kind of loop.

While Loop

A `while` loop evaluates a condition, and so long as that condition is `true`, the loop will keep executing.

```
int myVal = 0;
while(myVal < 1000) //This loop will execute 1000 times...
{
    DoSomething();
    DoSomethingElse();
    myVal = myVal + 1; //...because each time through the loop, we
                       //increase the value of myVal by 1
}
```

We must be careful when writing `while` loops; it is easy to accidentally create a loop that will never reach the given end condition. Such a situation is shown in the code block on the next page.

```
int myVal = 5;
while(myVal < 1000)
{
    DoSomething();
    DoSomethingElse(); //We didn't increment myVal,
                       //so this loop will never stop executing!
}
```

Because a `while` loop evaluates the condition before entering the loop, if that condition is `false` before the loop starts, the loop will not be executed.

Do While Loop

In contract to the `while` loop, a `do while` loop will always execute at least once, because the condition is evaluated at the *end* of the loop.

```
int myVal = 1;
do
{
    DoSomething();
    DoSomethingElse();
    myVal++;
} while (myVal < 1000);
```

If you must guarantee that the code in a loop run at least one time, use a `do while` loop.

Breaking the Loop (break, continue, return)

In many situations, we may want to stop executing the loop before the loop reaches its end condition. There are several keywords we can use for these types of conditions.

Break

The `break` keyword ends execution of the loop. No further iterations of the loop will execute.

```
for(int i = 0; i < 10; i++)
{
    if(i == 7)
    {
        break; //Will exit the for loop
    }
}
```

You might have noticed that this keyword was also used in the `switch` statement examples earlier in this chapter, and its function was similar.

Continue

The `continue` keyword ends execution of the *current* iteration of the loop but will restart the loop at the *next* iteration.

```
int myVal = 5;
while (myVal < 10)
{
    if (myVal == 7)
    {
        myVal++;
        continue; //If i == 7, processing stops here and resumes
                  //at the top of the loop with the next value 8.
                  //Console.WriteLine is never called in that case.
    }
    //The below output will not happen when myVal = 7
    Console.WriteLine("The current value of myVal is " +
myVal.ToString());
    myVal++;
}
```

Return

The `return` keyword, similarly to how it works in methods, will return a value to the calling code. The loop will therefore stop executing.

```
var emails = GetEmails(); //Method not defined here
foreach(var email in emails)
{
    if(email.Sender == "mybeloved@email.com")
    {
        return email; //Loop stops executing
    }
}
```

Summary

In C#, we control the flow of execution in our programs by using "decision point" statements such as `if`, `else`, and `switch` and by creating one of the four kinds of loops: `for`, `foreach`, `while`, and `do while`. We can also stop or modify execution of those loops using the `break`, `continue`, and `return` keywords.

Chapter 6: Methods, Parameters, and Arguments

As we build up to being able to write full C# programs, we must now "zoom out" a little bit further from mere code blocks and loops and discuss higher-level constructs such as methods. We will also talk about parameters and arguments.

Methods

A *method* in C# is a code block which takes inputs and optionally returns an output. A method may also be called a *function*.

Methods have five parts:

- An *access modifier* (e.g. `public`, `private`, etc.)
- A *return type*
- A name.
- An optional set of *parameters*.
- A collection of code statements bounded by curly braces { }, AKA a code block.

Here's a basic method:

```
public string GetHello(string name)
{
    return "Hello " + name + "!";
}
```

In this method, the access modifier is `public`, the return type is `string`, the name is `GetHello`, and there is one parameter, which has the type `string` and the name `name`.

Method Invocation

A method is "invoked" when it is called to run by another part of the code. For example, we can invoke the `GetHello()` method we wrote like so:

```
string name = "Jen";
var result = GetHello(name); //Method invocation AKA method call
Console.WriteLine(result); //Output: "Hello Jen!"
```

Access Modifiers

Methods can have one of six access modifiers: `public`, `private`, `protected`, `internal`, `protected internal`, and `private protected`. Each of these access modifiers restricts what other code can invoke this method in a different way.

For simplicity, the sample code in this chapter will only have `public` and `private` methods.

Return Types and Void

Methods can return any C# type, whether that is a primitive, class, struct, enumeration, etc.

Methods can also use the special return type `void`, which tells the C# compiler that the method will not return anything.

```
public string GetHello()
{
    return "Hello!";
}

public void DoSomething() { /*...*/}

string hello = GetHello();
DoSomething();
```

If our method has a return type but does not include the `return` keyword, we will get a compilation error.

```
public string GetGoodbye()
{
    string goodbye = "Goodbye!";
}
```

Error: not all code paths return a value.

Similarly, if our method has the return type `void` but attempts to return a value, we will get a different compilation error.

```
public void DoSomething()
{
    return "Something!";
}
```

Error: Since "DoSomething()" returns void, a return keyword must not be followed by an object expression.

Naming

The names of C# methods use *Pascal Casing* by convention; the first letter of every word in the method name should be capitalized. Note that you are not required to use Pascal Casing, but any method which does not use it will be seen as not conforming to standards.

Coming up with a good name for a method is a tricky subject; there are entire blog posts out there just for naming things. One tip we can follow is this: try to make it clear in the name what the method does and needs, without using too many words. The idea behind good naming is to communicate as much of the method's intent (or reason to exist) as possible.

Parameters and Arguments

Methods can optionally define a set of parameters they can accept. A single method can take any number of parameters, and each parameter needs a type and a name included in the method definition.

The concrete values given to the method invocation are called *arguments*. An example of how to use arguments is on the following page.

```
public int Add(int param1, int param2) //param1 and param2 are parameters
{
    return param1 + param2;
}

int value1 = 5; //Used as arguments below
int value2 = 16;
int sum = Add(value1, value2); //Sum == 21
```

Out Keyword

Occasionally we want to pass values to a method by reference rather than by value. We can do this using the `out` keyword in the method definition. This keyword is often used to allow the method to "return" more than one value.

```
string intString = "5";
int result;
bool hasValue = int.TryParse(intString, out result); //Result is now 5
```

C# does this itself with the TryParse methods, which we saw back in TryParse().

`out` parameters may only be modified by the method they are passed to.

Ref Keyword

We can also pass parameters by reference using the `ref` keyword:

```
public void Sum(ref int total, int second)
{
    total = total + second;
}

int total = 17;
int nextNumber = 2;
Sum(ref total, nextNumber); //total is now 19
Sum(ref total, 6); //total is now 25
```

Unlike `out` parameters, `ref` parameters must be initialized (given a value) before they can be passed to a method.

In Keyword

We might also want to pass a value by reference but not allow the method to modify that value; in effect, this makes the value read-only. For this, we can use the `in` keyword.

```
double pi = 3.14159;

public double GetCircumference(double radius, in double pi)
{
    //pi = 3.14285; //Uncomment this line to get an error.
    return 2 * pi * radius;
}
```

Params Keyword

It is possible to pass an arbitrary number of parameters to a method, provided they are all of the same type. The `params` keyword allows a method to take a group of parameters and automatically create an array of the given values:

```
public decimal GetTotalPriceForSeats(params decimal[] seatPrices)
{
    return seatPrices.Sum(); //Discussed in Chapter 14 - LINQ Basics
}

var totalPrice = GetTotalPriceForSeats(90, 91, 92, 93, 94);
```

If a method uses a `params` parameter, it must be the last parameter listed in the method definition.

Optional Parameters

Methods in C# might have some of their parameters be optional. When such a method is invoked, the invocation does not need to pass an argument for optional parameters; instead, a default argument can be used.

To do this, we specify the default argument for the parameter in the method definition. An example of how to use an optional parameter is on the next page.

```
public int Increment(int startValue, int increment = 1) //Increment has
                                                        //a default value
{
    return startValue + increment;
}

int byOne = Increment(10); //11, because the default increment 1 is used
int byFive = Increment(10, 5); //15
```

Please note that if a method has optional parameters, they must appear in the parameters list *after* all the required parameters.

Named Arguments

We can even pass arguments to a method out of order, if we know the parameter names.

```
public double GetPyramidVolume(int height, int baseArea)
{
    return 0.33333333333 * height * baseArea;
}

var volume = GetPyramidVolume(baseArea: 12, height: 4);
```

Summary

A method in C# is a code block that is executed when said method is invoked (called by another part of the code). A method must specify the parameters they can take, the type they return, their access modifier, and the lines of code they will execute.

Arguments (concrete values) are passed into method calls for each parameter in order to invoke the method. Many types of arguments can be used, including reference arguments, optional arguments, and named arguments.

Chapter 7: Classes and Members

The defining characteristic of C# programs, indeed, of all object-oriented programming languages, is that they support classes. We now have enough background knowledge to dive into what classes are, what they contain, and how they work.

Classes

A *class* in C# is a definition that can be used to create instances of that class. Classes are created using the `class` keyword. C# class instances are always reference types.

```
class TestClass { //An empty C# class }
```

By convention, C# classes use *Pascal Naming*; each word in the class name should have their first letter capitalized.

Members

Classes can have *members*. A class can have many kinds of members, including:

- *Fields* - Members of the class that hold values and must be accessed directly. Each of these will have a type.
- *Properties* - Members of the class which provides a way to read, write, or change a value held by a field.
- *Methods* - A code block containing a set of statements that will be executed. Methods may or may not return a value of a given type.
- *Constructors* - A special kind of method which sets the initial values for the properties in the class.

Class Instances

We create an instance of a class using the `new` keyword:

```
var myInstance = new TestClass();
```

We can then manipulate the instance by changing its property values, invoking methods or constructors, etc.

Class Fields

When writing a C# class, we define *fields* in the class by giving them a type.

```
class ExampleClass
{
    string Property1; //Default null
    int Property2; //Default 0
    decimal Property3; //Default 0.0
}
```

When an instance of a class is created, each of the class's fields will get their default value.

```
var myClass = new ExampleClass();
Console.WriteLine(myClass.Property1); //Null
Console.WriteLine(myClass.Property2); //0
Console.WriteLine(myClass.Property3); //0.0
```

Access Modifiers

Each field or property in a C# class can be declared with a *access modifier*, which is a keyword that specifies who or what can access that field or property.

- `private` - Property is only accessible inside of instances of this class.
- `protected` - Property is accessible by instances of this class AND instances of classes which inherit from this class.
- `public` - Property is accessible from any code in any assembly which references the assembly for this class.
- `internal` - Property is accessible from any code in the same assembly only.

By default, all fields and properties declared without an access modifier are `private`. We can set the access modifier on on the fields we defined earlier like so:

```
class ExampleClass
{
    public string Property1;
    protected int Property2;
    private decimal Property3;
}
```

Class Properties

Fields are direct members of the class and must be accessed directly. This is a bad idea in most cases; generally, we want to use *properties* rather than fields.

Properties provide a way to modify and access a value in the class. They are backed by a field, which holds said value, and use access modifiers.

Getter and Setter Methods

For `public` or `protected` properties we can declare *getter and setter methods*, which often use the syntax `{ get; set; }`:

```
class ExampleClass
{
    public string Property1 { get; set; }
    protected int Property2 { get; set; }
    private decimal Property3;
}
```

To be clear: a property has getter and/or setter methods, and a field has neither.

If we want to allow other classes or objects to access a private field (such as `Property3` in the example above), we can declare a public property and modify its getter and setter methods to read and change the field's value. An example of how to do so is on the following page.

```
class ExampleClass
{
    public string Property1 { get; set; }
    protected int Property2 { get; set; }
    private decimal Property3;
    public decimal PublicProperty3
    {
        get
        {
            return Property3;
        }
        set
        {
            Property3 = value;
        }
    }
}
```

In this case, the property `PublicProperty3` stores and retrieves values from the field `Property3`.

Auto-Implemented Properties

In most cases, if we want the property to be modifiable outside of the class it is declared in, declaring a public property with getter and setter methods is the way to go. Behind the scenes, doing this creates a private field and get/set methods for the property. Plus, it makes our code much cleaner:

```
class ExampleClass
{
    public string Property1 { get; set; }
    protected int Property2 { get; set; }
    public decimal Property3 { get; set; }
}
```

Properties written this way are called *auto-implemented properties*. Most of the time, when you are writing a program in C#, your classes will use auto-implemented properties.

Calculated Properties

We can also write *calculated properties* which use the values of other properties in the same class to return a value of their own. Calculated properties will only have a getter method:

```
class ExampleClass
{
    public string ItemName { get; set; }
    protected int Quantity { get; set; }
    public decimal ItemPrice { get; set; }
    public decimal TotalCost
    {
        get
        {
            return Quantity * ItemPrice;
        }
    }
}
```

Class Access Modifiers

Just like class properties, C# classes themselves can use access modifiers.

```
public class PublicClass { /*...*/ }
protected class ProtectedClass { /*...*/ }
private class PrivateClass { /*...*/ }
class InternalClass { /*...*/ } //Default access modifier is internal
```

Classes which are declared directly within a namespace may be either `public` or `internal`; `internal` is the default. Namespaces will be discussed in Chapter 11: Namespaces.

In general, a derived class (that is, a class which inherits from another class) cannot have greater accessibility than its parent. We cannot, for example, do this:

```
internal class ClassA { /*...*/ }
public class ClassB : ClassA { /*...*/ }
```

Error: Inconsistent accessibility: base class ClassA is less accessible than class ClassB

However, a `public` class can inherit from a `private` class. The derived class must at least as accessible as the parent class.

Methods & Constructors

A C# class can have methods. Just like properties, methods have access modifiers, and their default access modifier is `private`.

```
public class ClassC
{
    public string Property1 { get; set; }
    public void Method1()
    {
        /*...*/
    }
    public string MethodWithReturn()
    {
        /*...*/
        return "stringValue";
    }
}
```

Methods in a class may also have input parameters:

```
public class ClassD
{
    public string MethodWithParameters(int param1,
                                       string param2,
                                       decimal param3)
    {
        /*...*/
        return "stringValue";
    }
}
```

Methods in classes are subject to the same rules as methods in general; we covered how methods work in Chapter 6: Methods, Parameters, and Arguments.

Constructors

A C# class will always have at least one method which is a *constructor*; this method is responsible for assigning the default values for the properties of the class when an instance of it is created. The constructor method must have the same name as the class and no return type.

```
public class ClassE
{
    public ClassE() { /* Constructor Method */ }
}
```

We can specify input parameters to a constructor, just like any other method. Further, our classes can have multiple constructors, if each constructor has a different set of input parameters.

```
public class ClassF
{
    public string Property1 { get; set; }
    public int Property2 { get; set; }
    public ClassF(int param1)
    {
        Property2 = param1;
    }
    public ClassF(string param1)
    {
        Property1 = param1;
    }
    public ClassF(string param1, int param2)
    {
        Property1 = param1;
        Property2 = param2;
    }
}

var myClassF = new ClassF("stringValue");
var otherClassF = new ClassF("stringValue", 6);
```

Implicit Constructors

If we do not specify a constructor method, the C# compiler will create an *implicit constructor*, which will be invoked whenever we create a new instance of a class and don't pass any parameters to the constructor. This kind of constructor is also called a *public parameterless constructor*, because it will be public and have no parameters.

```
public class ClassG
{
    public string Property1 { get; set; }
    public int Property2 { get; set; }
}

var myClassG = new ClassG();
```

Of course, we can also create a public parameterless constructor:

```
public class ClassH
{
    public string Property1 { get; set; }
    public int Property2 { get; set; }
    public ClassH() { /* Public parameterless constructor */ }
}

var myClassH = new ClassH();
```

Summary

Classes in C# are collections of properties, methods, constructors, and other members; instances of a class are created using these definitions and the `new` keyword. Instances of classes are always reference types.

C# classes support many kinds of properties, including auto-implemented properties, calculated properties, and private properties. Properties have `get` and `set` methods which allow us to retrieve and modify the value of the property; properties which do not have getter or setter methods are termed *fields*, and directly modifying field values is often (though not always) bad practice.

Access modifiers (e.g. `public`, `private`, `protected`, etc.) allow programmers to specify the level of access for properties, methods, and the classes themselves.

Methods in C# classes behave the same as methods elsewhere; they can be invoked on instances of the class.

All C# classes must have at least one constructor. This is a special method that sets the initial values for a class's properties and fields when an instance of the class is created. Constructors do not specify a return type, not even `void`. We can define our own constructors, so long as each constructor has a different set of input parameters. If no constructors are defined, C#'s compiler will ensure that the class automatically has a public parameterless constructor.

Chapter 8: Structs and Enums

In each of the previous chapters, we have been "zooming out"; that is, we've been building up to larger and more complicated ideas that C# can implement. In this chapter, we're going to take a break from zooming out and learn about two special value types that C# provides us: structs and enums.

Structs

A *structure type* (or *struct*) is a C# type that, like classes, encapsulates data and functionality. We use the `struct` keyword to define a struct.

Like classes, structs can have methods, constructors, and properties. However, structs are always value types, while classes are always reference types.

```
public struct Player
{
    public string Name { get; set; }
    public int YearsPlaying { get; set; }
    public Player(string name, int yearsPlaying)
    {
        Name = name;
        YearsPlaying = yearsPlaying;
    }
}

var player = new Player();
player.Name = "Alex Hampton";
player.YearsPlaying = 2;
```

While similar to classes, structs have some very important differences:

- Structs cannot have a parameterless constructor, AND
- Structs do not support inheritance. We will discuss inheritance more thoroughly in Chapter 9: Inheritance and Polymorphism.

Typically, we use structs to define small, data-only objects with little or no behavior. To demonstrate this, we need only look at several of the primitive types we are

already familiar with. The keywords for many of the primitive types are just shortcuts to defined types that are created as structs.

```
int five = 5; //Actually System.Int32
decimal money = 12.4M; //Actually System.Decimal
bool isTrue = true; //Actually System.Boolean
```

Instantiating Structs

Because structs are value types, when we create an instance of a struct we must pass to it values for each of its properties. We can do this in three ways.

In the first way, we rely on the default values of the properties in the struct.

```
public struct Coach
{
    public string Name { get; set; }
    public int YearsCoaching { get; set; }
}

var coach = new Coach();
Console.WriteLine(coach.Name); //Null
Console.WriteLine(coach.YearsCoaching); //0
```

In the second way, we pass values for each public property during instantiation:

```
public struct Coach
{
    public string Name { get; set; }
    public int YearsCoaching { get; set; }
}

var coach2 = new Coach() //This is called inline instantiation
{
    Name = "John Smith",
    YearsCoaching = 12
};
Console.WriteLine(coach2.Name);
Console.WriteLine(coach2.YearsCoaching);
```

In the third way, we use a constructor method on the struct to give default values.

```
public struct Coach
{
    public string Name { get; set; }
    public int YearsCoaching { get; set; }
    public Coach (string name, int years)
    {
        Name = name;
        YearsCoaching = years;
    }
}

var coach3 = new Coach("Elaine Harkness", 6);
Console.WriteLine(coach3.Name); //Elaine Harkness
Console.WriteLine(coach3.YearsCoaching); //6
```

Readonly Structs

Due to restrictions on struct usage, Microsoft recommends we implement *immutable* structs in most cases. An immutable struct is one whose values can only be set at instantiation. We make a struct immutable by declaring it with the `readonly` keyword.

```
public readonly struct ReadonlyPlayer
{
    public string Name { get; } //No setter methods!
    public int TurnOrder { get; }
    public ReadonlyPlayer(string name, int turnOrder)
    {
        Name = name;
        TurnOrder = turnOrder;
    }
}

var readonlyPlayer = new ReadonlyPlayer("Matt", 1);
```

Note that because this struct is immutable, its properties can no longer have setter methods. Further, we cannot change the value of the struct's properties at runtime (see the example on the next page).

```
var readonlyPlayer = new ReadonlyPlayer("Matt", 1);
readonlyPlayer.Name = "Different Name"; //Compilation error!
```

Error: Property or indexer 'Name' cannot be assigned to – it is read only.

Readonly Instance Members

We can also declare the members of a struct as `readonly`; this is commonly done with methods where the method does not change the state of the struct.

```
public readonly struct Player
{
    public string Name { get; }
    public int TurnOrder { get; }
    public Player(string name, int turnOrder)
    {
        Name = name;
        TurnOrder = turnOrder;
    }
    public readonly string GetCustomDisplay()
    {
         return Name + " will play in position #" + TurnOrder.ToString();
    }
}

var player = new Player()
{
    Name = "Matt",
    TurnOrder = 1
};
Console.WriteLine(player.GetCustomDisplay());
```

Enumerations

The other kind of special value type we will discuss are called *enumerations*.

Enumerations (or *enums*) are a set of integer values which have been assigned names. Their major purpose is to make reading code easier and to eliminate the use of *magic numbers*, or numbers whose meaning is not obvious from their value.

We create a new enum using the `enum` keyword:

```
int color = 1; //What does 1 mean?
//Instead, do this
public enum Color
{
    Red = 1,
    Yellow = 2,
    Blue = 3,
    Green = 4
}

var myColor = Color.Red;
Console.WriteLine(myColor); //Red
```

Casting to Value

We can cast enum values to their integer value, and back:

```
var myColor = Color.Blue;
var myColorValue = (int)myColor;
Console.WriteLine(myColorValue); //3
var myColor2 = 4;
var myColorEnum = (Color)myColor2;
Console.WriteLine(myColorEnum); //Green
```

Default Values

Enumerations can be assigned values like in the above example, or use the default values (which start at zero and go up by one):

```
public enum CardType
{
    Resource, //0
    Science, //1
    Economic, //2
    Military //3
}
```

By default, enums are of type `int`; we can optionally specify another integer type instead. We can also assign any value of that type we like to represent each enum value:

```
public enum HairColor : short
{
    Brown = 5,
    Blonde = 38,
    Red = 145,
    Black = 2,
    Grey = 42
}
```

Because enumerations are little more than integers with names, they cannot have methods, constructors, or other features of structs and classes.

Enumerations as Bit Flags

A neat trick of enumerations in C# is their ability to represent combinations of values; to do this, we implement a specialized kind of enumeration called a *bit flag*.

For this to work, we need two things:

1. Our enum must be decorated with the `[Flags]` attribute, AND
2. Every value in the enum must be a power of 2.

By setting this up, we enable our enum to represent several selected values, rather than just one. For example, on the next page is an example of how to set up a `Months` enumeration as a bit flag.

```
[Flags]
public enum Months
{
    January = 1, //2^0
    February = 2, //2^1
    March = 4, //2^2
    April = 8, //2^3
    May = 16, //2^4
    June = 32, //2^5
    July = 64, //2^6
    August = 128, //2^7
    September = 256, //2^8
    October = 512, //2^9
    November = 1024, //2^10
    December = 2048 //2^11
}
```

If we want to represent multiple values, we can use the logical OR operator (|), which we previously discussed Logical OR (||).

```
Months birthdayMonths = Months.January
                      | Months.March
                      | Months.September
                      | Months.November;

Console.WriteLine($"Your family has birthdays in {birthdayMonths}");
//Output: Your family has birthdays in January, March, September, November
```

We can also use the Logical AND (&) to get the intersection of two groups (that is, values which appear in both sets).

```
Months birthdayMonths = Months.January
                      | Months.March
                      | Months.September
                      | Months.November;

Months otherBirthdays = Months.January
                      | Months.April
                      | Months.September;
Console.WriteLine($"The months in both groups are {birthdayMonths &
otherBirthdays}");
//Output: The months in both groups are January, September
```

Summary

Structs and enums are both specialized value types in C#.

Structs allow us to define small, encapsulated values and pass them around as a group. They can have constructors, methods, and properties. Generally, we use structs for objects that have little to no behavior. It is recommended that we create immutable structs, which means we must assign values to a struct instance when it is instantiated.

Enums are integer values that have been given a name; their primary purpose is to make reading our code a bit easier. We can use any integer type to represent the value of an enum, and `int` is the default. When creating an enum, we can either specify the integer value for each name or let C#'s compiler assign default values. Finally, we can use a specialized kind of enumeration called a bit flag to represent combinations of values.

Chapter 9: Inheritance and Polymorphism

Now that we've discussed most of the basic ideas we need for a C# program, let's talk about two concepts that are central to how C# (and indeed, all object-oriented programming languages) work: *inheritance* and *polymorphism*.

Inheritance

Inheritance allows a class to reuse the properties, methods, and behavior of another class, and to extend or modify that behavior.

The class which implements the original properties or methods and will be inherited from is called the *base class*; the class which inherits from the base class is called the *derived class*. A derived class inherits from a base class.

"Is A" and "Is A Kind Of"

When talking about inheritance, we normally think of the derived classes having an "is a" or "is a kind of" relationship with the base class.

For example, a bee is an insect, a Toyota Corolla is a car, and a dresser is a kind of furniture. In these examples, Insect, Car, and Furniture are the base classes, while Bee, Toyota Corolla, and Dresser are the derived classes.

```
public class Insect { /*...*/ }
public class Bee : Insect { /*...*/ }

public class Car { /*...*/ }
public class ToyotaCorolla : Car { /*...*/ }

public class Furniture { /*...*/ }
public class Dresser : Furniture { /*...*/ }
```

In C#, we specify that an object inherits from another object using the `:` operator, as shown above.

You can have multiple distinct classes inherit from a common base class (this is, in fact, a defining trait of inheritance). In the code on the following page, the derived classes `Dog` and `Worf` both inherit from base class `Animal`.

```
public class Animal //Base class
{
    public string SpeciesName { get; set; }
    public bool IsDomesticated { get; set; }
    public virtual void MakeSound()
    {
        Console.WriteLine("Basic Animal Sound");
    }
}

public class Dog : Animal //Derived class
{
    public string BreedName { get; set; }
    public Dog(string breedName)
    {
        SpeciesName = "Canis familiaris";
        IsDomesticated = true;
        BreedName = breedName;
    }
    public override void MakeSound()
    {
         Console.WriteLine("Bark!");
    }
}

public class Wolf : Animal //Derived class
{
    public Wolf()
    {
        SpeciesName = "Canis lupus";
        IsDomesticated = false;
    }
    public override void MakeSound()
    {
        Console.WriteLine("Awooooooooo!");
    }
}
```

Elsewhere in our code, we can create instances of `Dog` and `Wolf` and call their respective `MakeSound()` methods.

```
Dog dog = new Dog("Labrador Retriever");
dog.MakeSound();

Wolf wolf = new Wolf();
wolf.MakeSound();
```

Access Modifiers

Properties and methods in the base class have access modifiers that change whether the derived classes can use them.

- `private` members are not visible by derived classes.
- `internal` members are only visible by derived classes if they are in the same assembly as the base class.
- `protected` members are ONLY visible in derived classes.
- `public` members are visible to all code.

An example of using access modifiers for properties in a class is on the following page.

```
public class Animal2
{
    private string SpeciesName;
    protected bool IsDomesticated { get; set; }
    public bool IsExtinct { get; set; }
}

public class Dodo : Animal2
{
    //We cannot access SpeciesName here, because it is private
    public Dodo()
    {
        IsDomesticated = false,
        IsExtinct = true
    }
}

var dodo = new Dodo();
dodo.IsDomesticated = true; //COMPILATION ERROR; IsDomesticated is
protected
dodo.IsExtinct = true;
```

Remember that the term "member" can refer to a property, method, constructor, etc.

Overriding Methods

Derived classes can also change the implementation of the base class's methods. The derived class's implementation for said methods is used in place of the original implementation, which will not be executed.

In order to do this, the method in the base class must be marked with the `virtual` keyword, and the methods in the derived class must have the same name and parameters and be marked with the `override` keyword.

An example of overriding a virtual method is on the next page.

```
public class Animal
{
    public virtual void MakeSound() { /*...*/ }
}

public class Tiger : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Roar!");
    }
}

public class Hobbes : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Denial springs eternal.");
    }
}
```

Base Constructors

If the derived class's constructor needs to call a constructor in the base class, they can do so using the `base` keyword.

```
public class Animal
{
    public string SpeciesName { get; set; }
    public Animal(string speciesName)
    {
        SpeciesName = speciesName;
    }
}

public class BlueWhale : Animal
{
    public BlueWhale() : base("Balaenoptera musculus")
    { /*...*/ }
}
```

Implicit Inheritance

We know from all the way back in Chapter 1 System.Object `System.Object`. Due to inheritance, this also means that all objects ever instantiated in C# can all use methods defined in `System.Object`.

Let's define an new, empty `Vegetable` class and then create an instance of it to show what we mean:

```
public class Vegetable { }

Vegetable myVegetable = new Vegetable();
var stringDescription = myVegetable.ToString(); //Method implemented in
System.Object
var type = myVegetable.GetType(); //Method implemented in System.Object
```

Because all objects in C# must have a type, and all objects in C# inherit from `System.Object`, the class `Vegetable` will be able to use methods defined on `System.Object`. This is called *implicit inheritance*.

All objects in C# can implement these common methods, including `ToString()`, `GetType()`, `Equals()`, `GetHashCode()`, `MemberwiseClone()` and more!

No Multiple Inheritance!

C# does not permit *multiple inheritance*; a single C# class cannot inherit from multiple other classes. There are other ways to inherit behavior, though, and the next chapter will cover two of them: interfaces and abstract classes.

Polymorphism

Polymorphism, along with encapsulation and inheritance, are the three defining characteristics of object-oriented programming.

In short, polymorphism in C# means we can treat instances of a derived class as though they are instances of their base class. For example, on the next page we define a base class and two derived classes, as well as a method which takes an instance of the base class as a parameter.

```
public class Animal //Base class
{
    public string SpeciesName { get; set; }
    public string CommonName { get; set; }
}

public class Elephant : Animal { /*...*/ } //Derived class

public class Porcupine : Animal //Derived class
{
    public string OrderType { get; set; } //Old world or New world
}

public static class StaticMethods
{
    public static string GetAnimalDetails(Animal animal)
    {
        return animal.SpeciesName + "(AKA " + animal.CommonName + ")";
    }
}
```

If we instantiate an `Elephant` object and a `Porcupine` object, we can pass both of them to the `GetAnimalDetails()` method and the method will treat each as though they are instances of the `Animal` class:

```
var porcupine = new Porcupine();
porcupine.OrderType = "New World";

var details = GetAnimalDetails(porcupine);
```

However, the method `GetAnimalDetails()` will not be able to access the property `OrderType`, because that property is defined in the derived class `Porcupine` and not in the base class `Animal`.

```
public void GetAnimalDetails(Animal animal)
{
    string type = animal.OrderType; //COMPILATION ERROR
    return;
}
```

Virtual Methods

We can work with polymorphism by implementing a C# feature that we saw in the Inheritance examples: *virtual methods*. These are methods defined on the base class that allow for the derived classes to implement some additional behavior, and optionally run the behavior defined in the base class.

We do this using the `virtual` keyword in the base class and the `override` keyword in the derived class:

```
public class Animal
{
    public virtual void Eat(string meal)
    {
        Console.WriteLine("Digesting " + meal);
    }
}

public class Fox : Animal
{
    public override void Eat(string meal)
    {
        Console.WriteLine("Chewing " + meal);
        base.Eat(meal);
    }
}

public class Anteater : Animal //They have no teeth!
{
    public override void Eat(string meal)
    {
        Console.WriteLine("Swallowing " + meal); //NOTE: no call to base
    }
}

var myFox = new Fox();
myFox.Eat("rabbit"); //Output: "Chewing rabbit"
                     // "Digesting rabbit"
var myAnteater = new Anteater();
myAnteater.Eat("ants"); //Output: "Swallowing ants"
```

Virtual Properties

In addition to virtual methods, we can also have *virtual properties* on the base class that can be overridden by the derived classes.

```
public class Animal
{
    public virtual int YearDiscovered
    {
        get
        {
            return int.MaxValue;
        } //Unknown year
    }
}

public class VibraniumFairyWrasse : Animal
{
    public override int YearDiscovered
    {
        get
        {
            return 2019;
        }
    }
}
```

Yes, seriously. Species name: "Cirrhilabrus wakanda"!

Virtual methods and virtual properties allow us developers to extend the functionality of the base class without needing to use that base functionality.

Stopping Virtual Inheritance

It is possible to prevent derived classes from inheriting `virtual` methods. On the next page, we will see a new base class `A` and a derived class `B`:

```
public class A
{
    public virtual void GetDetails()
    {
        Console.WriteLine("A.GetDetails invoked!");
    }
}

public class B : A
{
    public override void GetDetails()
    {
        Console.WriteLine("B.GetDetails invoked!");
        base.GetDetails();
    }
}
```

A derived class can stop virtual inheritance by declaring a member, method, or property as `sealed` (note that class `C` inherits from class `A`):

```
public class C : A
{
    public sealed override void GetDetails()
    {
        Console.WriteLine("C.GetDetails invoked!");
        base.GetDetails();
    }
}
```

Which means that further-derived classes can no longer inherit the `sealed` method, property, or member:

```
public class D : C
{
    //We cannot override GetDetails()!
}
```

However, we can also implement a new version of the member, one whose definition is restricted to the class it was defined in, using the `new` keyword. You can see an example of doing this with class `E` on the next page.

```
public class E : C
{
    public new void GetDetails()
    {
        Console.WriteLine("E.GetDetails invoked!");
    }
}

var myClassE = new E();
myClassE.GetDetails(); //Calls the implementation in Class E
var myClassC = new C();
myClassC.GetDetails(); //Calls the implementation in Class C, then Class A
```

Fun with Polymorphism

Polymorphism becomes particularly useful in, say, a method that can accept many types but does something different with each of them.

```
public class Animal { /*...*/ } //Base class

public class Elephant : Animal //Derived class
{
    public string ElephantType { get; set; } //African, Asian, other?
}

public class Dolphin : Animal //Derived class
{
    public string DolphinType { get; set; } //Bottlenose, common, other?
}

public static class FunWithPolymorphism
{
    public static void OutputType(Animal animal)
    {
        if (animal.GetType() == typeof(Dolphin)) //GetType() defined
                                                 //on System.Object
        {
            var dolphin = (Dolphin)animal;
            Console.WriteLine(dolphin.DolphinType);
        }
        if (animal.GetType() == typeof(Elephant))
        {
            var elephant = (Elephant)animal;
            Console.WriteLine(elephant.ElephantType);
        }
    }
}

public static void Main(string[] args)
{
    var dolphin = new Dolphin() { DolphinType = "Bottlenose" };
    var elephant = new Elephant() { ElephantType = "Asian" };
    FunWithPolymorphism.OutputType(elephant);
    FunWithPolymorphism.OutputType(dolphin); Console.ReadLine();
}
```

Summary

Inheritance is the ability for classes to use members from other classes. The class that implements the original behavior is called a base class, and the class that inherits from a base is called a derived class.

Polymorphism allows for instances of derived classes to be treated as though they are instances of their base class. We can allow for implementation in a base class to be extended using `virtual` methods and properties, which can optionally call the base implementation in addition to their implementation.

Chapter 10: Interfaces and Abstract Classes

In the previous chapter, we discussed various ways implementing inheritance with C# classes and objects. In this chapter, we're going to discuss two ways a class can inherit a specific set of functionalities without inheriting an entire class.

First, we'll discuss a way to allowing class to implement a common set of property and method definitions (*interfaces*), and then we'll discuss a manner by which a class can be defined but only partially implemented so that its derived classes can use their own implementation (*abstract classes*). We will also point out some important difference between interfaces and abstract classes along the way.

Interfaces

Interfaces are special objects in C# that define a set of related functionalities which may include methods, properties, and other members. Think of interfaces as a contract, one where classes that implement an interface agree to provide implementations for all objects defined by that interface.

Interfaces cannot contain any implementations, and their names are generally prefixed with "I" to distinguish them from other C# objects. We create interfaces using the `interface` keyword:

```
public interface IAreaCalculator
{
    double GetArea();
}
```

Classes and structs can then implement an interface and define the behavior of the interface's methods (example on next page).

```
public class Circle : IAreaCalculator
{
    public double Radius { get; set; }
    public double GetArea()
    {
        return Math.PI * (Radius * Radius);
    }
}

public class Rectangle : IAreaCalculator
{
    public double Height { get; set; }
    public double Width { get; set; }
    public double GetArea()
    {
        return Height * Width;
    }
}

public class Triangle : IAreaCalculator
{
    public double Height { get; set; }
    public double Width { get; set; }
    public double GetArea()
    {
        return Height * Width * 0.5;
    }
}
```

Note that the implementing class *must* provide a definition for all methods defined on the interface. If it does not, we get a compilation error:

```
public class Oval : IAreaCalculator
{
    public double Radius1 { get; set; }
    public double Radius2 { get; set; }
}
```

Error: "Oval" does not implement interface member "IAreaCalculator.GetArea()"

Interfaces cannot be instantiated directly (attempting to do so will cause a compilation error, see the example on the next page).

```
var myAreaCalculator = new IAreaCalculator();
```

Error: Cannot create an instance of the abstract class or interface 'IAreaCalculator'

However, because of Polymorphism, we can use interfaces as the type of a variable, and the resulting object will only have the members of the interface be usable:

```
IAreaCalculator myCalc = new Circle()
{
    Radius = 2
}

double area = myCalc.GetArea(); //12.566
```

Interface Inheritance

Interfaces can inherit from one or more other interfaces:

```
public interface IMovement
{
    public void Move();
}

public interface IMakeSound
{
    public void MakeSound();
}

public interface IAnimal : IMovement, IMakeSound
{
    string SpeciesName { get; set; }
}
```

Any class or struct which implements an interface must implement all methods and properties from any of that interface's inherited interfaces (example on next page).

```
public class Dog : IAnimal
{
    public string SpeciesName { get; set; }
    public void MakeSound() //Defined in IMakeSound
    {
        Console.WriteLine("Bark!");
    }
    public void Move() //Defined in IMovement
    {
        Console.WriteLine("Running happily!");
    }
}
```

Implementing Multiple Interfaces

Likewise, a single class can implement multiple interfaces, and must define behavior for all methods and properties from those interfaces:

```
public interface IMovement {
    public void Move();
}

public interface IMakeSound {
    public void MakeSound();
}

public interface IAnimal2 {
    string SpeciesName { get; set; }
}

public class Cat : IMovement, IMakeSound, IAnimal2
{
    public string SpeciesName { get; set; }
    public void MakeSound()
    {
        Console.WriteLine("Meow!");
    }
    public void Move()
    {
        Console.WriteLine("Walking gracefully");
    }
}
```

In short, interfaces allow us to define a set of properties and method definitions which any implementing classes must provide their own implementations for.

Abstract Classes

Abstract classes serve a slightly different purpose than interfaces. An abstract class is a "partially implemented" class which other classes can inherit from, but if they do, they must provide their own implementations for any method in the abstract class that is not already implemented.

An abstract class is defined using the `abstract` keyword. This keyword tells us that the object being modified has a missing or incomplete implementation, and that classes which inherit from the `abstract` class must provide the missing pieces of the implementation.

```
public abstract class Basic
{
    //Members go here
}
```

Abstract Methods

Methods in an abstract class may also be declared `abstract`; these methods will not have an implementation. However, abstract classes can also contain methods *with* an implementation.

```
public abstract class Gem
{
    public abstract decimal GetValuePerCarat(); //Abstract method
    public string GetCommonColors() //Concrete method
    {
        return "Clear, Purple, Red, Black, White";
    }
}
```

Methods which are declared as `abstract` must be implemented in any derived class that inherits from the abstract class. An example of this is on the following page.

```
public class Garnet : Gem
{
    public override decimal GetValuePerCarat()
    {
        return 500M;
    }
}

public class Amethyst : Gem
{
    public override decimal GetValuePerCarat()
    {
        return 300M;
    }
}

public class Pearl : Gem
{
    public override decimal GetValuePerCarat()
    {
        return 400M;
    }
}
```

Extra special bonus points to whoever knows what this is from.

If our derived class does not implement all the `abstract` members, we get a compilation error.

```
public class Peridot : Gem { }
```

Error: 'Peridot' does not implement inherited abstract member 'GetValuePerCarat()'

An instance of the derived class may call methods defined in the derived class as well as any non-abstract methods in the base class:

```
var garnet = new Garnet();
var value = garnet.GetValuePerCarat(); //Defined in Garnet class
Console.WriteLine(value);
var commonColors = garnet.GetCommonColors(); //Defined in Gem base class
Console.WriteLine(commonColors);
```

Abstract Properties

Properties can also be declared abstract, and derived classes can implement them using the `override` keyword:

```
public abstract class Mineral
{
    public abstract double Hardness { get; }
}

public class Quartz : Mineral
{
    public override double Hardness
    {
        get
        {
            return 7.0;
        }
    }
}

public class Bismuth : Mineral
{
    public override double Hardness
    {
        get
        {
            return 2.25;
        }
    }
}
```

Other Details

Like an interface, an abstract class cannot be instantiated directly:

```
var myMineral = new Mineral(); //BUILD ERROR!
```

Error: Cannot create an instance of the abstract class or interface "Mineral".

Also, like an interface, an abstract class can be used as a variable type due to polymorphism. Methods which are defined in the derived classes will still be called when invoked.

```
Mineral myMineral = new Bismuth();
var valuePP = myMineral.GetValuePerPound();
```

Implementing and Inheriting

There are two basic rules to follow when trying to implement interfaces and inherit from classes:

1. A single class may implement as many interfaces as they like.
2. A single class may only inherit from one other class.

You might recall from the previous part of this series that C# does not permit multiple inheritance.

To see this demonstrated, let's consider the following interfaces and abstract class:

```
public interface IValuable
{
    decimal Value { get; set; }
}

public interface IHardness
{
    double Hardness { get; set; }
}

public abstract class Gem2
{
    public abstract string Color { get; set; }
}
```

Using these objects, we could implement the interfaces and inherit from the abstract class to build other classes:

```
public class LapisLazuli : Gem2, IHardness, IValuable
{
    public decimal Value { get; set; }
    public double Hardness { get; set; }
    public override string Color { get; set; }

    public LapisLazuli()
    {
        Value = 80M;
        Hardness = 5.5;
        Color = "Deep, ocean blue";
    }
}

//We've decided that Silica is not a valuable mineral,
//so it does not implement IValuable
public class Silica : Gem2, IHardness
{
    public double Hardness { get; set; }
    public override string Color { get; set; }

    public Silica()
    {
        Hardness = 7.0;
        Color = "Various";
    }
}
```

An Important Note!

Pretty much everything in this article is correct, until C# 8.0. At that point, the distinction between how interfaces behave and how abstract classes behave becomes much more unclear. For more details, check out Jeremy Bytes's blog post.

Summary

Interfaces create a contract, a collection of methods, properties, and other members that can be implemented by classes and structs. Classes which implement an interface must define all properties and method specified by the interface.

Abstract classes can define both concrete members which have a default implementation, and abstract members which must then be implemented by derived

classes. Concrete members can be overridden with a different implementation in derived classes.

Both interfaces and abstract classes cannot be instantiated, but due to polymorphism, an instance of a class which either implements an interface or inherits from an abstract class can be treated as though it is said interface or abstract class.

Chapter 11: Namespaces

All code needs organization, needs some way to separate out code components so that they are easy to find and reuse. C# enables such reuse and organization with *namespaces*.

Basics

Namespaces in C# are used to organize code. We can think of them as containers for related classes, methods, and objects.

```
System.Console.WriteLine("System is a namespace");
```

In the above line, `System` is a namespace and `Console` is a class in that namespace.

We include specific namespaces in each .cs file with the `using` keyword. A very basic but complete .cs file will have a set of using statements, the namespace for the file, a class or other object, and any members of that object.

```
using System;

namespace ProgramName //Namespace
{
    class Program //Class
    {
        public static void Main() //Class member
        {
            Console.WriteLine("Starting application...");
        }
    }
}
```

Visual Studio and other Integrated Development Environment (IDE) programs will create a default namespace for any new projects; this is generally the same as the project name.

Namespaces and Organization

The primary reason we use non-default namespaces (e.g. ones we create ourselves) is for better code organization. For example, in a large project, we might want to separate our models from our data access layer, e.g. whatever classes need to access a database or other data store.

In such a project, we might have the following namespaces:

```
using MyProject.Models.Users;
using MyProject.Models.Vehicles;
using MyProject.DataAccess.Repositories.Users;
using MyProject.DataAccess.Repositories.Vehicles;
```

Truth is, there's no hard rules about when to create custom namespaces and when not to. It depends on each project, and the preferences of the developers involved. I would recommend, though, that it's better to have too many namespaces rather than too few.

Aliases

It may happen that two different classes have the same name but are in distinct namespaces. This happens more often in larger projects.

Let's imagine that we have two `User` classes in different namespaces:

```
namespace OtherAPI.Models
{
    public class User { /*...*/ }
}
namespace MyProject.DTOs
{
    public class User { /*...*/ }
}
```

Let's also say that we need to use both classes in a third class in yet another namespace. A naive solution would be to use the `using` keyword to include those namespaces, but this will fail with a compilation error (example on the following page).

```
using OtherAPI.Models;
using MyProject.DTOs;

namespace MyProject
{
    public class OtherClass
    {
        public User User1 { get; set; }
        public User User2 { get; set; }
    }
}
```

Error: 'User' is an ambiguous reference between 'MyProject.DTOs.User' and 'OtherAPI.Models.User'.

Instead, we can solve this problem using namespace aliases:

```
using models = OtherAPI.Models; //Namespace alias
using MyProject.DTOs;

namespace MyProject
{
    public class OtherClass
    {
        public models::User User1 { get; set; } //Use of the alias public
        User User2 { get; set; } //MyProject.DTOs.User
    }
}
```

Please note, we don't need to have the situation described above to use namespace aliases. They can be used at any time.

Namespace Nesting

It is possible to create nested namespaces, such as in the code block on the next page.

```
namespace N1
{
    class C1 { /*...*/ }
    namespace N2
    {
        class C2 { /*...*/ }
    }
}
```

If we want to use the class `C2` we can call it using the *fully-qualified* namespace or `using` keyword.

```
var myVar = new N1.N2.C2();

//OR

using N1.N2;
var myVar = new C2();
```

We can also add other classes to the nested namespace `N2` by qualifying the namespace:

```
namespace N1.N2
{
    class C3 { /*...*/ }
}
```

Summary

Namespaces are a way to group related C# objects together by giving them a common identifier. Namespaces can be nested and can be referenced using either the fully-qualified namespace or the `using` keyword. We can also alias namespaces to make our code more readable when dealing with classes with the same name.

Chapter 12: Exceptions and Exception Handling

All programs will eventually encounter errors. C# includes some powerful tools to help us handle these kinds of unexpected situations. This process is called *exception handling* and is a part of all but the most basic programs.

Exceptions

An *exception* is an unexpected error that happens while a C# program is running. The C# runtime will transform an exception into an instance of a class, and we can use that instance to try to debug why the error happened.

All instances of an exception in C# must inherit from the class `System.Exception`. When an exception is encountered while a program is running, we say it has been *thrown*.

C# includes a huge variety of built-in exception classes. For example, if we attempt to divide by zero, an instance of `System.DivideByZeroException` will be thrown.

```
public class Program
{
    static void Main(string[] args)
    {
        int zero = 0;
        int two = 2;
        int result = two / zero; //Throws System.DivideByZeroException
    }
}
```

Understanding Exceptions

When an exception is thrown, the C# compiler will include information in that exception instance which is helpful when attempting to find out where the exception was thrown from. This includes:

- Stack Trace - The classes and methods that were called in order to execute the line of code that threw the exception. We use this to trace back through the

code and find out what kinds of arguments or method calls might have led to this exception being thrown.

- Error Message - A short message explaining what the exception was.
- Inner Exceptions - Exceptions can be wrapped in other exceptions. When this happens, the innermost exception (i.e. the exception that does not have any further inner exceptions) is often the most useful for debugging.

Exception Handling (try, catch, finally)

If we have a code block or line of code that we know might throw an exception at runtime, we can wrap it using the `try` keyword. Any exception which is caught in the `try` block can then be processed in a corresponding `catch` block.

```
try
{
    var two = 2;
    var zero = 0;
    var result = two / zero; //Will throw DivideByZeroException
    CallOtherMethod(); //Never invoked
}
catch(Exception ex)
{
    //Code that will "handle" the exception.
}
```

Execution of the code in the `try` block stops when an exception is encountered. In the above code, the method `CallOtherMethod()` will never be invoked, because the line above it will always throw an exception.

The `catch` keyword allows us to *handle* thrown exceptions. "Handling" an exception can mean many things. For example, we might want to log that the exception occurred in an error-logging system, redirect the code execution to a different method or block, or some combination of the two. An example using a `catch` block is on the following page.

```
try
{
    var two = 2;
    var zero = 0;
    var result = two / zero; //Will throw DivideByZeroException
    CallOtherMethod(); //Never invoked
}
catch(Exception ex)
{
    var errorLogger = new MyErrorLogger();
    errorLogger.Log(ex);
}
```

Swallowing Exceptions

Choosing to do nothing with a caught exception is called "swallowing" the exception and is generally considered bad practice:

```
try
{
    var result = two / zero; //Will throw DivideByZeroException
}
catch (Exception ex)
{
    //Swallowing exceptions is a bad practice!
}
```

However, something being bad practice does not mean that we should never do it. It means we should try not to do it, or only do it in specific situations where we know and understand the risks or complications that might result.

In the case of swallowing exceptions, the primary risk is that errors might occur, and because they were swallowed, we will have no idea that they even happened.

Multiple Catch Blocks

We can also `catch` specific kinds of exceptions and execute different code blocks for each. The most-specific exception needs to be listed first, and the least-specific goes last. An example of this kind of structure is on the next page.

```
try
{
    //Code that might throw an exception
}
catch (DivideByZeroException ex)
{
    //Handle the divide by zero situation
}
catch (ArgumentException ex)
{
    //Handle the argument exception situation
}
catch (Exception ex)
{
    //Handle all other exceptions
}
```

Finally Blocks

Before the execution of a `try catch` block ends, the C# compiler will check for the existence of a `finally` block. Code in a `finally` block will execute whether an exception is thrown.

```
try
{
    //Code that might throw an exception.
}
catch (Exception ex)
{
    //Code that is only executed if an exception is thrown.
}
finally
{
    //Code that is executed whether or not an exception is thrown.
}
```

A common situation in which we use a `finally` clause is to clean up our code. We might do that in situations where we are reading external files, so as not to cause a `StackOverflowException` or `OutOfMemoryException`.

In order to read a file from a local device, you must open a stream to the file. The stream remains open so long as we do not explicitly close it. We could use a `finally` block to close the stream whether an exception was thrown, and we do exactly this in the example on the following page.

```
System.IO.FileStream file = null;
System.IO.FileInfo fileInfo = null;

try
{
    fileInfo = new System.IO.FileInfo("C:\\path\\to\\file.txt");
    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);
}
catch(Exception e)
{
    //Handle the exception Console.WriteLine(e.Message);
}
finally
{
    if (file != null)
    {
        file.Close(); //Close the file stream
    }
}
```

We would do similar things in situations where we are reading from a database or datastore.

Throwing Exceptions

We can also force the C# compiler to throw an exception using the `throw` keyword.

```
if(someCondition)
{
    //Normal condition
}
else //Error Condition
{
    throw new Exception("Exception message goes here!");
```

```
}
```

We use this in situations where the program is now in an error state, and we need to stop execution before something unexpected or fatal happens. You might use this in `if`/`else` blocks or other decision statements.

One common scenario is to throw an instance of `NotImplementedException` for methods that do not have their implementation written yet:

```
public string Method()
{
    throw new NotImplementedException();
}
```

Note that the above method will not cause a compilation error, even though the method has a return type and is missing the `return` keyword.

Custom Exceptions

We can also create our own custom exception classes, such as this one:

```
public class CustomException : Exception
{
    public CustomException(string message) { /*...*/ }
}
```

Custom exception classes must inherit from either `System.Exception` or another class which inherits from `System.Exception`. Then, because of Polymorphism, an instance of our custom exception class can be treated as though it is of type `System.Exception`.

We can then throw instances of our custom exception using the `throw` keyword:

```
public static void MyMethod()
{
    throw new CustomException("This is a message!");
}
```

Re-Throwing Exceptions

There may be situations in which our `catch` block has caught an exception but needs to `throw` it so that some code block or `try catch` block elsewhere in the code can handle it.

There are two ways of doing this. The first is to simply use the `throw` keyword at the end of the `catch` block.

```
try
{
    //Code that might throw an exception
}
catch(Exception ex)
{
    //Handle the exception throw;
}
```

Doing this preserves the original stack trace so that it can be viewed when the exception is caught and handled later.

You can also "re-throw" the exception variable:

```
try
{
    //Code that might throw an exception
}
catch(Exception ex)
{
    //Handle the exception throw ex; //This is different!
}
```

However, if we do this, the stack trace is reset to start at this line of code; the stack trace that was originally in the exception is lost. Therefore, we normally prefer the `throw` method over the `throw ex` method.

Treat Exceptions as Exceptional

The key thing to remember about exceptions is that they are not normal situations and should not be treated as a normal part of the code flow.

For example, we should not throw an exception merely to reach another part of the code (e.g. treating an exception like a decision statement such as `if` or `switch`). Exceptions are not decision points.

Also, exceptions should not be used as a return type for methods; they are not intended to be used in this way.

Exceptions are exceptional and should be treated as such.

Summary

An exception in C# is an unexpected error encountered at while a program is executing. Exceptions are thrown by the program and can be caught and processed in `try`/`catch` blocks. Optional `finally` blocks execute their code regardless of exceptions thrown. We can throw exceptions explicitly using the `throw` keyword, and most importantly, exceptions are exceptional and should be treated as such.

This chapter is written as a basic introduction to exceptions and exception handling; more details can be found in the official C# documentation.

Chapter 13: Arrays and Collections

Very often, we will need to group objects together and then perform operations against said group, such as counting them, getting specific items, and more. C# provide us with two ways to group objects, *arrays* and *collections*, and while they are similar, they are used differently.

Arrays

An array is a collection of objects, each of which is of the same type. Items in an array are called *elements*.

We declare an array by specifying the type of the elements in it:

```
int[] numbers;
```

The above array has no elements and no defined size. We can create a new single-dimensional array by specifying the size of the array and using the `new` keyword:

```
int[] numbers = new int[5];
```

We initialize an array by giving it a set of values. In the below example, the size of the array is inferred by the compiler from the number of values given to the array.

```
int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7 };
```

We can also avoid the `new` keyword and give the array a set of values:

```
string[] months = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
```

To retrieve a specific value out of an array, we specify the position of the value within the array; that position is called the *index*. Note that since array indexes start at 0, if we want the seventh value, we must pass an index of 6 (example on the following page).

```
var month1 = months[0]; //"Jan"
var month2 = months[6]; //"Jul"
```

Multi-dimensional Arrays

Arrays can have multiple dimensions. For example, a two-dimensional array is an array of arrays.

We can declare a two-dimensional array by specifying the dimensions:

```
int[,] values = new int[5, 2]; //Holds 2 groups of 5 objects
```

Arrays can have as many dimensions as we want, however the more dimensions they have, the more complex they are to use. Typically, we don't create an array with more than two dimensions, and I personally have never in my life seen an array with more than three.

We can initialize a multi-dimensional array in a similar way as a single-dimensional array:

```
//This code creates a 3 x 4 multi-dimensional array
int[,] myNumbers = new int[,]
{
    {5, 2, 6, 4},
    {1, 8, 9, 2},
    {9, 3, 4, 2}
};
```

Note that if we initialize a multi-dimensional array with given values, we don't need to specify the array's size; it will be inferred from the number of values given, just like with single-dimensional arrays.

Accessing Elements

We can access elements in a multi-dimensional array by specifying the desired position in each dimension (remember that array indexes start at 0).

```
int[,] myNumbers = new int[,] { {5, 2, 6, 4}, {1, 8, 9, 2}, {9, 3, 4, 2}
};
var value = myNumbers[1, 3]; //2 ---------------------- ^
```

Ranges and Indices (C# 8.0)

Ranges

A new feature in C# 8.0 is the ability to "slice" a single-dimensional array and get a subset of that array's elements returned; these slices are called *ranges*.

For example, let's imagine we have the following array of ten names:

```
var names = new string[] { "Andy", "Brittany", "Charles", "Damian",
"Etan", "Franklin", "Georgina", "Hasan", "Indira", "Jannelle" };
```

This may technically count as a multidimensional array, since strings are themselves arrays of characters.

C# 8.0 introduces a new operator, the range operator `..`, that allows us to get a subset of a array. Let's say we want the names from Charlie (index 2) up to but not including Indira (index 8). We can get that range of names like so:

```
var nameSubset = names[2..8];
```

The object at the first index (2 in this case) *will* be included in the result set; the object at the second index (8) *will not* be included.

You can also declare ranges as variables. Said variables can then be used inside the `[` and `]` characters.

```
var range = 2..8;
var nameSubset = names[range];
```

The type of such a variable will be `System.Range`.

Open-Ended Ranges

We can also leave the range open-ended by not specifying a start point or an end point.

```
var nameSubset = names[3..]; //Gets all names from position 3
                             //to the end of the array,
                             //including the element at position 3.
var otherNames = names[..7]; //Gets all names from the beginning
                             //of the array, up to but not including
                             //the element at position 7.
```

Indices

We have already seen how to access an element using its index:

```
var name = names[4];
Console.WriteLine(name); //Etan
```

By doing this, we are accessing an element based on its position relative to the start of the array.

Starting in C# 8.0, we can also access elements based on their position relative to the *end* of the array. In other words, we can get the second-to-last element, third-to-last element, and so on.

We do this using the `^` operator:

```
Console.WriteLine(names[^1]); //Jannelle
Console.WriteLine(names[^3]); //Hasan
```

Note that if we were to use `^0`, we would get a runtime error.

Collections

Arrays are useful for dealing with a collection of a known size. Frequently, though, we don't know the number of elements we need to gather and operate on. For these situations, we are better off using *collections*.

Namespace

Generic collections are in the `System.Collections.Generic` namespace, which we need to include in any file that wants to use them:

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

The collection types we are talking about in this article are also examples of generics, which are discussed in Chapter 15: Generics.

List<T>

The most common collection type in C# is `List<T>`. T is a placeholder for a type; when we create an object of type `List<T>`, we need to specify the type of the elements that will be held by the list.

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

Just like with arrays, we can initialize collections by specifying their elements:

```
List<int> years = new List<int> { 2020, 2019, 2018, 2017, 2016 };
```

We can use a `foreach` loop to access all elements of a list (see Chapter 5: Code Blocks, Basic Statements, and Loops).

```
List<int> years = new List<int> { 2020, 2019, 2018, 2017, 2016 };

foreach(var year in years)
{
    Console.WriteLine(year.ToString());
}
```

We can also access individual elements in a collection using array indexing:

```
List<string> daysOfTheWeek = new List<string> { "Mon", "Tue", "Wed",
"Thu", "Fri", "Sat", "Sun" };
var day = daysOfTheWeek[3]; //"Thu"
```

Methods

We interact with elements in a collection primarily using methods defined on the collection class. For example, we can add elements to the list using the `Add()` method.

```
List<string> daysOfTheWeek = new List<string> ();
daysOfTheWeek.Add("Sun");
daysOfTheWeek.Add("Mon");
daysOfTheWeek.Add("Tue");
daysOfTheWeek.Add("Wed");
daysOfTheWeek.Add("Thu");
daysOfTheWeek.Add("Fri");
daysOfTheWeek.Add("Sat");
```

There are many other methods we can utilitize; here is a sample of several of them.

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

names.Add("test name"); //Adds new elements
names.Add("second name");
names.Add("third name");

bool exists = names.Contains("test name"); //Checks if a particular value
                                           //exists in the collection

List<string> aFewNames = names.GetRange(0, 2); //Returns a copy List<T>
                                               //starting from position 0
                                               //and getting the next
                                               //two elements

names.Insert(2, "second and a half name"); //Inserts an element at
                                           //the specified position

int index = names.IndexOf("test name"); //Returns the zero-based index of
                                        //the first instance of the
                                        //element.

names.Remove("test name"); //Removes the first occurrance
                           //of the specified element

names.Clear(); //Removes all elements
```

Combining Lists

It is possible to combine lists of the same type into a single list. We do this using the `AddRange()` method:

```
List<string> names1 = new List<string>{ "alex", "amy", "angela", "adam" };
List<string> names2 = new List<string>{ "brianna", "bob", "barb" };
names1.AddRange(names2);
```

Other Collection Types (Dictionary, Queue, Stack)

Besides `List<T>`, there are other useful collection types in C#. To be perfectly honest, I could write an entire book on each of these types to fully cover their usage, but I will leave the deeper exploration of these types to you, my dear readers.

Dictionary <TKey, TValue>

A *dictionary* is a set of values, each having a unique *key*. We can add elements to a dictionary using the `Add()` method.

```
Dictionary<string, string> imageTypes = new Dictionary<string, string>();
//imageTypes.Add("key", "value");
imageTypes.Add("bmp", "Bitmap");
imageTypes.Add("jpeg", "Joint Photographic Experts Group");
imageTypes.Add("png", "Portable Network Graphics");
imageTypes.Add("gif", "Graphics Interchange Format");
```

The `Add()` method will throw an exception if the specified key already exists in the dictionary:

```
Dictionary<string, string> imageTypes = new Dictionary<string, string>();
imageTypes.Add("bmp", "Bitmap");
imageTypes.Add("jpeg", "Joint Photographic Experts Group");
imageTypes.Add("png", "Portable Network Graphics");
imageTypes.Add("gif", "Graphics Interchange Format");
imageTypes.Add("jpeg", "JPEG"); //EXCEPTION
```

System.ArgumentException: "An item with the same key has already been added. Key: jpeg"

When retrieving values out of a dictionary, we use the key to retrieve the value:

```
Dictionary<string, string> imageTypes = new Dictionary<string, string>();
imageTypes.Add("bmp", "Bitmap");
imageTypes.Add("jpeg", "Joint Photographic Experts Group");
imageTypes.Add("png", "Portable Network Graphics");
imageTypes.Add("gif", "Graphics Interchange Format");

string name = imageTypes["png"]; //"Portable Network Graphics"
```

Queue<T>

A *queue* is a collection of items that is implemented in a first-in-first-out style. Think of this as waiting in line: you enter the line at the back, wait your turn, and when you are at the front, your turn is next. This means that elements of the collection will be removed from the collection in the same order they were added.

We add elements to a `Queue<T>` using the `Enqueue()` method, and remove the element at the "front of the line" using the `Dequeue()` method:

```
Queue<int> orders = new Queue<int>();

orders.Enqueue(1);
orders.Enqueue(2);
orders.Enqueue(3);
orders.Enqueue(4);

var firstValue = orders.Dequeue(); //1
var secondValue = orders.Dequeue(); //2
```

Stack<T>

A *stack* is a collection of items implemented in a last-in-first-out style. This means that the most-recently-added item will be the first one removed. Items added to a stack are said to be "pushed" and items removed are said to be "popped".

We add elements to a `Stack<T>` using the `Push()` method and remove elements using the `Pop()` method (complete example on the following page).

```
Stack<int> elements = new Stack<int>();

elements.Push(1);
elements.Push(2);
elements.Push(3);
elements.Push(4);

var firstValue = elements.Pop(); //4
var secondValue = elements.Pop(); //3
```

Summary

Arrays are collections of objects with the same type of a defined size. We access elements in an array using an index object and can initialize arrays both with and without elements. We can also have multi-dimensional arrays, where we can define X groups of Y size, or larger.

We can get a subset of the elements in an array using the range operator `..`, and elements based on their position relative to the end of the array using the `^` operator.

Collections are groups of elements with the same type, but with an adjustable size; they are more flexible than arrays.

Collections include the base `List<T>` class, as well as more specialized classes such as `Dictionary<TKey, TValue>`, `Queue<T>`, and `Stack<T>`.

We normally use methods such as `Add()`, `Remove()`, and `Contains()` to interact with elements in a collection, though we can also access elements in collections using array indexes. The `foreach` keyword provides a simple way to iterate through each element in a collection.

Chapter 14: LINQ Basics

LINQ is one of the best reasons to program in C#. It provides a simple way to query and manipulate groups of objects, and does so in an easy-to-read manner while still allowing for complex queries to be run. In many ways, it's the bread-and-butter of C# programmers who must work with databases and other data stores, or large collections of objects.

What is LINQ?

As mentioned above, LINQ (which stands for Language INtegrated Query) allows us to query and manipulate groups of objects in C#. It does this in two ways: a *query syntax* which looks a lot like SQL queries, and an *API syntax* which consists of a set of method calls.

Here's an example of the query syntax. This code block will create a list of integers, use LINQ to get the even integers out of that list, and write the even numbers to the console.

```
List<int> myNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };

var evenNumbers = from x in myNumbers
                  where x % 2 == 0
                  select x; //Get all even numbers

foreach(var num in evenNumbers)
{
    Console.WriteLine(num.ToString());
}
```

Here's that same query using the API syntax:

```
List<int> myNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
var evenNumbers = myNumbers.Where(x => x % 2 == 0);

foreach(var num in evenNumbers)
{
    Console.WriteLine(num.ToString());
}
```

In most situations, the API syntax is more concise, but certain queries are simpler to write and more easily understood with the query syntax.

Namespace

LINQ operations can be found in the `System.Linq` namespace:

```
using System.Linq;
```

Anatomy of a Query and Projections

Let's break down the query we saw earlier:

```
List<int> myNumbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };

var evenNumbers = from x in myNumbers
                  where x % 2 == 0
                  select x;
```

A basic LINQ query has three parts:

1. A `from` and `in` clause. The variable after the `from` keyword specifies a name for an *iterator*; think of it as representing each individual object in the collection. The `in` clause specifies the collection we are querying from.

2. An optional `where` clause. This uses the variable defined by the `from` keyword to create *conditions* that objects must match in order to be returned by the query.

3. A `select` clause. The `select` keyword specifies what parts of the object to select. This can include the entire object or only specific properties.

A slightly more complex query, using a custom class, appears on the next page.

```
public class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int BirthYear { get; set; }
}

var users = new List<User>()
{
    new User()
    {
        FirstName = "Terrance",
        LastName = "Johnson",
        BirthYear = 2005
    },
    new User()
    {
        FirstName = "John",
        LastName = "Smith",
        BirthYear = 1966
    },
    new User()
    {
        FirstName = "Eva",
        LastName = "Birch",
        BirthYear = 2002
    }
};

//Get the full combined name for people born in 1990 or later
var fullNames = from x in users
                where x.BirthYear >= 1990
                select new { x.FirstName, x.LastName };
```

This code block creates a *projection*. We can use LINQ to select properties of types without needing to select the entire instance, and the resulting collection consists of only the properties we selected, not the entire object.

For comparison, on the following page is an example of that same query using API syntax:

```
//Get the full combined name for people born in 1990 or later
var fullNames = users.Where(x => x.BirthYear >= 1990)
                     .Select(x => new { x.FirstName, x.LastName });
```

The rest of the samples in this chapter will be in API syntax unless otherwise noted.

Filtering and Ordering (First, Single, Distinct, OrderBy)

The two most basic operations we can perform on a LINQ query are to filter it (meaning to select only the values we want) and to order it.

Filtering

There are many ways to filter the results of a query, other than using a `where` clause.

First() and FirstOrDefault()

For example, we may want only the first item returned. To do this we must use the `=>` operator, which is the "goes to" operator, to define a condition which records must match in order to be selected.

```
var first = users.First(); //First element in the collection

//First element that matches a condition
var firstCond = users.First(x => x.BirthYear > 2001);
```

The `First()` method throws an exception if no items are found. We can have it instead return a default value by using `FirstOrDefault()` (for all C# classes, the default value will be `null`):

```
//First element in collection or default value
var firstOrDefault = users.FirstOrDefault();

//First element that matches a condition OR default value
var firstOrDefaultCond = users.FirstOrDefault(x => x.BirthYear > 2005);
```

Single() and SingleOrDefault()

We can also get exactly one item using `Single()` or `SingleOrDefault()`:

```
var singleUser = users.Single(x => x.FirstName == "John");

var singleUserOrDefault = users.SingleOrDefault(x => x.LastName ==
"Johnson");
```

Both `Single()` and `SingleOrDefault()` will throw an exception if more than one item matches the condition.

Distinct()

LINQ can even return all distinct items in a collection:

```
var indistinctNumbers = new List<int> { 4, 2, 6, 4, 6, 1, 7, 2, 7 };

var distinctNumbers = indistinctNumbers.Distinct();
```

Ordering

We can order results from a LINQ query by their properties using the methods `OrderBy()` and `ThenBy()`.

```
///Same User class as earlier
List<User> users = SomeOtherClass.GetUsers();

var orderedUsers = users.OrderBy(x => x.FirstName)
                        .ThenBy(x => x.LastName); //Alphabetical order
                                                  //by first name
                                                  //then last name
```

Note that we cannot use `ThenBy()` without first having an `OrderBy()` call.

There are also descending-order versions of these methods:

```
var descendingOrderUsers
    = users.OrderByDescending(x => x.FirstName)
           .ThenByDescending(x => x.LastName); //Reverse alphabetical
                                               //by first name
                                               //then last name
```

We can also use the `orderby` and `descending` keywords to perform an identical query using query syntax:

```
var users = new List<User>();

var myUsers = from x in users
              orderby x.BirthYear descending, x.FirstName descending
              select x;
```

Aggregation (Sum, Min, Max, Count, Average)

When operating on a collection of number values, LINQ provides a few aggregation methods, such as `Sum()`, `Min()`, `Max()`, `Count()`, and `Average()`. Each of them can optionally be used after a `Where()` clause.

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

Console.WriteLine("Sum: " + numbers.Sum()); //55

Console.WriteLine("Min: " + numbers.Where(x=> x >= 2).Min()); //2
Console.WriteLine("Max: " + numbers.Where(x => x < 7).Max()); //6

//Returns the number of elements: 10
Console.WriteLine("Count: " + numbers.Count());

//Returns the average of numbers whose value is > 3. Result: 7
Console.WriteLine("Average: " + numbers.Where(x => x > 3).Average());
```

Method Chaining

Note the last line in the previous example, the one that uses the `Average()` method. The great thing about LINQ's API syntax is that we can chain methods to produce concise, readable code, even for complicated queries.

For example: say we have a collection of users, and we need to get all combined user names (first + last) ordered by the first name alphabetically, where the first letter of the last name is J and the birth year is between 2000 and 2015.

The resulting LINQ method calls look like this:

```
var resultUsers = moreUsers.Where(x => x.LastName[0] == 'J'
                                    && x.BirthYear >= 2000
                                    && x.BirthYear <= 2015)
                           .OrderBy(x => x.FirstName)
                           .Select(x => x.FirstName + " " + x.LastName);
```

In this way, even complex queries become relatively simple LINQ calls.

IEnumerable<T> and Conversion

When using LINQ, the return type of a query is often of type `IEnumerable<T>`. This is a generic interface that collections implement in order to be *enumerable*, which means they can create an iterator over the collection which can return elements within it.

Most of the time, operating on a collection of `IEnumerable<T>` is fine if we just need certain values or a projection. We can even use `IEnumerable<T>` elements in `for` or `foreach` loops, as we saw way back in the first two code samples in this chapter.

However, sometimes what we really want is a full-blown collection. For these times, LINQ includes methods that will convert `IEnumerable<T>` to a concrete collection, such as a `List<T>` or an array.

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

var evenNumbers = numbers.Where(x => x % 2 == 0);

List<int> list = evenNumbers.ToList();

int[] array = evenNumbers.ToArray();
```

Existence Operations (Any, All)

LINQ can check for the existence of objects in a collection that match given conditions. These existence methods generally return Boolean values.

For example, let's say we have a list of users, and we want to know if any of the users were born in the year 1997. We would write that query like this:

```
bool isAnyoneBornIn1997 = users.Any(x => x.BirthYear == 1997);
```

We might also use the `Any()` method with no condition to check if there are any elements in a collection:

```
var users = SomeOtherClass.GetCertainUsers();

//True if there are any elements, false otherwise.
bool hasAny = users.Any();
```

We can also check if *all* the users in a particular collection were born in the year 1997 using the `All()` method:

```
bool isEveryoneBornIn1997 = users.All(x => x.BirthYear == 1997);
```

We can even check if a collection contains a specific value:

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

bool hasAFive = newNumbers.Contains(5);
```

Set Operations (Intersection, Union, Except)

LINQ allows us to perform *set operations* against two or more sets of objects.

Intersection

An *intersection* is the group of objects that appear in both of two lists.

```
var intersectionList1 = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var intersectionList2 = new List<int> { 2, 4, 6, 8, 10, 12, 14 };

var intersection = intersectionList1.Intersect(intersectionList2);
//{ 2, 4, 6, 8 }
```

Union

A *union* is the combined list of unique objects from two separate lists. An element which appears in both lists will only be listed in the union object once.

```
var unionList1 = new List<int> { 5, 7, 3, 2, 9, 8 };
var unionList2 = new List<int> { 9, 4, 6, 1, 5 };

var union = unionList1.Union(unionList2);
//{ 5, 7, 3, 2, 9, 8, 4, 6, 1 }
```

Except

There is also the LINQ method `Except()`, which produces the group of elements that are in the first set, but not in the second set.

```
var exceptList1 = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var exceptList2 = new List<int> { 7, 2, 8, 5, 0, 10, 3 };

var except = exceptList1.Except(exceptList2); //{ 1, 4, 6, 9 }
```

Grouping

LINQ provides many ways to group elements. These features are best demonstrated with an example.

Say we have the following `Book` class:

```
public class Book
{
    public long ID { get; set; }
    public string Title { get; set; }
    public string AuthorName { get; set; }
    public int YearOfPublication { get; set; }
}
```

Also imagine that we have a set of instances of `Book` in a collection (example code on the next page).

```
var books = new List<Book>()
{
    new Book()
    {
        ID = 1,
        Title = "Title 1",
        AuthorName = "Author 1",
        YearOfPublication = 2015
    },
    new Book()
    {
        ID = 2,
        Title = "Title 2",
        AuthorName = "Author 2",
        YearOfPublication = 2015
    },
    new Book()
    {
        ID = 3,
        Title = "Title 3",
        AuthorName = "Author 1",
        YearOfPublication = 2017
    },
    new Book()
    {
        ID = 4,
        Title = "Title 4",
        AuthorName = "Author 3",
        YearOfPublication = 1999
    },
    new Book()
    {
        ID = 5,
        Title = "Title 5",
        AuthorName = "Author 4",
        YearOfPublication = 2017
    },
};
```

One query we might want to run is to list each book in order by publication year. For this query, we don't care about titles or author names, we only care about the count of books in each publication year.

We can execute this query by using a `group by` clause. Such a query has the following format:

```
var results = from collectionVar in collectionName
              group collectionBar by collectionVar.PropertyName
                  into varGroupName
              orderby varGroupName.Key //orderby is optional
              select new
              {
                  Key = varGroupName.Key,
                  Objects = varGroupName.ToList()
              };
```

As you can see, `group by` queries are one of the kinds of queries that are easier to implement using query syntax.

Using this format, our query to get books in order by publishing year looks like this:

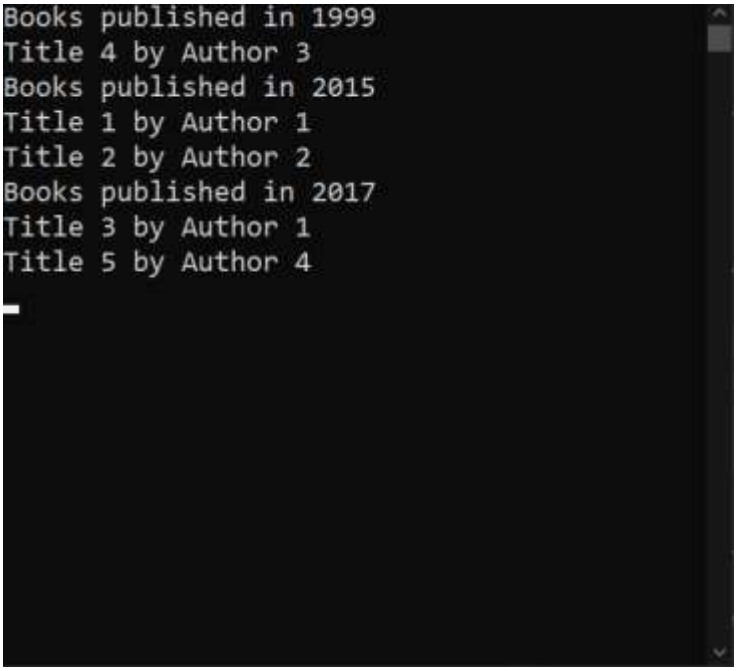
```
List<Book> books = SomeOtherClass.GetBooks();

var results = from b in books
              group b by b.YearOfPublication into g
              orderby g.Key
              select new { Year = g.Key, Books = g.ToList() };
```

We could then use a nested `foreach` loop to output all the books:

```
foreach(var result in results)
{
    Console.WriteLine("Books published in " + result.Year.ToString());
    var yearBooks = result.Books;
    foreach(var book in yearBooks)
    {
        Console.WriteLine(book.Title + " by " + book.AuthorName);
    }
}
```

This query gives the results shown in the image below.



Summary

LINQ (Language Integrated Query) is a set of technologies that allow us to operate on and select elements from collections. Among the many operations we can perform are queries, ordering, conversion, set operations, existence operations, and grouping. All these functionalities are available in either query syntax or API syntax; the latter is favored most of the time, but some functionalities are easier in the former.

There are quite a few more advanced things we can do with LINQ beyond what we discussed in this chapter. If you would like more samples, check out the 101 LINQ Samples repository.

Chapter 15: Generics

In the previous chapter, we made a lot of samples that used the `<T>` syntax. This syntax is representative of a concept in C# called *generics*.

A generic in C# is a type that uses objects of a different type. Said different type is not specified until an instance of the generic object is created. We can identify a generic type by looking for the syntax `<T>`, where T is a placeholder that represents the type being used by the generic.

```
public class MyGeneric<T> { /*...*/ }

//Elsewhere
var myGeneric = new MyGeneric<int>();
var myOtherGeneric = new MyGeneric<OtherClass>();
```

We have already seen generics in use with LINQ in Chapter 14: LINQ Basics and collections in Chapter 13: Arrays and Collections. In this chapter, we will dive deeper into what these types can do for C# developers, including how to create both generic classes and generic methods, and how to create constraints.

Creating a Generic

The most common use for generic types is to create collection classes.

For example, let's say we wanted to implement a custom collection class called `StackQueue` which implements methods for both the `Stack<T>` class and the `Queue<T>` class provided by C#.

```
public class StackQueue<T> { /*...*/ }
```

You may recall from Other Collection Types that an instance of `Queue<T>` has methods for `Enqueue()` and `Dequeue()`, and an instance of `Stack<T>` has methods for `Push()` and `Pop()`. Since `Dequeue()` and `Pop()` are the same operation (remove the element at the front of the list) our `StackQueue<T>` class will implement `Enqueue()`, `Push()`, and `Pop()`.

The implementation of `StackQueue<T>` is on the next page.

```
public class QueueStack<T>
{
    private List<T> elements = new List<T>();

    //Insert at "back of line" or bottom of list
    public void Enqueue(T item)
    {
        Console.WriteLine("Queueing " + item.ToString());
        elements.Insert(elements.Count, item);
    }

    //Insert at "front of line" or top of list
    public void Push(T item)
    {
        Console.WriteLine("Pushing " + item.ToString());
        elements.Insert(0, item);
    }

    //Remove the element at the top of the list
    public T Pop()
    {
        var element = elements[0];
        Console.WriteLine("Popping " + element.ToString());
        elements.RemoveAt(0); return element;
    }
}
```

For the `Enqueue()` and `Push()` methods, the placeholder `T` is used as the type of a parameter. For the `Pop()` method, `T` is used as the return type.

Because `StackQueue<T>` is a generic class, we can put off specifying the type that it will operate on until an instance of it is created:

```
var myStackQueue = new StackQueue<int>(); //T is now int
myStackQueue.Enqueue(1);
myStackQueue.Push(2);
myStackQueue.Push(3);
myStackQueue.Enqueue(4);

//At this point, the collection is { 3, 2, 1, 4 }
var firstValue = myStackQueue.Pop(); //3
var secondValue = myStackQueue.Pop(); //2
```

We will get a build error if we attempt to `Push()` any object which is not of the type specified by our `StackQueue<T>` instance:

```
myStackQueue.Push("a string");
```

Error: Argument 1: Cannot convert from 'string' to 'int'.

Generic Methods

Instead of creating an entire generic class, we can implement a generic method for situations where a generic class might be more than what we need.

```
public static class SwapMethods
{
    public static void Swap<T>(ref T first, ref T second)
    {
        T temp;
        temp = first;
        first = second;
        second = temp;
    }
}

public static void Test()
{
    int a = 5;
    int b = 3;
    Swap<int>(ref a, ref b);
    Console.WriteLine(a + " " + b); //Output: 3 5
}
```

Constraints

It is possible to tell the C# compiler to only allow certain types in a generic type; this is called a *constraint*. For example, we could restrict our `StackQueue<T>` class to only be usable on reference types:

```
public class StackQueue<T> where T : class { /*...*/ }
```

If we then tried to create an instance of `StackQueue<T>` with a value type we get an error.

```
var myStackQueue = new StackQueue<int>();
```

Error: The type 'int' must be a reference type in order to use it as parameter T in the generic type or method 'StackQueue<T>'.

There are many kinds of constraints we can use, and they work for all generic types including generic classes and generic methods. Let's see a few of them.

Specific Class or Interface

We can constrain a generic class to use a specific class:

```
public class StackQueue<T> where T : OtherClass { /*...*/ }
```

Due to Inheritance, this instance of `StackQueue<T>` will also accept instances of any classes which, in turn, inherit from `OtherClass`.

We can also constrain generics to use types which implement a specific interface:

```
public class StackQueue<T> where T : ISomeOtherInterface { /*...*/ }
```

Public Parameterless Constructor

We can constrain a generic class to only use types which have a public parameterless constructor.

```
public class StackQueue<T> where T : new() { /*...*/ }
```

You may remember from Constructors that any class which does not implement a custom constructor will automatically have a public parameterless constructor.

Nullables

We can constrain generic types into using either a specific type or a `null` instance using the nullable syntax:

```
public class StackQueue<T> where T : class? { /*...*/ } //Will accept
                                                        //a class or null
```

Why Use Constraints?

Constraints are about expectations. By creating a constraint, we are creating an expectation on any instance of the generic type that said instance must follow the rules of the constraint.

The primary reason we use constraints is to force compilation errors when we accidentally use a generic type for something other than what it is meant for. That way, we get these errors immediately, as opposed to waiting for some unknown condition at runtime to cause them.

Summary

Generics are types in C# which can use other types in their implementation. The type being used is not specified until an instance of the generic type is created. Generic types are often used for collections. We most often create generic classes or generic methods.

We can place constraints on generic types to restrict the kinds of types they can use. Restrictions might include using a specific class or any classes which inherit from it, using types with a public parameterless constructor, or using nullable types. Constraints work on all generic types.

Chapter 16: Tuples and Anonymous Types

As we continue our journey in C# and its features, it's worth taking a bit of time to discuss two ways to pair or group small sets of objects: *tuples* and *anonymous types*.

Tuples

A tuple is a group of values gathered in a simple structure. Unlike a collection such as `List<T>`, a tuple is immutable and of a fixed size.

We can instantiate a tuple using a special syntax.

```
(double, int) myTuple = (8.2, 6);
```

C# provides a way to get values out of a tuple as though they are properties of a class:

```
Console.WriteLine(myTuple.Item1); //8.2
Console.WriteLine(myTuple.Item2); //6
```

We can also provide property names for the values inside the tuple:

```
(double Average, int Min, int Max) secondTuple = (4.5, 2, 17);
Console.WriteLine("Average: " + secondTuple.Average.ToString()
                  + ", Max: " + secondTuple.Max.ToString()
                  + ", Min: " + secondTuple.Min.ToString());
```

A Quick Note

Tuples became available in C# 7.0. A few of the features in this article are from later versions of C#; and where this happens, it will be noted.

Why Use Tuples?

The most common use case for tuples is as a return type from a method. This is particularly useful if the method would otherwise return a class which might only be used in this one instance.

For example, let's say we have a method to get basic statistics from a list of integers:

```
public class Statistics
{
    public double Average { get; set; }
    public int Min { get; set; }
    public int Max { get; set; }

    public static Statistics GetStats(List<int> values)
    {
        Statistics stats = new Statistics();
        stats.Average = values.Average();
        stats.Min = values.Min();
        stats.Max = values.Max();
        return stats;
    }
}

static void Main(string[] args)
{
    List<int> values = new List<int> { 6, 2, 7, 9, 2, 5, 3, 8, 10, 6 };
    var stats = Statistics.GetStats(values);
    Console.WriteLine("Average: " + stats.Average
                      + ", Max: " + stats.Max
                      + ", Min: " + stats.Min);
}
```

The code in the above example can be reduced using tuples (example is on the next page).

```
public static (double Average, int Min, int Max) GetTupleStats(List<int>
values)
{
    return (values.Average(), values.Min(), values.Max());
}

static void Main(string[] args)
{
    List<int> values = new List<int> { 6, 2, 7, 9, 2, 5, 3, 8, 10, 6 };
    var stats = GetStats(values);
    Console.WriteLine("Average: " + stats.Average
                      + ", Max: " + stats.Max
                      + ", Min: " + stats.Min);
}
```

Tuples also allows us to replace out parameters from methods and can be used in place of anonymous types, which are defined later in this chapter. We talked about `out` parameters and methods in Out Keyword.

Tuple Assignment and Equality (C# 7.3)

Just like other variables, tuples can be assigned, provided the value types and amounts match:

```
(decimal, int) myTuple1 = (5.67M, 4);
(decimal pricePerUnit, int units) myTuple2 = (1.1M, 1);

myTuple2 = myTuple1;

Console.WriteLine("Price per Unit: $" + myTuple2.pricePerUnit
                  + ", Units: " + myTuple2.units);
```

Output: "Price per Unit: $5.67, Units: 4"

As of C# 7.3, we can also compare tuples using the `==` and `!=` operators.

```
(string, int) person1 = ("John Smith", 45);
(string name, int age) person2 = ("Kayla Johnson", 52);

Console.WriteLine(person1 == person2); //False
Console.WriteLine(person1 != person2); //True
```

Note that tuples compare their values from left-to-right, so if the types of values in the same place don't match, we get a build error:

```
(string, int) person1 = ("John Smith", 45);
(int, string) person2 = (45, "John Smith");

Console.WriteLine(person1 == person2); //ERROR
```

Error: Operator '==' cannot be applied to operands of type 'string' and 'int'.

Further, if two tuples do not have the same number of values, the equality operator is short-circuited, and we get a different build error.

```
(string, int, DateTime) person4
    = ("John Smith", 45, new DateTime(1975, 1, 14));

Console.WriteLine(person1 == person4);
```

ERROR: Tuple types used as operands of an == or != operator must have matching cardinalities. But this operator has tuple types of cardinality 2 on the left and 3 on the right.

Tuple Deconstruction

One of the neater features in C# 7.0 and later is the ability to "deconstruct" tuples. Deconstructing tuples allows us to assign the values in the tuple to multiple individual variables.

```
var person = ("John Smith", 45, 50000.00M); //Create a tuple
(string name, int age, decimal salary) = person; //Deconstruct the tuple

Console.WriteLine(name + ", age " + age + " makes $" + salary + "/year.");
```

We can also use the `var` keyword outside the parentheses to let the compiler determine the variable types:

```
var person = ("John Smith", 45, 50000.00M);
var (name, age, salary) = person;

Console.WriteLine(name + ", age " + age + " makes $" + salary + "/year.");
```

We can even use existing variables:

```
var person = ("John Smith", 45, 50000.00M);
string name;
int age;
decimal salary;
(name, age, salary) = person;

Console.WriteLine(name + ", age " + age + " makes $" + salary + "/year.");
```

Anonymous Types

Anonymous types are similar to tuples in many ways but are used differently. We instantiate an anonymous type using the `new` and `var` keywords:

```
var myAnonymous = new { Name = "John Smith", Age = 4 };
```

The most common scenario in which anonymous types are used is to select properties from another object or collection of objects, usually via LINQ:

```
public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
    public DateTime DateOfBirth { get; set; }
}

var users = GetUsers(); //Method not defined here
var properties = users.Select(x => new { x.Name, x.Age });
```

As we mentioned in Anatomy of a Query and Projections, when we `Select()` properties, we create a projection; this projection is an anonymous type.

Anonymous types are read-only; once instantiated, their values cannot be changed. Further, anonymous types don't have a type in the traditional sense; their type is generated and assigned to them by the compiler. Consequently, they come with many more restrictions than tuples:

- Anonymous types cannot be used as parameters or return values.
- Anonymous types may only have properties; constructors or other methods are not permitted.
- Anonymous types inherit only from `System.Object`, and therefore cannot be cast to anything except `System.Object`.

Summary

Both tuples and anonymous types are groups of values, though their purposes are not the same.

Tuples are groups of values with a defined size. They are most often used as return types from methods but can also be used to replace `out` parameters and classes which might otherwise not be necessary.

Anonymous types are read-only groups of values. Anonymous types do not have a type in the traditional sense (it is assigned to them by the compiler); they cannot be used as parameter or return types; and they are most often used with LINQ queries to get projections of classes.

Chapter 17: Attributes and Reflection

There are occasionally circumstances in which we need to know data about C# objects, rather than just the object's data. C# provides us with two techniques that allow us to do this: attributes which store the data about elements, and reflection which allows us to access information about elements.

Attributes

Attributes in C# provide a way to associate *metadata* to C# elements. We specify that we are using an attribute by placing it above the declaration for the element (e.g. the class, property, method, etc. definition) and using braces `[]`.

The .NET Framework provides several attributes for us. In fact, we have already seen one usage of them in Chapter 8: Structs and Enums: the `[Flags]` attribute.

```
[Flags] //Attribute
public enum DayOfWeek
{
    Sunday = 1,
    Monday = 2,
    Tuesday = 4,
    Wednesday = 8,
    Thursday = 16,
    Friday = 32,
    Saturday = 64
}
```

There are also several attributes defined by the .NET Framework we can use, such as the `[Serializable]` attribute, which tells the C# compiler that a given class can be serialized to another format such as JSON or XML:

```
[Serializable]
public class SerializableClass { /*...*/ }
```

It is important to note that Attributes do not provide functionality to the decorated elements; rather, they are "baked in" to the assembly at compile time. In order to

access Attributes and the data they contain, we use a technique called Reflection, which will be discussed later.

Custom Attributes

C# allows us to write our own custom attributes. Such an object must inherit from the `System.Attribute` abstract class.

Let's imagine we are building a class to represent individual novels published throughout human history. We might want an Attribute to represent information about the novels' authors, including their name and the year the novel was published. Such an attribute might look like this:

```
public class AuthorInformationAttribute : Attribute
{
    public int YearPublished { get; set; }
    public string AuthorName { get; set; }

    public AuthorInformationAttribute(int year)
    {
        YearPublished = year;
    }
    public AuthorInformationAttribute(string name, int year)
    {
        AuthorName = name;
        YearPublished = year;
    }
}
```

Attributes are a class; they can have properties, methods, constructors, and other members just like any other class. Refer to Chapter 7: Classes and Members for more information about classes.

Decorating Elements

To use our custom attribute, we would *decorate* another C# element with it. The other element can be a class, a method, a field, a property, an entire assembly, or other things; we will decorate another class:

```
[AuthorInformation("Miguel de Cervantes", 1605)]
public class DonQuixote { /*...*/ }
```

Note that since our custom attribute has multiple constructors, we can use either one:

```
[AuthorInformation(1706)] //Rough date of first English publication
public class OneThousandAndOneNights { /*...*/ }
```

Attribute Usage

We can restrict the kinds of elements that our custom Attributes can decorate using the `[AttributeUsage]` attribute:

```
[AttributeUsage(AttributeTargets.Class
                | AttributeTargets.Interface)]
public class AuthorInformationAttribute : Attribute { /*...*/ }
```

There are many options for the `AttributeTargets` flag, including `Class`, `Interface` `Method`, `Constructor`, `Enum`, `Assembly`, and more.

`AttributeUsage` also allows us to define whether objects that inherit from the decorated object also get the attribute. We do this by setting the `Inherited` property to `true`.

```
[AttributeUsage(AttributeTargets.Class
                | AttributeTargets.Interface, Inherited = true)]
public class AuthorInformationAttribute : Attribute { /*...*/ }
```

We can also specify whether there can be multiple instances of this attribute on a single element by setting the `AllowMultiple` property to `true`.

```
[AttributeUsage(AttributeTargets.Class
                | AttributeTargets.Interface, AllowMultiple = false)]
public class AuthorInformationAttribute : Attribute { /*...*/ }
```

But we are still left with a question: how do we access the values defined in our custom Attribute?

Retrieving Attributes

We can retrieve information stored in attributes using a special method defined by the .NET Framework and the `Attribute` class: `GetCustomAttribute()`.

```
var bookType = typeof(DonQuixote); //Type of the class
var attributeType = typeof(AuthorInformationAttribute); //Type of the
                                                        //attribute

var attribute = (AuthorInformationAttribute)Attribute
                .GetCustomAttribute(bookType, attributeType);
Console.WriteLine("Published by " + attribute.AuthorName
                  + " in " + attribute.YearPublished);
```

Note that in the first line, the type we need is the type of the decorated element, whether it is a class, a method, an interface, etc. We get that type using the typeof keyword, which we discussed all the way back in GetType() and typeof.

Reflection

Reflection is a technique that allows us to gather data *about* an object, rather than the data within the object itself. This information might include an object's type, information about an object's members (including methods, properties, constructors, etc.), and information about a particular assembly. It also includes any information stored in an Attribute on the element.

The simplest form of Reflection, and one that we have GetType() and typeof, is the `GetType()` method:

```
int myInt = 5;
Type type = myInt.GetType();
Console.WriteLine(type);
```

However, there are more options we can use. For example, we can also use Reflection to get information about the assembly that contains a given type (example on the next page).

```
Assembly assembly = typeof(DateTime).Assembly;
Console.WriteLine(assembly);

Assembly bookAssembly = typeof(OneThousandAndOneNights).Assembly;
Console.WriteLine(bookAssembly);
```

The code from earlier that uses `Attribute.GetCustomAttribute()` is also an example of Reflection.

Reflection is a large topic, and we won't discuss all the things we can do with it here. One of the ways we can use reflection is to get information about a method in a class:

```
public class ReflectedClass
{
    public string Property1 { get; set; }
    public int Add(int first, int second) //This method
    {
        return first + second;
    }
}

ReflectedClass reflected = new ReflectedClass();
MemberInfo member = reflected.GetType().GetMethod("Add"); //Pass name of
                                                          //method
Console.WriteLine(member); //Output: Int32 Add(Int32, Int32)
```

We can also get information about the defined property, as well as information about the object's constructor, which will be the implicit public parameterless constructor since we did not define an explicit one.

```
PropertyInfo property = reflected.GetType().GetProperty("Property1");
Console.WriteLine(property); //Output: System.String Property1
ConstructorInfo constructor = reflected.GetType()
                                       .GetConstructor(new Type[0]);
Console.WriteLine(constructor); //Output: Void .ctor()
```

Creating an Instance using Reflection

There is a very powerful class in the .NET Framework called `Activator` which can create instances of objects from types. Let's use that class to create a new instance of our `ReflectedClass` from earlier.

```
using System; //Activator is in this namespace
ReflectedClass newReflected = new ReflectedClass();
var reflectedType = newReflected.GetType();
object newObject = Activator.CreateInstance(reflectedType);
Console.WriteLine(newObject); //_17AttributesAndReflection.ReflectedClass
```

Reflection and Generics

Working with Reflection and generic types is a bit trickier than normal types. There is a property on the `Type` class to determine if the type is generic:

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7 };
Console.WriteLine(numbers.GetType().IsGenericType); //True
```

We can also do more complex things, like create a new instance of a generic `List<T>` using Reflection:

```
//Get the generic type definition from this object.
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7 };

//Create an array of arguments for the generic parameters
Type d = numbers.GetType().GetGenericTypeDefinition();
Type[] typeArgs = new Type[] { typeof(int) }; //T is int

//Make the generic type
Type constructed = d.MakeGenericType(typeArgs);

//Instantiate an object of the constructed generic type
object list = Activator.CreateInstance(constructed);

Console.WriteLine(list.GetType());
//Output: System.Collections.Generic.List`1[System.Int32]
```

Summary

Attributes assign metadata to C# elements, including classes, properties, methods, and more. This metadata is compiled away when the project is built, and describes the element, not the element's data.

We can create custom attributes that inherit from the `Attribute` class. We can restrict where those attributes are used with the `AttributeUsage` attribute, and we can retrieve attribute data using reflection.

Reflection is a technique that allows us to retrieve metadata and information *about* an element, rather than the element itself. The most basic way to do reflection is to use the `GetType()` method, but we can also use reflection to get information about methods, constructors, properties, and more.

We can even use reflection to create instances of objects, so long as we already have the type. Finally, using reflection to create generic objects is possible, but trickier; we need the type for the generic object as well as the types for all the generic parameters.

Chapter 18: Expressions, Lambdas, and Delegates

In the previous chapters, we've seen many concrete concepts that C# implements. Now, we need to take some time to talk about a few more abstract ideas. These include expressions, lambdas, and delegates, and in this chapter we'll discuss each of these and show examples.

Expressions

An *expression* is a nebulous idea in C#: it is a group of operators and operands. An expression can be a value, an assignment, a variable, an operation, a method invocation, a property access, and more.

```
long myLong = 6444296L; //Value
DateTime myDate = DateTime.Now.AddDays(7); //Method invocation
var sum = 6 + 7; //Operation
short? myNull = null; //Null
bool hasValue = myNull.HasValue; //Property access
//All of the above are also assignment expressions.
```

In the example above, both `6444296L` and `long myLong = 6444296L;` are expressions, since both values and assignments are expressions. Similarly, `DateTime.Now.AddDays(7)` is an expression, as is `6 + 7` and `myNull.HasValue`, and each of their corresponding assignments are *also* expressions unto themselves.

Expressions can be combined to form other expressions:

```
var a = 10; //Assignment
var b = a + 5; //Assignment and addition operation
var c = b % 5 == 0 ? b : a; //Assignment, modulus, and conditional
```

We learned about the conditional operator all the way back in Other Operators

C# includes many more operators, but we're going to focus on two that will most likely see widespread usage in your C# applications: the condition operator (`?:`) and the null-coalescing operator (`??`).

Conditional Operator (?:).

The last line in the above example breaks down into four expressions: a modulus operation `%`, an equality operation `==`, a conditional operation `?:`, and an assignment `=`. Remember this line; we will use it again later in this chapter.

In short, an expression produces a result and can be included in another expression. An exception to this is `void`, since methods which return that can still be included in an expression.

```
Console.WriteLine("Hello world!"); //An expression which returns void.
```

Lambda Expressions

Lambda expressions are any expression that uses the operator `=>`, which we read aloud as "goes to". You have already seen some examples of this operator back in Chapter 14: LINQ Basics; in fact, LINQ is where most of the lambda expressions you write will exist.

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var oddNumbers = numbers.Where(x => x % 2 != 0);
var sumOfEven = numbers.Where(x => x % 2 == 0).Sum();
```

The true definition of a lambda expression in C# is any piece of code that uses the `=>` operator AND has either an expression or a code block as its body.

```
(input-parameters) => expression

(input-parameters) => { code_block }
```

The idea is that the input parameters will be used by the expression or code block, and the result of that expression or code block (if one exists) will be returned.

Delegates

Any given lambda expression is also a *delegate*, which for the purposes of this article is a code block or expression that will optionally take inputs and return an output. That may sound like a method, but delegates do not behave like methods.

We can create our own delegates using the `=>` operator and the `Func<T>` class, which is called an *expression lambda* since the body of the lambda is a single expression.

```
var num = 64;
Func<int, double> squareRoot = x => Math.Sqrt(x);
Console.WriteLine($"Square Root of {num} is {squareRoot(num)}");
//Square Root of 64 is 8

num = 144;
Console.WriteLine($"Square Root of {num} is {squareRoot(num)}");
//Square Root of 144 is 12
```

A `Func<T>` defines the type of each input in order first, and then the type of the output. In the example above, the `squareRoot()` function takes an input of type `int` and outputs a value of type `double`.

We can use the type `Action<T>` when we want to use a code block instead of a single statement and where we do not want to return a value from a delegate. This is called a *statement lambda* since the body of the lambda is a collection of statements (AKA a code block). Note that delegates can access variables that are declared outside of the delegate.

```
List<string> names = new List<string>();
Action<string> sayHi = name =>
{
    Console.WriteLine($"Hi {name}!");
    names.Add(name);
};

sayHi("Nicky");
sayHi("Ravi");
sayHi("Danielle");
```

Delegates can also be created using the `delegate` keyword, however this is not the recommended way to create them, and thus examples using that keyword will not be included in this chapter.

A Delegate Example

Lambdas, delegates, and expressions save us from having to write a lot of code. Consider the code block on the next page, which we saw earlier in this chapter.

```
var a = 10; //Assignment
var b = a + 5; //Assignment and addition operation
var c = b % 5 == 0 ? b : a; //Assignment, modulus, and conditional
```

The last line is the most interesting, because it breaks down into four separate expressions (modulus, boolean compare, conditional, and assignment). Let's create a `Func<T>` delegate for each expression, not including the assignment. We can also create a combined `Func<T>` which calls the individual functions in order.

The full example for this is on the next page.

```
Func<int, int> modulus = x => { return x % 5; };
Func<int, bool> boolCompare = x => { return x == 0; };
Func<bool, int, int, int> conditional = (isTrue, x, y) =>
{
    return isTrue ? x : y;
};
Func<int, int> combined = x =>
{
    var modResult = modulus(x);
    var boolResult = boolCompare(modResult);
    return conditional(boolResult, 1, 0);
};

Console.WriteLine(combined(25)); //1
Console.WriteLine(combined(36)); //0
```

If this example seems ridiculous, that's because it is. If this code is only expected to be used one time, making the individual components of it into delegates is overkill. On the other hand, if we need to do this multiple times, having these delegates already defined can be useful. We can therefore see both sides of delegates: they can make commonly invoked code easier to invoke, but they also make our code more complex.

Please note that delegates have much more functionality than what is described here; you can find more information about them in the official Microsoft documentation:

Summary

Expressions are groups of operators and operands, which can be mere values, operations, variables, method invocations, and more.

Lambda expressions are any expression that uses the operator `=>` and are commonly used in LINQ statements.

Lambda expressions are also examples of delegates, which are collections of lines of code that can take inputs and optionally return outputs. We use the type `Func<T>` to define delegates that return an output, and `Action<T>` for delegates that will not return an output. `Func<T>` and `Action<T>` can themselves be combined into other delegates.

Chapter 19: String Manipulation and Cultures

In this chapter, we're going to take a good look at the `string` type and how it can be manipulated, displayed, parsed, etc. In order to do that, we first need to talk about an idea in .NET that allows us to automatically format and parse strings, as well as dates and times. That idea is called *cultures*.

Cultures

C# and the .NET Framework have many different "cultures", which are collections of rules about how text, dates, and times are meant to be written and can be compared. These rules include:

- The letters or symbols used in text.
- The format of dates and times (e.g. month/day/year vs day/month/year).
- The "word breakers" that determine what a word is (you can imagine that English and German word breakers are very different).
- Plus many other things.

Cultures are often identified by a code such as "en-US" (American English), "en-GB" (British English), "zu" (isiZulu), "zh-Hant" (Traditional Chinese), or "ar-sa" (Saudi Arabian Arabic).

By default, a C# program will use the culture that is set in the operating system of the machine it is running on.

We should be aware that it is possible, though rare, for cultures to change their desired formats and standards. If the United States were to decide that their standard date format is now yyyy/dd/MM, for some reason, the culture for "en-US" would change. Also, users of a system can make changes to the culture that system runs on (in the operating system), and C#/.NET will respect those changes, so it is theoretically possible for any given culture to have different rules on different machines.

Namespace

Culture information can be found in the `System.Globalization` namespace.

Creating and Using a Culture Object

We can create a culture object using the culture's code and the `CultureInfo` class.

```
CultureInfo myCulture = new CultureInfo("es-ES"); //Spain – Spanish
```

When converting objects to strings, we can pass in the format object of the culture to an overload of `ToString()`.

```
CultureInfo myCulture = new CultureInfo("es-ES"); //Spain - Spanish

DateTime now = DateTime.Now;

//Will output in the dd/MM/yyyy with 24-hour clock format
Console.WriteLine(now.ToString(myCulture.DateTimeFormat));

myCulture = new CultureInfo("en-US"); //English - United States

DateTime now = DateTime.Now;

//Will output in the MM/dd/yyyy with 12-hour clock format
Console.WriteLine(now.ToString(myCulture.DateTimeFormat));
```

The Invariant Culture

There is a special culture we can use called the *invariant culture*. This culture is not associated with any particular region of the world; however, it is based on English. Further, this culture does not change.

We access this culture using the `CultureInfo.InvariantCulture` property.

```
var invariantCulture = CultureInfo.InvariantCulture;

//Will output in the MM/dd/yyyy format
Console.WriteLine(now.ToString(invariantCulture.DateTimeFormat));
```

Escape Characters

When writing a string, it is possible to use specific characters to represent text features such as tabs or newlines; we call these *escape characters*.

For example, if we wanted to create a string with a newline character in it, we would use the `\n` escape character.

```
string withNewline = "This is on the first line.\nThis is on the second.";
```

Escape characters are always marked by a backslash (\). There are several kinds of escape characters we can use, including:

- `\t` - Horizontal tab
- `\r` - Carriage return
- `\b` – Backspace

Escaping Literals

Sometimes we want to include characters that normally mark the beginning and end of strings in the string itself, or the escape characters within. We can escape such characters, including:

- `\'` - Single quote
- `\"` - Double quote
- `\\` - Backslash

```
string dialogue = "She said, \"I didn't know that was him!\".";
```

String Operators ($ and @)

There are two special operators we can use with strings. One is `$`, the interpolated string operator, which allows us to insert variables or values directly into the string, as shown in the example on the next page.

```
string name1 = "Jack";
string name2 = "Quint";

Console.WriteLine($"Good morning {name1} and {name2}!");
```

The other operator is `@`, the *literal string operator*, which tells C# to ignore any escape characters found in the string. This is particularly useful for file paths, such as this:

```
string filePath = @"C:\this\is\a\file\path";
```

Without the `@` operator, each `\` in the string above would be read as an escape character.

Formatting Numeric Strings

We can use special character sequences to format strings that contain numbers using the `ToString()` method.

Standard Formats

For strings that contain numbers, we can use a set of standard formats.

```
decimal money = 5.67M;
Console.WriteLine(money.ToString("C")); //Currency, e.g. $5.67

double percentage = 0.67;
Console.WriteLine(percentage.ToString("P")); //Percentage, e.g. 67.00%
```

More formats can be found in the official Microsoft documentation.

Custom Formats

Let's imagine that we have a phone number that, for whatever reason, is stored in our database as an integer:

```
int phoneNumber = 2125559731;
```

We can use a custom format to output this number as a string to look like a phone number:

```
int phoneNumber = 2125559731;
string format = "(###) ###-####";
Console.WriteLine(phoneNumber.ToString(format));
//Output: (212) 555-9731
```

More information about custom numeric formats is available in the official Microsoft documentation.

Concatenation

Many times when developing a C# app, we will want to combine strings to form a new string; this is called *string concatenation*. There are several ways to do this, and some are more valid than others.

Naïve Concatenation

A naive way to concatenate strings is to use the `+` operator.

```
int value = 6;

Console.WriteLine("The value is " + value.ToString());
```

For many scenarios, this is perfectly fine. However, if you need to deal with more than just a few strings, there are better options available.

String.Concat()

Another option is the method `String.Concat()`, which combines a variable number of strings into a single `string` object. This method uses the `params` keyword we discussed Params Keyword.

```
string hello = "Hello ";
string firstName1 = "Jack, ";
string firstName2 = "Quint, ";
string firstName3 = "June, ";
string firstName4 = "and Dirk!";
```

```
var combined = string.Concat(hello,
                             firstName1,
                             firstName2,
                             firstName3,
                             firstName4);
Console.WriteLine(combined);
```

Extra special bonus points to whomever knows what these names are from.

However, this method gets harder to use and harder to read the more strings you must deal with. For large numbers of strings, there's a better way.

StringBuilder

.NET provides us with a class called `StringBuilder` which is intended to deal with large numbers of strings efficiently. To use it, we instantiate an object, add strings, and have it generate the combined result with the `ToString()` method.

```
StringBuilder builder = new StringBuilder();

for(int i = 0; i < 100; i++)
{
    builder.Append(i);
    builder.Append(" ");
}

Console.WriteLine(builder.ToString());
```

Searching Within Strings

C# and .NET provide a few methods that allow us to search within strings for specific values.

Contains()

The `string.Contains()` method returns a boolean that says whether or not the string contains another string. The string that is being searched for must be an exact match (including capitalization) for text in the string being searched. An example is shown on the next page.

```
string sentence = "This is a sentence.";
Console.WriteLine(sentence.Contains("is a")); //true
Console.WriteLine(sentence.Contains("isa")); //false
```

IndexOf()

We can get the first index of a given substring using the `string.IndexOf()` method.

```
string sentence = "This is a sentence.";

int index = sentence.IndexOf("is");
Console.WriteLine($"Found the substring 'is' at position {index}");
//Position 2. Note that this is the position of "is" in the word "this".
```

StartsWith() and EndsWith()

Finally, we can check if a string either starts with or ends with a given substring using the `string.StartsWith()` and `string.EndsWith()` methods.

```
string sentence = "This is a sentence.";

bool startsWith = sentence.StartsWith("This"); //true
bool endsWith = sentence.EndsWith("tence"); //false, missing the period
```

Modifying a String (Trimming, Padding, Case)

C# includes methods by which we can "modify" a string. Recall that, because a string is immutable, any method which "modifies" a string will, in fact, be returning a new string instance with the changes applied.

Trimming

We can remove all whitespace from the beginning, end, or both sides of a string using the `Trim()` methods. This is called *trimming*.

An example which shows how to trim strings is on the next page.

```
string sentenceWhitespace = " This has some whitespace. ";

//Removes starting whitespace
Console.WriteLine(sentenceWhitespace.TrimStart() + "End of line.");

//Removes ending whitespace
Console.WriteLine(sentenceWhitespace.TrimEnd() + "End of line.");

//Removes both
Console.WriteLine(sentenceWhitespace.Trim() + "End of line.");
```

Padding

We can add characters to either the start or end of a string using the `PadLeft()` and `PadRight()` methods. Note that each method takes the total length of the string *after* padding, so if you want a 7-character string to be padded to 10 characters, you pass 10 for this parameter.

```
string sevenDigitPhone = "1234567";

Console.WriteLine(sevenDigitPhone.PadLeft(10, '0')); //0001234567
Console.WriteLine(sevenDigitPhone.PadRight(10, '0')); //1234567000
```

Change Case

We can change the case of a string to all-uppercase or all-lowercase using their corresponding methods `ToUpper()` and `ToLower()`. An example of doing this is on the next page.

```
string mixedCaseString = "ThIs Is A mIxEd cAsE StRiNg.";

Console.WriteLine(mixedCaseString.ToUpper()); //THIS IS A MIXED CASE
STRING
Console.WriteLine(mixedCaseString.ToLower()); //this is a mixed case
string
```

Equality Comparisons

Strings are a bit trickier to compare for equality than, say, numbers. When we compare two strings, we must decide if:

- we will care about UPPER- and lower-case letters, or not,
- we want to compare strings in the same culture or not, and if not, which culture to use, AND
- whether we want to use an *ordinal comparison* or a *linguistic comparison*.

Information about ordinal or linguistic comparisons can be found in Microsoft's official documentation. Both terms are also defined in the glossary.

Naïve Comparison

We can do a simple comparison of two strings using the `==` and `!=` operators. This kind of comparison requires that the two strings are either both `null` or have the same length and identical characters at each position.

```
string test1 = "This is a semicolon ;";
string test2 = "This is a semicolon ;";
string test3 = "This is a semicolon ;"; //This one is a Greek question
mark ;.

Console.WriteLine($"test1 and test2 are {(test1 == test2 ? "equal" : "NOT
equal")}"); //equal
Console.WriteLine($"test2 and test3 are {(test2 == test3 ? "equal" : "NOT
equal")}"); //NOT equal
```

String.Equals()

We can use the `String.Equals()` method to check for equality between two strings. We can optionally pass values from the class `StringComparison` to specify how we want to check for equality. We can also choose to ignore the casing of the string using options that include `IgnoreCase` (see the example on the next page).

```
string firstString = "This is the First String.";
string secondString = "This Is The First String.";

Console.WriteLine(firstString.Equals(secondString)); //False

//True
Console.WriteLine(firstString
                  .Equals(secondString,
                          StringComparison.OrdinalIgnoreCase));
//True
Console.WriteLine(firstString
                  .Equals(secondString,
                          StringComparison.InvariantCultureIgnoreCase));
//False
Console.WriteLine(firstString
                  .Equals(secondString,
                          StringComparison.InvariantCulture));
```

Splitting Strings

We can split strings into substrings using the `String.Split()` method and identifying a character that should be split on. For example, we can split a string into a collection of works by specifying a space as a *delimiter*. Please see the example on the next page.

```
string toBeSplit = "This is a bunch of words and we will split this
sentence.";
var words = toBeSplit.Split(' ');

foreach (var word in words)
{
    Console.WriteLine(word);
}
//OUTPUT
//This
//is
//a
//bunch
//of
//words
//and
//we
//will
//split
//this
//sentence.
```

Note that the character we are splitting by will not be included in the elements of the resulting array.

We can also split by multiple characters by passing in an array of them to the `Split()` method.

```
string toParse = "This:is a\tstrange/sentence.";

char[] delimiters = { ':', ' ', '\t', '/' };
var words = toParse.Split(delimiters);
foreach (var word in words) { Console.WriteLine(word); }
//OUTPUT
//This
//is
//a
//strange
//sentence.
```

We can even split by using multi-character delimiters, per the example on the next page:

```
string toSplit = "This...is our final>>odd sentence.";
string[] stringDelimiters = { ">>", "..." };

words
 = toSplit.Split(stringDelimiters, StringSplitOptions.RemoveEmptyEntries);

foreach (var word in words)
{
    Console.WriteLine(word);
}

//OUTPUT
//This
//is our final
//odd sentence.
```

Summary

Cultures are collections of rules about to display and format text, dates, times, and more. C# can create and use them for many kinds of functionality.

When dealing with strings, we can manipulate them in many ways, including with:

- Operators like `$` and `@`.
- Escape characters such as `\n`, `\t`, or `\\`.
- Standard numeric formats like `"C"` for currency or `"P"` for percentages.
- Custom numeric format strings such as `"(###) ###-####"` for a phone number.
- Concatenation with `+`, `String.Concat()`, or the `StringBuilder` class.
- Searching in strings with `Contains()`, `IndexOf()`, `StartsWith()`, and `EndsWith()`.
- Trimming with `Trim()`, `TrimStart()` or `TrimEnd()`.
- Padding with `PadLeft()` and `PadRight()`.
- Changing the case with `ToUpper()` and `ToLower()`.

- Equality comparisons with `==`, `!=`, and `String.Equals()`.
- Splitting strings using `String.Split()` and `char` or `string` delimiters.

Chapter 20: Dates and Times

This chapter is an introduction to the wide world of dates and times in C# and .NET. We are only going to cover a few of the basics, such as the `DateTime` and `TimeSpan` structs and the `TimeZoneInfo` class.

There is much, much more that can be done with dates and times in C# than can be reasonably covered in a single chapter. If you want some idea of the scope of things that can be done with dates and times in C#, check out the Microsoft official documentation.

Dates and times, by default, rely on the set Culture of the machine the app is running on. This culture controls how dates are rendered as strings, what format it expects a date and time to exist in, and more. We discussed Cultures Cultures

DateTime

The `DateTime` struct represents a single moment in time. This is typically represented by a date and a time. Each of the following can be represented by an instance of `DateTime`:

- 15th June 1215
- 28th January 1986, 1139 Hours
- 18th February 1930
- 24th October 1648
- 12th April 1945 3:35 PM

We can instantiate a new `DateTime` instance with a default value like so:

```
DateTime myDateTime = new DateTime();
//1 January 0001 0000 Hours (Midnight)
```

However, that's not very useful. What's more useful is to use overloads of `DateTime`'s constructor to create actual dates (see the example on the next page).

```
DateTime fifteenthJune = new DateTime(1215, 6, 15);
var eighteenthFebruary = new DateTime(1930, 2, 18);
var twelfthApril = new DateTime(1945, 4, 12, 15, 35, 0);
```

Let's look closer at that last line. To create a new instance of `DateTime` with a specified date, we pass these parameters in this order:

1. The year
2. The month (1-12)
3. The day (1-31)
4. The hour (0-23)
5. The minute (0-59)
6. The second (0-59)

Using Computed Values

We can also use certain values that are computed for us by the machine our C# code is running on, such as:

```
var now = DateTime.Now; //The date and time now
var utcNow = DateTime.UtcNow; //The Universal Coordinated Time (UTC) value
for now.
var today = DateTime.Today; //Today's date, with a time value of midnight.
```

Please note that, like most other so-called "primitive" types, `DateTime` is in fact a `struct` and not a class.

DateTime Manipulation

We can manipulate instances of `DateTime` by calling methods defined by it, such as `AddDays()`, `AddHours()`, `AddYears()`, etc. You can see some examples of these methods in the code block on the next page.

```
var date = DateTime.Now;
date = date.AddHours(12);
date = date.AddMonths(1);
date = date.AddDays(7);
date = date.AddMinutes(-30);
```

There are no methods on the `DateTime` struct that will "subtract" days or hours or years, so we can instead use negative numbers, like in the last line of the above sample.

Recall that any instance of `DateTime` is immutable. We discussed the meaning of immutability Readonly Structs.

Because any instance of `DateTime` is immutable, methods on the `DateTime` struct do not modify the value of the struct instance; rather they return a new instance with the modified value. This is the same way that string modification methods behaved Modifying a String (Trimming, Padding, Case).

Displaying Dates and Times

When we want to display the value of a `DateTime` instance, we normally need to convert it to a `string` first. There are several ways of doing this. The most basic is to call the `ToString()`, which uses the machine's culture to produce a "standard" string date and time.

```
Console.WriteLine(DateTime.Now.ToString()); //10/26/2020 1:21:27 PM
```

If we want to display a date and time in a different format, we can use an overload of `ToString()` along with date and time format strings.

Format Strings

Microsoft defines a bunch of string placeholders that we can use to create custom date and time format strings. A few of these placeholders include:

- *dd* for the day of the month and *dddd* for the day name (e.g. Monday).
- *M* for the number of the month (e.g. "1", "11"), *MM* for the number of the month with a leading zero (e.g. "01", "11"), *MMM* for the month's abbreviation (e.g. "Jan"), and *MMMM* for the entire month name (e.g. January).

- *h* for the hour (e.g. "7", "12") and *hh* for the hour with a leading zero (e.g. "07", "12").
- *gg* for the era (e.g. "A.D.")

Using these placeholders, some of the formats we can generate might include:

- "MM/dd/yyyy" - 10/24/2020
- "dd MMMM yyyy" - e.g. 24th October 2020
- "dddd, dd MMM yyyy h:mm:ss gg" - Saturday, 24 Oct 2020 7:59:32 A.D.

When we want to convert a date using one of these formats, we call the `ToString()` method like so.

```
Console.WriteLine(DateTime.Now.ToString("dddd, dd MM yyyy h:mm:ss gg"));
```

More information on the various formats we can use is available [in Microsoft's official C# documentation](#).

Shorthand Methods

There are a few shorthand methods on the `DateTime` struct that we can use instead of specifying an entire format.

```
var display = DateTime.Now.ToShortDateString();
Console.WriteLine(display); //10/26/2020

var displayTime = DateTime.Now.ToShortTimeString();
Console.WriteLine(displayTime); //1:51 PM

var longDisplay = DateTime.Now.ToLongDateString();
Console.WriteLine(longDisplay); //Monday, October 26th, 2020

var longDisplayTime = DateTime.Now.ToLongTimeString();
Console.WriteLine(longDisplayTime); //1:51:13 PM
```

All of these methods will use the current culture to generate these strings. The outputs above were generated using my culture, en-US (American English).

Parsing a String to a DateTime

There are multiple ways to parse a string into an instance of `DateTime`, and which one we use depends on what the original string is and whether we want to parse using a specific format.

The first two methods we are going to talk about we first discussed Parsing. We will show some examples with them again here and discuss how we can now use format strings to use them in more specific manners.

Parse()

The `DateTime.Parse()` method will attempt to read a string and convert it to an instance of `DateTime`. Note that this method will use the current culture of the machine it is running one.

```
var date = DateTime.Parse("10/24/2020 05:50AM");
```

If the string cannot be parsed to an instance of `DateTime`, the `Parse()` method will throw an exception.

TryParse()

If we do not wish for the parsing method to throw an exception if it fails, we can use the `DateTime.TryParse()` method, which returns a `bool` for whether or not the string can be parsed and uses an out parameter (which we discussed Out Keyword) for the actual `DateTime` instance.

```
DateTime parsedDate;
bool isParsable = DateTime.TryParse("5/4/2012 11:30", out parsedDate);
Console.WriteLine(parsedDate.ToString());
```

ParseExact() and TryParseExact()

Both parse methods also have sibling methods, `ParseExact()` and `TryParseExact()`. These methods will attempt to parse a string using a given string format. If the string does not match the given format exactly, the parsing will fail. Please see the example code block on the next page.

```
string format = "ddd MM/dd/yyyy h:mm:ss";
parsedDate = DateTime.ParseExact("Mon 10/26/2020 9:15:45", format, null);
Console.WriteLine(parsedDate);
```

TimeSpan

The `TimeSpan` struct represents a *duration* of time, whereas `DateTime` represents a *single point* in time. Instances of `TimeSpan` can be expressed in seconds, minutes, hours, or days, and can be either negative or positive.

We can create a default instance of `TimeSpan` using its parameterless constructor; the value of such an instance is `TimeSpan.Zero`.

```
TimeSpan newTimeSpan = new TimeSpan();
Console.WriteLine(newTimeSpan);
```

Alternately, we can instantiate a new `TimeSpan` by passing the time duration that it represents.

```
//7 days, 8 hours, 10 minutes, and 35 seconds
TimeSpan newTimeSpan = new TimeSpan(7, 8, 10, 35);
Console.WriteLine(newTimeSpan); //7.08:10:35
```

Calculations with DateTime and TimeSpan

A subtraction operation with two `DateTime` instances results in a `TimeSpan` instance. For example, if we know the start time and date and the end time and date of a train journey, we can calculate the duration between them, as we do in the example on the next page.

```
DateTime startDate = new DateTime(2020, 11, 10, 9, 35, 0);
DateTime endDate = new DateTime(2020, 11, 14, 15, 10, 20);

TimeSpan duration = endDate - startDate;
Console.WriteLine(duration); //4.05:35:20
```

We can also use the `DateTime.Add()` and `DateTime.Subtract()` methods to add or subtract `TimeSpan` durations from `DateTime` instances.

```
DateTime initialDate = DateTime.Now;
var newDate = initialDate.Add(duration);
Console.WriteLine(newDate);

newDate = newDate.Subtract(duration);
Console.WriteLine(newDate); //Now
```

Because `TimeSpan` instances can be positive or negative, C# includes the `Duration()` and `Negate()` methods, which can be used to get the absolute value and negative value, respectively.

```
TimeSpan negative = new TimeSpan(-4, 30, 12);
Console.WriteLine(negative);
Console.WriteLine(negative.Duration()); //Positive value
Console.WriteLine(negative.Negate()); //Positive value; negating a
negative value results in a positive value.
```

TimeZoneInfo

C# and the .NET Framework can represent Earth's time zones using the `TimeZoneInfo` class.

We normally instantiate the `TimeZoneInfo` class by using the `TimeZoneInfo.FindSystemTimeZoneById()` method and passing in the name of the time zone. For example, the code block on the next page shows how I would get my home time zone.

```
TimeZoneInfo mountainStandard
    = TimeZoneInfo.FindSystemTimeZoneById("Mountain Standard Time");
Console.WriteLine(mountainStandard);
//(UTC-07:00) Mountain Time (US & Canada)
```

Time zones are defined by the system running the C# application. We can display all time zones defined on the current system in the command line by running the code on the next page.

```
var allTimeZones = TimeZoneInfo.GetSystemTimeZones();
foreach(var timeZone in allTimeZones)
{
    Console.WriteLine(timeZone);
}
```

There are many more things we can do with time zones, such as instantiating adjustment rules for daylight savings time. Check out the official Microsoft documentation for more details.

Summary

The `DateTime` struct represents a single point in time, and an instance of `DateTime` is immutable. We can instantiate it using years, months, days, and so on. We manipulate `DateTime` values by calling methods such as `AddDays()`, which returns a new instance of `DateTime`.

We can use format strings such as "MM/dd/yyyy" both to display instances of `DateTime` as strings and to parse new `DateTime` values from strings. For simple outputs, we can use shorthand methods such as `GetShortDateString()`.

Parsing a string to a `DateTime` value can be done either using a specific format with `ParseExact()` or `TryParseExact()` or using the machine's default culture with `Parse()` and `TryParse()`.

The `TimeSpan` struct represents a duration of time, expressed in days, hours, minutes, and seconds, and can be either positive or negative. We can subtract two instances of `DateTime` to get a `TimeSpan` value of the duration between them, and we can add `TimeSpan` instances to a `DateTime` to get a new `DateTime` instance.

Finally, the `TimeZoneInfo` class represents time zones. We can find an individual time zone using `FindSystemTimeZoneByID()`, or get all system time zones using `GetSystemTimeZones()`.

Footnotes

15th June 2015 is the date that the Magna Carta was signed.

28th January 1986 at 1139 Hours is the date and time of the Space Shuttle Challenger disaster.

18th February 1930 is the day that Pluto was discovered.

24th October 1648 is the date of the signing of the Peace of Westphalia.

12th April 1945 3:35 PM is the date and time of Franklin Delano Roosevelt's death.

Chapter 21: Indexers

We have now come to the part in this book where we can explore some cool but little-used C# features. Among these is the ability to access values in a class instance in the same way we access array values; we do this using a C# feature called *indexers*.

Indexers allow C# class instances to be indexed like arrays. This means we can access a value without explicitly specifying the type of said value.

This is particularly useful when defining custom collection classes. Let's say we create a custom collection class called `IntMathCollection` which keeps the elements of the collection in an array:

```
public class IntMathCollection
{
    private int[] _values = new int[1000];
}
```

For simplicity, let's populate the collection of `_values` with randomly generated integers 1-10000.

```
public class IntMathCollection
{
    private int[] _values = new int[1000];
    public IntMathCollection()
    {
        Random rand = new Random(DateTime.Now.Millisecond);
        for(int i = 0; i< 1000; i ++)
        {
            _values[i] = rand.Next(1,10000);
        }
    }
}
```

Setup

The reason we might have such a class is to find statistics about the collection of integers, such as the mean, median, and mode. Let's implement some methods to find these statistics.

The mean is a simple average:

```
public class IntMathCollection
{
    //...Other methods and properties
    public double Mean()
    {
        return _values.ToList().Average();
    }
}
```

The median is the number that appears in the middle of an ordered list of values. Since this array will always have an even number of values, we return the average of the two numbers in the middle of the ordered array.

```
public class IntMathCollection
{
    //...Other methods and properties
    public int Median()
    {
        var orderedValues = _values.ToList().OrderBy(x => x).ToArray();
        var value1 = orderedValues[499];
        var value2 = orderedValues[500];
        return (value1 + value2) / 2;
    }
}
```

Lastly, the mode is the single value that appears the greatest number of times in the collection. We can group the values and then order them using LINQ.

You can see an example of how to use LINQ to find the mode in the code block on the next page.

```
public class IntMathCollection
{
    //...Other methods and properties
    public int Mode()
    {
        return _values.GroupBy(v => v)
                      .OrderByDescending(g => g.Count())
                      .First()
                      .Key;
    }
}
```

The implementation of these methods provides a reason for this class to exist separately of another, more generic collection like `List<T>`.

Creating an Indexer on a Class

If we wanted to access a specific value in the `IntMathCollection` instance, we are currently unable to, since the class does not define a way to do this. We can create an indexer using the `this` keyword, and access individual values using the `value` keyword.

```
public class IntMathCollection
{
    public int this[int i]
    {
        get { return _values[i]; }
        set { _values[i] = value; }
    }
    //...Other methods and properties
}
```

Note that an indexer is actually just a special property of the class, complete with `get` and `set` accessors. Further, we can use this new indexer like so:

```
IntMathCollection myInts = new IntMathCollection();
Console.WriteLine("Element at position 67 is " + myInts[67]);
Console.WriteLine("Element at position 813 is " + myInts[813]);
Console.WriteLine("Element at position 490 is " + myInts[490]);
```

Using Expression Body Members

Since C# 7.0, we have been able to implement `set` accessors using the expression body member syntax (we could implement `get` accessors using this syntax before C# 7.0). Our indexer can now be simplified to the following:

```
public class IntMathCollection
{
    public int this[int i]
    {
        get => _values[i];
        set => _values[i] = value;
    }
    //...Other methods and properties
}
```

Creating an Indexer on an Interface

We can also create an indexer on an interface using a slightly different syntax.

```
public interface IHasIndexer
{
    string this[int index] { get; set; }
}
```

Recall that we previously discussed interfaces Interfaces

Classes which implement this interface must then provide the implementation for the indexer.

```
public class HasIndexer : IHasIndexer
{
    private string[] _values = new string[10];
    public string this[int index]
    {
        get => _values[index];
        set => _values[index] = value;
    }
}
```

Other Details

It is also possible to do the following:

- Create an indexer with more than one index value. This would most commonly be done for accessing Multi-dimensional Arrays.
- Use values other than `int` for the index (such as `string`, `double`, or Chapter 2: Primitive Types, Literals, & Nullables).

Check out the official Microsoft documentation for more.

Summary

Indexers in C# allow class instances to be accessed like arrays. We create an indexer using the `this` keyword and get the value being given with the `value` keyword. We can implement indexers on both classes and interfaces, with interfaces needing a slightly different syntax.

Chapter 22: Iterators

A few chapters ago, we talked about Chapter 13: Arrays and Collections, and how easy they were to deal with. In this post, we'll talk about a feature of C# that allows us developers to iterate over many kinds of collections and return elements from them one-by-one. Let's learn about *iterators*!

The yield Keyword

Iterators in C# take advantage of a special keyword: `yield`. The `yield` keyword is used exclusively when iterating over collections and arrays and allows us to return elements in a collection or array individually.

To demonstrate this, let's consider the Fibonacci sequence, where each number is the sum of the previous two numbers. The sequence begins like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Let's build a C# method to calculate the Fibonacci sequence to a specified number of places.

```
public static IEnumerable<int> Fibonacci(int length)
{
    int a = 0, b = 1;

    for (int i = 0; i < length; i++)
    {
        yield return a;
        int c = a + b;
        a = b;
        b = c;
    }
}
```

We should note two things about this code block. First, the return type of `IEnumerable<int>` states that the return type will be a collection of integers.

We should also note the strange placement of the `yield return a;` line of code. This line tells the compiler that when an iterator iterates over this collection, the value

it will get will be the value of `a` at this point. We have placed it before the other lines inside the `for` loop because we want to return the value of `a` before it is changed.

What's most interesting about this `yield return` code block is how we can use it in a `foreach` loop. We can run this method to show the first ten numbers in the Fibonacci sequence like so:

```
static void Main(string[] args)
{
    // Display the Fibonacci sequence up to the tenth number.
    foreach (int i in Fibonacci(10))
    {
        Console.Write(i + " ");
    }
}
```

Which results in this output:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

We can even have multiple `yield return` statements in a single code block, such as this example:

```
public static IEnumerable<int> SomeNumbers()
{
    yield return 1;
    yield return 2;
    yield return 7;
    yield return 8;
    yield return 5;
}
```

The object returned from this method is of type `IEnumerable<int>`, which means we can treat it as a collection of integers and output each number or use LINQ to find the average of all the numbers. You can see an example of how to do this in the code block on the next page.

```
var someNumbers = SomeNumbers();

string allNumbers = "";
foreach(int number in someNumbers)
{
    allNumbers += " " + number.ToString();
}

Console.WriteLine(allNumbers.Trim());
Console.WriteLine("Average: " + someNumbers.Average());
```

In short, the `yield` keyword allows us to return elements in a collection one-by-one to a calling method, and the `IEnumerable<T>` interface allows us to specify that a collection of objects can have an iterator.

Now that we know how the `yield` keyword works, we can learn about iterators themselves.

Implementing Iterators

Like the indexers from the previous chapter, iterators are most useful on collections. Let's create a custom `MonthsOfTheYear` class which holds a list of the months in a year.

```
public class MonthsOfTheYear : IEnumerable  {
    private string[] _months =
    {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };
}
```

We can now create an iterator for this class which will `yield return` each month name in the private `_months` array.

```
public class MonthsOfTheYear : IEnumerable
{
    //...Other properties and methods
    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < _months.Length; index++)
        {
            // Yield each month of the year.
            yield return _months[index];
        }
    }
}
```

We can then instantiate an object from the `MonthsOfTheYear` class and iterate over each name:

```
MonthsOfTheYear months = new MonthsOfTheYear();
foreach(var month in months)
{
    Console.WriteLine(month);
}
```

This allows us to treat the instance of `MonthsOfTheYear` as though it itself is a collection.

Iterators and Generic Collections

We can also create iterators on custom generic collections.

Back in Chapter 15, we discussed Creating a Generic. In that chapter, we created a class `StackQueue<T>` that could both *enqueue* elements (put them at the "back of the line") and *push* elements (put them at the "front of the line").

The class we wrote, `StackQueue<T>`, looked like the example on the next page.

```
public class StackQueue<T>
{
    private List<T> elements = new List<T>();

    public void Enqueue(T item)
    {
        Console.WriteLine("Queueing " + item.ToString());
        elements.Insert(elements.Count, item);
    }

    public void Push(T item)
    {
        Console.WriteLine("Pushing " + item.ToString());
        elements.Insert(0, item);
    }

    public T Pop()
    {
        var element = elements[0];
        Console.WriteLine("Popping " + element.ToString());
        elements.RemoveAt(0);
        return element;
    }
}
```

We will modify this class so that we can implement an iterator.

Modifying StackQueue<T>

The very first thing we need to do is to have our `StackQueue<T>` class implement the `IEnumerable<T>` interface.

```
public class StackQueue<T> : IEnumerable<T>
```

Now we need a private variable `_top` that tracks the number of elements in the collection. We will use this in our iterator to know when to stop getting values out of it.

To implement this, we need our `Enqueue()`, `Push()`, and `Pop()` methods to increment or decrement `_top`. Our modified class now looks like the example on the following page.

```
public class StackQueue<T> : IEnumerable<T>
{
    private List<T> elements = new List<T>();
    private int _top = 0; //NEW

    public void Enqueue(T item)
    {
        Console.WriteLine("Queueing " + item.ToString());
        elements.Insert(elements.Count, item);
        _top++; //NEW
    }

    public void Push(T item)
    {
        Console.WriteLine("Pushing " + item.ToString());
        elements.Insert(0, item);
        _top++; //NEW
    }

    public T Pop()
    {
        var element = elements[0];
        Console.WriteLine("Popping " + element.ToString());
        elements.RemoveAt(0);
        _top--; //NEW
        return element;
    }
}
```

Now we can implement our iterator using the `yield` keyword:

```
public class StackQueue<T> : IEnumerable<T>
{
    //...Other properties and methods
    public IEnumerator<T> GetEnumerator()
    {
        for(int index = 0; index < _top; index++)
            yield return elements[index];
    }

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

Finally, we can instantiate, populate, and iterate over a `StackQueue<T>` instance like this:

```
var myStackQueue = new StackQueue<int>(); //T is now int

myStackQueue.Enqueue(1);
myStackQueue.Push(2);
myStackQueue.Push(3);
myStackQueue.Enqueue(4); //At this point, the collection is { 3, 2, 1, 4 }

foreach(var item in myStackQueue)
{
    Console.WriteLine(item);
}
```

Summary

The `yield` keyword allows us to return elements in a collection or array one-by-one to a calling method. We use `yield` to construct iterators, which iterate over collections. Iterators in a class behave much like properties of that class and enable the class to be used in `foreach` loops with minimal code. Finally, iterators can also be used in custom generic collection classes like `StackQueue<T>`; we just need a variable to keep track of the elements in the collection.

Glossary

- *Abstract class*: A base C# class for which the implementation is either missing or only partially complete. This class can be inherited and may optionally have its members overridden.
- *Access modifier*: A C# keyword that specifies which other parts of the code can call the block with the modifier. Values include `public`, `private`, `protected`, and `internal`.
- *Anonymous types*: A data structure that encapsulates a set of read-only properties into a single object, without the need to explicitly define a type first. Commonly used in LINQ projections.
- *Arguments*: Values passed to a method for each of its parameters.
- *Array*: A data structure that stores multiple elements of the same type. Arrays are of a fixed size.
- *Auto-implemented properties*: Properties which use the `{ get; set; }` syntax and do not specify custom getter and setter methods. The compiler implements basic get and set functionality for these properties.
- *Base class*: A C# class which is inherited by another class. Other classes are said to "inherit" from this class.
- *Bit flag*: A specialized kind of enumerations that can be used in combinatorial logic. Each value in a bit flag must be a power of 2.
- *Boolean*: A value which must be either `true` or `false`.
- *Boolean logic*: Combining multiple Boolean values to get a resulting value.
- *Calculated properties*: Properties which return values based on the current value of other properties in the same class.
- *Casting*: "Forcing" a value to change from one type to another type. Casting a value to an incompatible type will cause errors.

- *Code block*: A group of lines of code that will run in order. Code blocks are bounded by curly braces `{ }`.
- *Collection*: A data structure which provides a flexible way to deal with groups of objects of the same type. The collection can grow and shrink dynamically as elements are added to or removed from it.
- *Condition (LINQ)*: A Boolean expression used to determine whether an individual element will be returned as part of a query.
- *Condition (loop)*: A Boolean expression in a loop; the loop will continue to execute so long as the condition is `true`.
- *Constraint (generics)*: Used to restrict the kinds of types that can be used in a generic type.
- *Constructor*: a specialized method that sets the initial values of properties or fields in a class.
- *Conversion*: Attempting to change an instance's type to another type; conversion is more forgiving than casting but is also computationally more expensive.
- *Culture*: A set of definitions, including the writing system, default date and time formats, string formats, the calendar used, string sort order, and more.
- *Decorate*: The term we use for adding an attribute to a C# element; we say we "decorated" the element with the attribute.
- *Delegates*: A set of code that will be dynamically invoked later in the execution. Can contain either a single expression or multiple expressions.
- *Delimiter*: An identifying mark or character used to separate pieces of information from one another.
- *Derived class*: a C# class which is inheriting from another class.
- *Dictionary*: A collection where each element has a unique key and a value.
- *Element*: An individual object stored in an array or collection.

- *Enumerable*: Able to be counted one-by-one.
- *Enumerations*: Integer values which have been assigned names. Used to avoid magic numbers.
- *Escape characters*: Special characters in a string that are processed differently from the rest of the string. For example, the escape character '/n' will be rendered as a newline.
- *Exception*: An unexpected error which occurs while our program is executing.
- *Exception handling*: The process by which code generates and handles exceptions.
- *Expression*: A group of operators and operands that produces a result and can be used in other expressions. This includes values, variables, method invocations, property accesses, and more.
- *Fields*: Direct-access properties in a class. Generally, we want to avoid using these in favor of using properties with getters and setters.
- *Floating-point numeric types*: C# types which represent non-whole numbers, such as `float`, `double`, and `decimal`.
- *Fully-qualified*: When we use an entire namespace identifier, we say that we are using the "fully-qualified" namespace (e.g. `System.Collections.Generic`).
- *Garbage Collection*: A feature of C# where unused allocations in the memory heap are automatically freed. This process requires no input from the developer.
- *Generic*: a C# type that uses other types. The type being used will be defined when an instance of the generic type is created. Uses the syntax `<T>`.
- *Getter and setter methods*: Specialized methods which retrieve (get) and change (set) the value of a particular property.

- *Handle*: When dealing with exceptions, the code that does something with the thrown exception (e.g. logging it, changing the code that will be executed, etc.) is said to "handle" the exception.

- *Immutable*: Unable to be changed. In programming, an immutable object cannot have its state changed after it has been instantiated.

- *Implement*: Classes which use a specific interface are said to be implementing that interface.

- *Implicit casting*: Changing a value from one type to another without specifying how to do this; C#'s compiler will determine how to do the casting.

- *Implicit constructor*: A constructor generated automatically by the compiler. Classes without a defined constructor will have an implicit constructor.

- *Index*: The position of an element in an array or collection.

- *Indexer*: A special kind of property in a C# collection class that allows calling code to access elements in the collection.

- *Inheritance*: A characteristic of object-oriented programming languages that allows for classes to adopt and use members, properties, methods, etc. defined in another class.

- *Initializer (loop)*: A variable that changes value after each run of a loop.

- *Instance*: An object created from a definition, where the definition can be a class, struct, value, etc. Instances are said to be instantiated from classes, structs, or other C# objects.

- *Instantiate*: To create an instance of. For example, in the line `int myYear = 2020;`, the variable `myYear` has been instantiated with a type of `int`.

- *Integral numeric types*: C# types which represent whole numbers, such as `int`, `long`, `short`, and `byte`.

- *Iterator*: Objects which "iterate" or move through collections, returning individual elements of said collection.

- *Iterator (LINQ)*: A representation of an object in a collection, which is used to iterate over the entire collection.
- *Iterator (loop)*: Specifies the amount by which the initializer variable will change after each execution of the loop.
- *Interface*: A "contract" which classes can implement and thereby agree to create implementations for each of the interface's members.
- *Key*: An object which uniquely represents a value in a collection. Most used with dictionaries.
- *Lambda expression*: A specific kind of expression that uses the "goes to" operator `=>`. Code in the body of a lambda can be invoked later in the execution.
- *Linguistic comparison*: Comparing two or more strings by using culture-specific rules, such as whether to ignore the "-" character in the word "co-op".
- *Literal marker*: a single character which identifies the type of the specified value. For example, using the literal marker M in `5.67M` means that this value will be instantiated as type `decimal`.
- *Loop*: A code block that may be executed multiple times. There are four variants: `for`, `foreach`, `while`, and `do while`.
- *Magic numbers*: Numbers which do not have an apparent meaning. Using magic numbers is considered an anti-pattern and should be avoided.
- *Member*: a property, method, constructor, or other object that is part of a class.
- *Metadata*: Data about data; in C#, means information that will be compiled into the assembly using Attributes.
- *Method*: A code block that can be “invoked” or called by other code. It must have a name and an access modifier and may have parameters.
- *Multiple inheritance*: A situation where a single class inherits from multiple other classes. This is not permitted in C#; attempting to do this will cause a build error.

- *Namespace*: A "container" that groups C# objects together. We can use all items in a given namespace with the `using` keyword.
- *Null*: A literal type that represents an unknown value.
- *Null checking*: A programming style or pattern in which reference types are checked for a null value prior to being used.
- *Nullable types*: A value type in C# that allows an object to be `null`, in addition to its normal value range. We use the operator `?` to identify a nullable type.
- *Operands*: Values that used in an operation and operated upon by operators.
- *Operators*: Symbols or groups of symbols that perform operations (such as addition) on two or more operands.
- *Ordinal comparison*: Comparing two or more strings by looking at the position and content of their characters.
- *Padding*: The act of adding characters to the beginning or end of a string until that string is of a specified length.
- *Parameters*: Names and types of values that are expected to be passed into a method invocation.
- *Parsing*: The process by which we attempt to take a `string` value and create a new instance of another type with that value.
- *Pascal Casing*: A naming strategy in which each word in the name will begin with a capital letter (e.g. `ThisIsAPascalCasedName`)
- *Polymorphism*: A characteristic of object-oriented programming languages that allows for a derived class to be treated as though it is an instance of its base class.
- *Precision*: The degree to which values of "numbers after the decimal point" are correct. Types `float` and `double` sacrifice precision for speed, whereas type `decimal` maintains precision but is more computationally expensive.

- *Primitive types*: "Basic" C# types, including `int`, `char`, `bool`, `decimal`, and others.
- *Projection (LINQ)*: A set of properties from a class that we select as part of a LINQ query. Can also contain properties from multiple objects.
- *Property*: A specific value member of a class. Has getter and setter methods.
- *Public parameterless constructor*: A constructor which has no parameters. Implicit constructors are always public parameterless constructors.
- *Queue*: A collection implemented in a first-in-first-out (FIFO) style.
- *Range*: A subset of elements in an array, accessed using the `..` operator.
- *Reference type*: An object which is a pointer to a location on the memory heap; the value of the object exists at that location.
- *Return type*: The type of the object that will be returned by a method. If the method will not return any object, the return type will be `void`.
- *Set operations (LINQ)*: Operations on two or more collections. Can produce the intersection, union, or except group.
- *Signed*: When referring to numbers, a signed number can represent either positive or negative values.
- *Stack*: A collection implemented in a last-in-first-out (LIFO) style.
- *String concatenation*: The act of combining two or more `string` objects into a single `string` object.
- *Strongly typed*: A programming language is strongly typed if every kind of object it can represent has a type. C# is strongly typed because every instance must have a type, e.g. `int`, `string`, `object`, etc.
- *Structure type*: A value type that is similar to a class and can have methods and properties. We use the `struct` keyword to create it.
- *Thrown*: When an exception occurs in our program, we say that the program has "thrown" the exception.

- *Trimming*: The act of removing unnecessary whitespace from the beginning, end, or both sides of a `string` object.
- *Tuple*: A lightweight data structure that can group multiple elements.
- *Type*: A set of properties about a given object, such as space required, maximum size, etc.
- *Type Inheritance*: Types can assume the properties, attributed, constraints, and characteristics of other types.
- *Type Safe*: Objects which are created with a type will continue to behave as that type. Alternately: types cannot be combined in an unsafe manner.
- *Unsigned*: When referring to numbers, an unsigned value can only represent positive numbers.
- *Value type*: An object which carries its value around with it.
- *Virtual methods*: Methods in a base class that can be overridden by that base's derived classes.
- *Virtual properties*: Properties in a base class that can be overridden by that base's derived classes.

Keyword Reference

Several of the keywords below are “contextual” keywords; their purpose changes on where they are used. Where this happens, it will be noted, and each purpose of the keyword will be described.

- `abstract` - Identifies a class with a partial or missing implementation. Inheriting classes must provide the remainder of the implementation.
- `base` – Used in derived classes to invoke constructors in the base class.
- `break` – Stops the execution of a loop OR marks the end point of a `case` in a `switch` statement.
- `catch` - Specifies a kind of exception that will be "caught" and processed.
- `class` – Used to create a C# class definition
- `continue` – Ends the current iteration of a loop and resumes execution at the next iteration.
- `delegate` - Creates a new delegate. We prefer to use lambdas => when creating a delegate instead of this keyword.
- `descending` - In a LINQ query, specifies that the objects are to be ordered by the given property in descending order.
- `do while` – Creates a loop that must execute at least one time and continues executing so long as a given condition is `true`.
- `else` – Specifies a code block that will be executed if none of the previous `if` or `else if` conditions were `true`.
- `else if` – Specifies a code block that will be executed if a given condition is `true`. Must occur in tandem with an `if` block.
- `enum` – Used to create an enumeration.

- `finally` - Specifies a code block that will be executed regardless of whether an exception was thrown. Is optional, but if it is used, it must be after a `try` block.
- `for` – Creates a loop that executes a defined number of times.
- `foreach` – Creates a loop that executes over each item in an array or collection.
- `from` - In a LINQ query, specifies a variable name to use for the iterator over a collection.
- `get` – Creates a getter method for a property.
- `if` – Specifies a code block that will be executed if the statement's condition is `true`.
- `in` - In a LINQ query, specifies the source collection the query will execute against.
- `interface` – Identifies a new C# interface object. Objects which implement an interface must provide implementation for all the interface's members.
- `namespace` - Creates a namespace with the given identifier.
- `new` – Used to create a new instance of an object.
- `orderby` - In a LINQ query, specifies one or more properties to order the results by.
- `out` – In methods, specifies that a parameter will be passed-by-reference. `out` parameters may only be modified by the method they are passed to, and they do not need to be initialized.
- `override` – Used to implement new functionality in place of already-defined functionality in a base class or interface.
- `params` – Used as a shortcut to pass a set of values of the same type to a method as a collection.

- `readonly` – Contextual keyword. Used to mark properties as only changeable during instantiation or through the constructor, OR to mark structs as immutable.
- `ref` - Used to pass parameters by reference. Must be given a value before being passed to a method.
- `return` – Returns a value to the calling code, OR ends execution of a method, loop, or other code block.
- `sealed` - Specifies that a class or member cannot be inherited.
- `select` - In a LINQ query, specifies the objects or projections that will be created by the query.
- `set` – Creates a setter method for a property.
- `struct` – Create a structure type.
- `this` – refers to the current instance of a C# class. Can also be used in extension methods, and to create indexers.
- `throw` - Throws an exception.
- `try` - Specifies a block of code that could throw an exception, where that exception will now be handled.
- `using` – Allows a file to use items defined in a namespace. Must occur at the beginning of the file.
- `value` – Contextual keyword. In `set` accessors, refers to the value being assigned to the property or indexer.
- `virtual` - Specifies that a member, property, method, etc. can be overridden.
- `void` – Used as the return type for methods that will not return any values.
- `where` - In a LINQ query, specifies one or more conditions that objects in the collection must satisfy in order to be selected.
- `while` – Creates a loop that will be executed for as long as the loop's condition is `true`.

- `yield` – Used to identify an iterator. Tells the compiler to return elements in a collection one-by-one to a calling method.