NEWSTONE

Mastering C# (C Sharp Programming)

A Step by Step Guide for the Beginner, Intermediate and Advanced User, Including Projects and Exercises

Michael B. White

Mastering C#
(C Sharp Programming)

A Step by Step Guide for the Beginner, Intermediate and Advanced User, Including Projects and Exercises

Table of Contents

Introduction

If I had a dollar for every time I spoke to a person who dreamed of being a video game programmer believing it was going to make them rich and then admitted they didn't know how to program, well, I'd be a millionaire by now.

If you can't program, how on earth are you going to become the next big thing in video games?

Although, you could go into game design, I suppose, and pay others to do everything else for you.

Let's be honest, though, even a game designer needs to understand everything that goes into a game and that includes programming.

So, whether you want a career as a game designer, developer or just want to learn how to program, C# is a great place to start. I have to be honest with you, though. When you first start to learn to program, it can be a little on the boring side –lots of theory and not a lot of programming. To try and make things better, I've added lots of examples in this guide that you can try for yourself and to break the theory up into more manageable sections.

C# is one of the main programming languages used for gaming. While the focus of this guide is not on games and game design, but if that's where you want to be aiming then this is the guide for you. Gaming requires all the main computer science fields – programming, data structures, algorithms, GUI design, networking, AI, computer graphics and more.

So, read on for a brief look at what C# is and what it's all about before we dive headfirst into programming.

What is C#?

Generally, every computer program is nothing more than a set of instructions, sometimes simple, sometimes more complex, that the computer will follow. While computers are quite stupid on the one hand, they are very good at following instructions and it will be down to you to create those instructions.

However, you can’t write your instructions in any old way because a computer can only understand binary, which is a language written in 0’s and 1’. In the old days, a series of switches would be used, turned on and off to represent the o’s and 1’s but this became tedious.

That’s why programming languages were created; you write your instructions in good old-fashioned English text format – known as source code – and then you give that code to another program, which then translates your source code into a language that the computer understands. You will often see this called “compiled code”, “executable code”, “binary file” or just an “executable”.

Do a quick search on the internet and you will find hundreds of different programming languages – each has its own purpose and not all languages can be used for everything. You can compare the computer languages to spoken languages – some are quite similar to one another while others are wildly different. Some are used all the time while others are only used by small numbers of people and most good programmers will be au fait with at least a few of them.

Out of all of them, C# is undoubtedly one of the best. I wouldn’t say it was the easiest, but it is popular. It belongs to the C/C++ language family which means that syntax between the languages is very similar, and that includes Java too. Once you have learned C#, you will find it very easy to go on and learn C/C++ and Java – if you don’t want to, at the very least, you will be able to read code written in those languages and understand it.

Likewise, if you already have a good grounding in those languages then C# will be easy for you.

About This Guide

This guide has been divided into five parts. In the first part, we’ll go over the basics of C# programming before moving onto intermediate and advanced subjects, and then we’ll take a look at some very advanced subjects. We’ll finish with an in-depth look at unsafe code, an important thing for you to learn, and then an introduction to

Windows Forms. You will also find a short quiz at the end of the first advanced section and the answers are provided at the end of the guide.

You are not required to have an understanding of C# and this is more of a crash course in C# programming so don’t expect to learn absolutely everything – it would far too long a book.

What I have tried to do is give you a head start, enough information about the most important parts of C# so that you could put your own program together.

All I can say to you now is, let’s get started on this amazing and exciting journey.

Part 1: Beginner Guide

Before we can even think about the programming aspect, we really need to get some of the background out of the way. You may know what C# is but did you know that it runs on the .NET framework?

.NET Framework

.NET is a framework developed by Microsoft for software applications. It is used by developers to build web applications, mobile, and Window applications and offers wide support for industry standards along with a huge range of functionalities.

Inside .NET you will find the FCL, or the Framework Class Library, a large number of class libraries. Software that is written in .NET executes in the CLR, or Common Language Runtime environment and both of these are core parts of the framework.

.NET provides a number of different services, such as networking, memory management, memory safety, and security, and it provides full support for multiple programming languages, of which C# is one.

These are all the parts that make up the framework:

- **CLR -** The Common Language Runtime is an execution engine, an interface between the operating system and the framework.
- **FCL -** The Framework Class Library contains a collection of thousands of classes that you can use to build your applications. The core of the library is BCL, or Base Class Library, which provides the basic functionalities.
- **WinForms –** Windows Forms is a collection of managed libraries, smart client technology for the framework that makes it much simpler to do common tasks, such as reading and writing to file systems.
- **ASP .NET –** A web framework that Microsoft designed and developed, used for developing websites, web services, and web applications. It helps with the integration of JavaScript, CSS, and HTML.
- **ADO .NET –** A .NET framework module that establishes the connection between data sources and the applications. Those data services could be XML or SQL Server, for example. ADO .NET is made up of classes that are used for retrieving connecting, inserting, and deleting data.

- **WPF –** Windows Presentation Foundation is a Microsoft graphical subsystem that is used in Windows applications for rendering the UI and it makes use of DirectX.
- **WCF –** Windows Communication Foundation is a framework used to build applications that are service-oriented. With WCF, asynchronous messages can be sent between service endpoints.
- **WF –** WorkFlow Foundation is another technology from Microsoft that provides users with an API, a re-hostable designer and an in-process workflow engine to help with the implementation of long-running processes in .NET applications.
- **LINQ –** Language Integrated Query was first introduced with .NET 3.5 and is used for making queries with C# for data sources.
- **Entity Framework –** An ORM-based framework, open-source, used for working with databases that use .NET objects. It takes a lot of effort out of the developer's work in handling the database and is the recommended database technology.
- **Parallel LINQ –** Otherwise known as PLINQ, this is a parallel implementation of LINQ to objects, combining the readability and simplicity of LINQ with the power of parallel programming. It can speed up LINQ query execution by ensuring that all computer capabilities available are used.

Apart from all this, there are other APIs and models in .Net to improve and enhance it. Over the course of this guide, we will be looking at some of the above in detail.

.NET Common Language Runtime (CLR)

This is a runtime environment used for managing and executing any code written in a language supported by .NET. It can turn code into native code so that the CPU can execute it. Its functions include:

- Conversion of programs into native code
- Exception handling
- Ensures type-safety
- Memory management

- Security
- Performance improvements
- Independent of any one language
- Independent of any one platform
- Provides garbage collection

It also provides language features, i.e. interfaces, overloading, and inheritance for object-oriented programming.

- **BCL Support –** Base Class Library is the library that provides .NET applications with class support
- **Thread Support –** management of parallel execution for multi-threaded applications
- **COM Marshaller –** provides communication between applications and COM objects
- **Type Checker –** checks types in the application to ensure they match the CLR standards
- **Code Manager –** manages the code at execution runtime
- **Garbage Collector –** releases any portions of unused memory, and allocates it to other applications
- **Exception Handler –** handles runtime exception so the application doesn't fail
- **ClassLoader –** loads the classes at runtime

.NET Framework Class Library

This is a large collection of classes, interfaces, namespaces, and value types used in .NET applications. The large number of classes support:

- Data types – base and user-defined
- Exception handling
- I/O and stream operations
- Communication with the underlying operating system
- Data access
- Creation of GUI applications – Windows-based
- Creation of web-client and server applications
- Creation of web services.

.NET Framework Class Library Namespaces

These are the namespaces used most commonly and that contain interfaces and classes:

Namespace	Description
	Includes the common data types, arrays, string values, and methods for data conversion
.Data	
.Data.Common	
.Data.OleDb	
.Data.SqlClient	
.Data.SqlTypes	All used for database access, retrieving databases, and performing commands
.IO	
.DirectoryServices	
.IO.IsolatedStorage	Used for accessing, reading, and writing files
.Diagnostics	Used for debugging and application execution tracing
.Net	
.Net.Sockets	Used for communication over the web when P2P applications are created
.Windows.Forms	
.Windows.Forms.Design	Used for creating applications (Windows-based) that use Windows UI components
.Web	
.WebCaching	

.Web.UI

.Web.UI.Design

.Web.UI.WebControls

.Web.UI.HtmlControls

.Web.Configuration

.Web.Hosting

.Web.Mail

.Web.SessionState All used for the creation of ASP.NET apps that run over the internet

.Web.Services

.Web.Services.Description

.Web.Services.Configuration

.Web.Services.Discovery

.Web.Services.Protocols Used for the creation of XML Web services and components that are published over the internet

.Security

.Security.Permissions

.Security.Policy

.WebSecurity

.Security.Cryptography Used for the purposes of authentication, encryption, and authorization

.Xml

.Xml.Schema

.Xml.Serialization

.Xml.XPath

.Xml.Xsl	Used for the creation and access of XML files

.NET Framework Base Class Library

This is a subsystem of the framework used for providing CLR with library support so it runs as it should. The System namespace can be found here, along with the core types.

This gives you an idea of what .NET is all about. Now we can move on and install Visual Studio, the recommended IDE for C#.

Installing Visual Studio

In order to start programming in C#, you need Microsoft Visual Studio Community Edition. There are a number of different versions available, including Professional, Community Edition, Premium, and Ultimate. The Community Edition is fine for learning C# and, unlike the others, it's free. When you're finished learning, have a look at the other versions and decide whether the investment is worth it – the Professional version has much the same features as the Community Edition but Ultimate and Premium have a lot more.

The Community Edition is ideal for open source development, educational use. It is also ideal if you work for a company that has less than five developers, fewer than 250 computers, and makes less than $1 million in revenue.

That said, you can do commercial development quite easily with Visual Studio Community Edition so you can use it for developing games or creating software applications, which you can sell for cash without any penalties.

Great stuff!

Let's install it.

1. Head to https://visualstudio.microsoft.com/vs/community/ and hit the download button – note, it is also available for Mac OS so make sure you choose the right version.
2. Just follow the steps on the screen to install it, just as you would any other software. Be prepared - this can take some time, 20 minutes or more if your internet connection isn't the best. You will also likely see other packages being installed, such as .NET Framework. Visual Studio needs these – so when you are asked for permission, do say 'Yes'.
3. Once it's all nicely installed, you will be asked to create a Microsoft account. You may already have one in which case it's easy. What you can be reassured of is that Microsoft will NOT use your email address irresponsibly. The most you will get

from them is software update emails and none of the spam that other email providers push through.

Running Visual Studio

You can run Visual Studio in the same way that you would run any program on your computer. If you are given an icon on the desktop, just click it. Otherwise, head to your Start menu and you'll find it listed.

Well, that was easy, wasn't it? Shall we do a bit of coding, just to get you in the right frame of mind? This is all about getting used to how Visual Studio works – we're going to start with the requisite Hello World program. This is a nice simple program that prints a simple message but is enough to let you see the basics of how C# works. And it lets you have a go at compiling and running a program without introducing any bugs (hopefully!)

1. Open Visual Studio and click on File -> New Project. Or just click on the button on the Start Page that says "New Project"
2. A dialog box will appear; here, you specify the project type and give it a name. On the left of the box are a selection of template categories so click the one that reads "Visual C#"
3. Go to the top center of the screen where there is a list and select "Console Application Template". There is an empty template but the one we want for now has some code in it already – leave the other templates for the more complicated stuff.
4. Go to the bottom of the template and type a name in – mine is called "HelloWorld!" but you can name yours whatever you want – keep the name simple and informative and remember what it is.
5. Press on OK and your new project is set up.

A Brief Look at Visual Studio

Before we go any further, it's worth taking a look at what you see in front of you. You should have a Toolbox on the screen but you can ignore this for now. In the middle of the screen you should see text that begins with:

using System

This is the source code for your program and is what you will be working with – we'll take a closer look at it in a minute.

Project Explorer should be on the right side of the screen and this is where you will see all the files contained in your project and that includes the main file we'll be doing some work on shortly. This is called Program.cs and the .cs file extension tells you that it is a text file with C# code. To see the code in any of the files in Explorer, just double-click on them.

At the bottom of the screen is Error List. When you are writing C# code, if you make a mistake and the computer can't work out what it should mean, you will see an error in that part of the screen – it's very helpful as it tells you exactly where the error is and what is wrong with it so you can figure out how to fix it.

At the bottom right of the screen, you may see a window called Properties – this shows you all the properties associated with an item or file you select – don't worry about this right now.

If you want a window on view that you can't see, click on View -> Other Windows and click the window you want. Right now, all you need is the main editor windows and the Program.cs code showing in it.

Saving Your Project

Your project has not been saved automatically; this is important – if you were to spend ages writing a load of code, editing it, compiling it, and then running it, only to have it crash, you would lose everything. Get into the habit of saving your program! Click on File -> Save All and save it in the default directory called Projects.

Modifying Your Project

Right! Now down to work.

In your Editor window, you should see this code:

using System;

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

namespace HelloWorld
{
class Program
{
static void Main(string[] args)
{
}
}
}
```

I'll talk you through what all that does in a minute but first, we're going to make a change and tell the code that we want it to print "Hello World!".

Go to the lines in the middle of the code that reads:

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

In between the opening and closing curly bracket ({}) add the following line of code:

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

Your code should now look like:

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

namespace HelloWorld
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine("Hello World!");
}
}
}
```

That’s it, your very first C# program – how easy was that!

Compiling and Running Your Project

Remember – right now your computer is not magic and it will not understand a word you have written. What it will understand are instructions given to it in binary format, a series of 0’s and 1’s. Visual Studio contains a compiler which takes your code and transforms it into binary so your computer can read it.

That’s our next step – compile the code and run it.

It is easy, I promise you.

Press the F5 key on your keyboard or click on Debug -> Start Debugging in the menu.

This will run the program in Debug Mode so, if anything happened while it was running, it wouldn’t just crash; Visual Studio would take care of it, stop the program and show you what the error is so you can fix it.

That’s it, you created a program, you compiled it, and you executed it.

If your program does not compile and execute, make sure it looks exactly the same as the code I showed you.

But wait a minute! Your program is running but you can’t see what it did because it just disappeared off the screen, that little black console window appearing for a Nano-second before going away. What happened?

Remember I said that computers weren’t all that intelligent? Well, that’s what happened; your computer ran that program and, because it didn’t have anything else to do because you hadn’t given it any more instructions, it just closed itself down. We don’t want it to do that so we need to work on a solution.

There are two ways, each having its own set of strengths and weaknesses.

Solution 1

Run the programming without debugging it – if you do this, it will pause before it shuts down. Click on Debug -> Start Without Debugging or press on CTRL+F5. Try it; you will see the “Hello World!” message printed on your screen and you will see another message, this one saying, “Press any key to continue”. Do that and the program will close.

This is not without its disadvantages. Because we are not running the program in debug mode, if anything were to happen to it, the application would crash and you would lose it.

So what’s the alternative?

Solution 2

Add another line of code. This new line will tell the program to wait before it shuts down. That line of code goes below the line that reads:

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

So add in this line now:

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

Now your code should read:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace HelloWorld

{

class Program

{

static void Main(string[] args)
{
Console.WriteLine("Hello World!");
Console.ReadKey();
}
}
}
```

Now we've added yet another line to the code, which you will find with every console app you write, and this can start to get a little annoying but you do have the added bonus of running your program in debug mode, which is a great feature, as you will discover.

Bottom line

This really is only going to be an issue when you write a console app, which is pretty much what we are going to be doing most of the time.

Let's take a look at what you've written and what it all means. I'll go through each line of the code and what it means so that you at least have a basic understanding of what's going on.

***Using* Statements**

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

Notice that the first four lines of the code all start with the word "using". This is a keyword (more about them later) which is a word reserved for use by the C# language. In the C# compiler, it has a special meaning – the compiler knows that it has to do something special – use code that someone else has written and that we want to have access to.

Whenever you see a line of code that starts with something like “using System”, you know that the code is making use of a load of code called “system” that is written elsewhere. If we didn’t include this line, the compiler would have no idea where it should look to find specified things and your program would not be able to run. We have four of these statements in the code added by default and these can stay as they are for now.

Namespaces – A Brief Introduction

We’ll go into these in more detail later. As I mentioned, you, and anyone else can create code and access it with the “using” keyword and statement. To ensure that these codes are managed properly, all related code is placed into a namespace – there’s a good reason for the name, for later discussion.

For now, the next line in the code, after the “using” statements, is:

namespace HelloWorld

{

Curly braces are used in C# for enclosing related code sections called code blocks. You will see them all over the place in C# code. Every time you see an opening curly brace ({), it must be matched with a closing curly brace (}). This is one of the main reasons for errors so always check your code to make sure opening brackets have been correctly closed off. Anyway, I digress. Note the closing brace after the code line.

This code is letting us know that everything written between those curly braces belongs to a namespace called “Hello World!” and namespace is another keyword that gives the line a special meaning but the “Hello World!” part could be called anything – it will still work. In fact, it only says this because I called my program “Hello World!”. If you called yours something different, that’s the name you will see.

Class and Method Declarations

In the next part of the code, we have some class and method declarations. We are going to be mentioning these very often throughout this guide so all you need for now is just to have a basic understanding of them. This is the part of the code we are talking about now:

```
class Program
{
static void Main(string[] args)
{
```

And then to close it

```
}
}
```

Fundamentally, a class is something like a blueprint for a specific object type. Most of the code that you write will belong to one of these classes and, in our case, that class is called “Program”. As we go through this guide, you will be making a lot of classes because they are one of the most critical parts of C# and other OOP or Object-Oriented Programming languages.

Inside a class, we have methods – you will see these called functions in other languages. More about these later. For now, a method is a block of code that does a specific job. What our compiler is going to be looking for is the “Main” method.

The rest of what you see on that line:

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

This is known as the “method signature. This is where specific properties are defined for the method and these are also vital to the C# compiler when it comes to finding the “Main” method – do NOT modify these in any way.

We’ll talk about these later in the guide. What you need to know right now is that the compiler will look for this line so it knows where your program starts.

Anything that goes in the “Main” method is run by the computer and that is the method we will be focusing on for a little while.

What’s Next?

Wow, a lot of work and we haven’t got much done yet. That’s fine, we have to start somewhere.

Before we take any further steps into programming, I want to talk about another important feature – comments. Comments are used for making notes about parts of your code - so let’s look at them now.

C# Comments

Comments are a way for you to write notes on your code to tell you what it is doing and, written correctly, the computer will ignore them. To write a comment, you simply add a pair of slashes (//) in front of the comment and, if you want to add a comment that goes over several lines, you use slashes and asterisks – for example

/ this is a comment*

that goes over multiple

lines/*

What we are going to look at is what they are, why and how you need to use them. There are three ways. Many programming tutorials either ignore these or they hide them away toward the back of the book. I choose to put them under your nose right at the start – that's how important they are!

So, what is a comment? Basically, it is a piece of text that the human eye can read and that the computer ignores. You should get into the habit of using them in your code. They are useful to describe what a piece of code does – you might come back to it later or someone else might read it, and comments will tell you exactly what's what.

However, it isn't enough just to write any old text. Your comments have to be good because they are the only explanation you have for code that is potentially tricky. It is the main way that a coder will communicate with other coders who look at their work, be it five minutes after you write it or five years.

How to Write Comments in C#

Basically, there are three ways to do this. We'll only look at two because the third way doesn't apply to anything we will be doing for some time.

The first way is to use a pair of slashes at the start of the comment and anything that follows it is considered part of the comment and ignored by the computer. You will note that, in Visual Studio, when you write a comment correctly it changes color so you can see it clearly.

For example:

```
// This is a comment, where I can tell you what is going to happen...
Console.WriteLine("Hello World!");
```

You can write your comment on the same line as your code:

```
Console.WriteLine("Hello World!"); // This is a comment too.
```

And anything on that line after the // is ignored.

The second way is to use a slash and an asterisk:

```
Console.WriteLine("Hi!"); /* This comment stops here... */
Console.WriteLine("and you can add more stuff here");
```

We also use this method for multi-line comments like this:

```
/* This is a multi-line comment.
It goes over several lines.
Isn't it cool? */
```

You could use the // method for multiline comments:

```
//  This is a multi-line comment.
//  It goes over several lines.
//  Isn't it cool?
```

That seems to be the most preferred method but you must use whatever suits you – both are perfectly valid.

Another type of comment starts with three slashes (///) and is used for the automatic creation of technical for the code. It is only used in certain places though - so, when we get to them, I will go into it in more detail.

Writing Good Comments

Commenting is very easy; making good comments, not so much. To start with, your comments should be added at the time you write your code, not hours later. You might have forgotten what you were doing and this happens even with experienced programmers.

Second, your comments should add good value to your code. This is a bad example:

```
// Resets the score to 0

score = 0;
```

The code itself tells us that so the comment is a waste of time and effort. What it should read is something like this:

```
//Resets the score to 0 because the game is beginning again

score = 0;
```

There isn’t a need to add comments to every line of code but try to have one for each related section. You can over-do things but this is better than not having any comments at all – get into the habit of doing it.

One more thing before we head into the real programming – keywords.

C# Keywords

Like all computer programming languages, C# has its own set of reserved keywords. These words have a special meaning to the compiler and are reserved for use by the compiler only. This means you cannot use them when you are naming variables, classes, interfaces, or anything else.

The C# keywords fall under these categories:

Modifier Keywords

These keywords indicate who can modify the types and type members. They are used to prevent or allow parts of the program to be modified by other parts. The keywords are:

abstract

async

const

event

extern

new

override

partial

readonly

sealed

static

unsafe

virtual

volatile

Access Modifier Keywords

These keywords are applied to the class, method, property, field and other member declarations and are used for defining class and class member accessibility. The keywords and their meanings are:

public – this modifier will allow parts of the program within the same or another assembly access to the type and the type members.

private – this modifier places restrictions on other program parts from accessing the type and type members – access is granted only to code from the same class or same struct.

internal – this modifier will allow code from the same assembly access to the type or type members. If no modifier is specified, this is the default.

protected – this modifier allows code from the same class or from a derived class (of that class) access to the type and type members

Statement Keywords

These keywords relate entirely to program flow:

if

else

switch

case

do

for

foreach

in

while

break

continue

default

goto

return

yield

throw

try

catch

finally

checked

unchecked

fixed

lock

Method Parameter Keywords

These are keywords that are applied to method parameters:

params

ref

out

Namespace Keywords

These are the keywords that are applied with namespaces and the related operators:

using

. operator

:: operator

extern alias

Operator Keywords

The operator keywords are used for miscellaneous operations:

as

await

is

new

sizeof

typeof

stackalloc

checked

unchecked

Access Keywords

These keywords are used for accessing the containing or base class of a class or object:

base

this

Literal Keywords

These keywords apply to the current value or instance of an object:

null

false

true

value

void

Type Keywords

These are used for the data types:

bool

byte

char

class

decimal

double

enum

float

int

long

sbyte

short

string

struct

uint

ulong

ushort

Contextual Keywords

These are only considered as keywords when they are used in specific contexts. They are not classed as reserved and so you can use them for names or identifiers:

add

var

dynamic

global

set

value

When you use one of the contextual keywords on Visual Studio code, they will not change to the blue color, which is the default color in VS.

Query Keywords

These are contextual keywords for LINQ Queries:

from

where

select

group

into

orderby

join

let

in

on

equals

by

ascending

descending

These may not be used as identifiers or names but you can use them with the @ prefix. For example, because ‘class’ is a reserved keyword you cannot use it in names or identifiers. However, you can use @class – see the example below:

```
public class @class
{
public static int MyProperty { get; set; }
}

@class.MyProperty = 100;
```

What’s Next?

Now that you have the basics under your belt, we can move on and start delving into the actual programming topics. We’ll start with variables.

Variables

Variables are an incredibly important fundamental of C# programming - well, of any programming really. We use variables to store data and when you write a program that does something real, it will use those variables. We'll be looking at what they are, how to create them, and how to add to them, as well as looking at all the different variable types.

What are they?

One of the most important parts of any language and any program you write in that language is being able to store information and data. For example, if you were writing a game, you might need to store data on how many lives a specific player had left or their game score.

From your math classes, you might remember discussing variables but these would have been unknown quantities that you needed to solve. With programming, things are different. Your variable is a box, something that you put things into. You can put anything you like into the box so that it contains varying or variable contents, hence the name.

Like most programming languages C# has many of these boxes, each one able to store a different kind of data, such as characters, numbers, and so on. It may (or may not) be of interest to you but languages like this are known as "strongly typed" because they have so many boxes. Those languages that have just one box that can hold anything are called "weakly typed".

We'll look at the different types of variables in C# but first, we need to see how to create one.

Creating Variables

The best way to learn how to create a variable is to actually do it. So go ahead and open Visual Studio and create a new console project (refer back to the first chapter if you can't remember how to do this).

Bring up the Main method on the screen and add this line of code to it:

int score;

That’s a variable. Not much to it really. There’s a little more to it than that though. Creating a variable or defining it as it is known, requires two steps. The first step is to indicate the variable type and C# has specific keywords used for doing that.

Remember; the variable type will determine what can be stored inside it. In our case, we used an int type which indicates that the variable can store integer values. An integer value is a whole number and their corresponding negatives – o, 1, 2, 3 4 … and -1, -2, -3, -4 and so on. In the variable type that we defined, we can store 100 or we could store 2578, for example, but we cannot store 2.854 because it is not an integer. We also could store “hotdog” because that isn’t an integer either – more about types shortly.

The second part is to provide your variable with a name. Just keep in mind that this name is really only for human use – the computer really doesn’t care what the name is and once you pass it to the computer it will rename it to a memory location only. Make sure you choose a name that makes sense for what is going into the variable.

In math terms, variables are often named with a single letter, for example, ‘X’. In computer programming, variable names are more precise, for example, “score” Also, every statement in C# ends with a semi-colon (;), which is how the computer knows that it is the end of the statement.

Assigning a Value to a Variable

Once you have declared the type and the name of the variable, you need to give it a value. We call this “assigning a value” and we use the assignment operator to do it (=). We’ll be discussing operators later in the guide but, for now, add this line underneath the last one you input:

score = 0;

We'll get onto basic math in the next part of the guide but, for now, understand that the = sign in programming means something different to the = sign in math. When you see a statement like 'score = 0', try to think of it as meaning "score is assigned a value of 0" and not 'score equals 0'.

You can assign any value that you want to score, for example:

score = 2;

score = 17;

score = -3587;

One more thing you can do, to make life easier for yourself is to define a variable and assign a value to it in one go. For example:

int theTreeOfLife = 75;

In one simple statement, we defined a variable called theTreeOfLife, gave it an int so it holds integer numbers, and assigned it a value of 75.

Types of Variables

Integers are just one type of variable. Let's look at the other types that you will use at one time or another in a C# program, as well as discuss when you would use them and what the differences between them are:

Type	**Description**	**Bytes**	**Data Range**
short	stores small integers	2	-32,768 to 32,767
int	stores medium integers	4	-2,147,483,648 to 2,147,483,647
long	stores large integers	8	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	stores small unsigned	1	0 to 255

	integers (no + or-)		
ushort	stores small unsigned integers	2	0 to 65,535
uint	stores medium unsigned integers	4	0 to 4,294,967,295
ulong	store large unsigned integers	8	0 to 18,446,744,073,709,551,615
float	stores small real numbers (floating point)	4	±1.5 × 10^−45 to ±3.4 × 10^38, 7 digits of accuracy
double	stores large real numbers	8	±5.0 × 10^−324 to ±1.7 × 10^308, 15 to 16 digits of accuracy
decimal	small range real numbers high accuracy	16	±1.0 × 10^−28 to ±7.9 × 10^28, 28 to 29 digits of accuracy
char	stores one character/letter	2	any character
string	store a sequence of letters, numbers or symbols	any	sequence, any character, any length
bool	stores true or false values	1	true or false

Let’s see if we can explain this a little better – starting with the three types of integers at the top – short, int, long. The higher the number of bytes, the larger the number can go to but that equates to more memory. Before you decide which one to use, you have to know the size of the number you may be storing. Most of the time, you will probably use int but this won’t always be the case.

Next, note that each of these three int types also has an unsigned equivalent – ushort, uint, ulong. None of these require a sign and every value is assumed to be a positive value. This means that these types can store numbers that are twice as large as their signed equivalent and, although you can’t use negative numbers, you can go to 0.

The type of byte is just one single byte and is also unsigned. You might think that this is quite limiting because it can only go up as far as 255. However, we also use this type for pushing raw bytes that have no meaning, at least as far as the computer is concerned. For example, this could be the contents of a file or an image, data that is being transmitted across networks, which could integers, or any other type of information. Whatever it is, the program doesn’t really care about it at the moment.

Next, we have the float, decimal, and double types. These store what is known as real numbers in the math world but, in computer programming terms, they are floating point numbers. All the other types we talked about can store a number of 7 but not 4.5. A fractional number is one that has a decimal point and it must be stored in a variable that has a floating point type. For smaller numbers that don’t need much accuracy, you can use the float type. For larger numbers that need more accuracy, there’s the double type. The decimal type has a much smaller range than these two but has far more accuracy. Depending on your requirements, you will need to make a choice between them. If, for example, you are going to develop a game, you won’t see the decimal type very often – float and double are used a whole lot more.

For text, we have the char and string types. The char type stores one single character or letter while the string type stores a sequence of letters, symbols, and numbers.

Lastly, we use the bool type for logic. You will be seeing quite a lot of this in the guide as this is the type we use for determining if something is true or false and then, depending on which one, to do different things.

You can write any one of these variables in much the same way as we did earlier in the guide, using Console.WriteLine, like this:

```
int score = 75;
Console.WriteLine(score);

bool levelComplete = false;
Console.WriteLine(levelComplete);

string message = "Hello World!";
Console.WriteLine(message);
```

Writing a Good Variable Name

Before we end this section, we need to have a chat about how you name your variables. There is much of debate on how this should be done so I want to give you my take on it.

The idea behind the name is to make it easy to know what the variable is for, not just by you but by anyone who looks at your code. It’s no good giving it a name only to find that six months later you can’t for the life of you remember what it was for because the name meant nothing. Writing readable code includes writing good variable names and you cannot ignore this.

To start with, there are a number of rules built-in to C# programming regarding naming variables. Every variable name must start with a letter, either a lower-case or an upper-case letter. After that, you can have any combination of letters, numbers, and the _ (underscore) character. Do anything else and the compiler will throw a hissy fit and will refuse to compile your program.

Most importantly is naming your variables in a way that describes what goes into them. For example, if you are storing game scores in a variable, call it something like “score”, “playerScore” or something like it. Don’t call it “ps” or “tiger” or anything that is meaningless.

Don’t use abbreviations if you can help it. For example, if you called your variable “plscr”, would you remember what it meant later on? Will you have to sit and really think hard about it? You are not playing a word search game or a fill-in-the-missing-blanks game – if you have to guess, don’t do it.

The name of a variable should have a minimum of three letters (although there are exceptions) so don’t worry if your name is longer. It’s better to have a name that properly describes it and we no longer have the old limit of eight characters for the name. However, don’t go too mad and have a name that is too long.

If you find yourself adding numbers to the end of your name, you are doing something wrong. Think about renaming your variables instead. For example, rather than score1 and score2, consider something like playerScore and gameScore. Do you see where I’m going? Also, don’t use words like ‘text’, ‘data’, ‘number’, or ‘item’ as they don’t offer enough description.

Mostly, we use nouns to name a variable, such as a person, a thing, a place, an idea or use a noun phrase, such as ‘playerScore’.

Ensure that each word in your variable name stands out from the next one. This makes it infinitely more readable and, because spaces are not allowed, there are a couple of ways you can do this. The first is to capitalize second and subsequent words in the name, leaving the first with a lower-case – playerScore, for example. The second is to use an underscore, for example, player_score. Both of these look far better than playerscore and are more readable. Either is acceptable, although I personally prefer the first one.

A Whole Example

This example code shows you the ways that the variables are defined and assigned:

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

namespace Variables
{
class Program
{
static void Main(string[] args)
{
// This will declare the variable and assign them
// in one step.
short levelNumber = 4;
int score = 0;

// This will declare the variable and then assign it
// later on.
long aBigNumber;
aBigNumber = -19;

// This one is a byte and can contain any value
// between 0 and 255 inclusive.
byte aSingleByte = 85;

// These are the unsigned variable types.
ushort thisHoldsPositivesOnly = 65;
uint soDoesThis = 806587;
ulong andThisButBiggerNumbers = 587695028;

// These are the variables that hold text
char myFavoriteLetter = ‘d’; // This is my favorite because it is for dog.
string aMessage = "You have a new message on your private terminal.";

// And lastly, the  Boolean type.
bool stillWorking = true;
stillWorking = false;
}
}
}
```

Quick Primer

- Variables can store all kinds of things.
- The byte, short, int, and long types can store integer values that are 1, 2, 4, and 8 bytes respectively.
- The float and double types can store floating point values of 4 and 8 bytes respectively.
- The decimal type can store a floating point number of 16 bytes, and with a smaller range and more accuracy than the double type.
- The char type stores one character.
- The string type stores text values.
- The bool type can store a value of true or false.
- Variables are declared by stating a type and name, for example, int aNumber;

- The name must begin with a letter, lower, or upper-case, and may contain numbers, letters, and the _(underscore) and can be of any length.
- The = sign is used to assign a value to a variable, for example, int aNumber = 8;
- Variables can be declared and assigned in one statement, for example, int aNumber = 8;

What's Next?

Now you have an understanding of variables and how they work. We can move on and start doing some proper work. First, we look at some basic math.

Basic Math

If you are familiar with any other computer programming language then you'll find C# arithmetic very easy to grasp, especially as it is much the same as C, C++, and Java.

Now that we have a grasp on variables, we can delve into math; it's pretty simple, basic arithmetic, much like what you learned at school. Let's face it, computers just love math, mainly because it's just about all they can do and they can do it fast.

What we are going to look at here is some of the more basic C# arithmetic – we'll be going deeper in another chapter. We'll start with addition, subtraction and move on to multiplication, division, and the modulus operator, which is a new one. From there, we'll look at numbers (positive and negative), the order of operations (operator precedence), and then finish by looking at compound assignment operators – these can do the math and assign values in one swift operation.

Addition, Subtraction, Multiplication, and Division

This bit is straightforward, even if you have never looked at another computer programming language. We'll start with a basic math problem, one that you most likely did in first or second grade. We are going to add two numbers together and store the value in a variable with the name of 'a'. Here's the code:

int a = 4 + 5;

The same principle works for subtraction, this time the value is stored in a variable named 'b':

int b = 5 - 4;

You are not limited to using numbers only in your math; you can also use variables:

int c = a + 3;

int d = a - b;

It doesn’t stop there; we can also chain several operations together in one line:

```
int sum = 1 + 2 - 3 + 4 - 5 + a - b + c - d;
```

Division and multiplication work in much the same way although there are a couple of things you need to be aware of when doing division with integers. We’ll look at that more a bit later; for now, we’ll stick with the double or float type with division rather than using integers. The multiplication sign in C# is *, which is the asterisk symbol and the division sign is the forward slash, /.

```
float totalCost = 22.54f;

float tipPercent = 0.18f; // this is the equivalent of 18%

float tipAmount = totalCost * tipPercent;

double moneyMadeFromProgram = 100000; //  Note – commas are not required in the number because C# won’t have a clue what to do with it.

double totalProgrammers = 4;

double moneyPerPerson = moneyMadeFromProgram / totalProgrammers; // We're mega rich!
```

Remember the first program we did? Hello World!? We used the Console.WriteLine code to show a user some text and we can do that with numbers as well:

```
// The area of a circle has a formula of pi * r ^ 2

float radius = 4;

float pi = 3.1415926536f;  // The 'f' ensures it is treated as a float and not a double.

float area = pi * radius * radius;


// Note that, when we use a ‘+’ symbol with a text string, the number at the end is concatenated to the end of the text string.
```

Console.WriteLine("The area of a circle is " + area);

In this example, we have used three variables: one for the circle radius, one for the value of PI, and one for the area, all calculated using a standard formula. The result is then printed or displayed for the user.

Note that we used the '+' operator with our text strings. This is a truly great feature in C# because the language knows that strings can't be added mathematically but there is a way of handling it intelligently – simply concatenate or join the text strings together so one follows another. If you were to run the code above, what you would see displayed is "The area of a circle is 50.26548". There is a bit more going on here but we'll talk about it in a later tutorial.

Modulus Operator

Back in your school days after learning division, you would do something like "23 divided by 7 is 3 with a remainder of 2". In computer programming, there is a special operator for that remainder calculation and it's called the modulus operator, often just called the 'mod'. The operator will only give you the remainder of the division calculation, not the actual division result. The percentage symbol (%) is the sign used for the modulus so, wherever you see the % in C# programming, remember that it does not mean percentage, it means 'get the remainder'.

Just to refresh, 7 goes into 23 3 times with a remainder of 2. This is how you would do this in your C# code, assuming that we are dividing 23 oranges between 7 people:

int totalOranges = 23;

int people = 7;

int remainingOranges = totalOranges % people; // this is 2.

At first look, you might think it isn't that useful an operator but once you learn how to use it, I guarantee you will find plenty of ways.

The Unary "+" and "-" Operators

You have seen these operators already. Unary means that the operator has just a single number or operand to work with. So far, everything we have used has had two operands, making them binary operators. For example, with 4 + 5, the + operator is working with the number that comes before and the one that comes after it.

The unary operator works only on the number that follows it. Here's an example; it will probably make more sense to you:

// These are the unary '+' and unary '-' operators. They will only work on the number that follows them.

int a = +7;

int b = -36;

// The binary '+' and binary '-' operators are also known as addition and subtraction, work on the numbers that come before them and after them.

int a = 4 + 5;

int b = 5 - 22;

The unary signs, the + and -, are doing nothing more than signifying whether the number they prefix is a plus or a minus. There is one more, but we'll look at that later.

Parentheses and Order of Operations

Do you remember learning about the order of operations in math at school? Well, computer programming also has this and the same rules apply. Just to refresh your memory, as far as any mathematical formula or equation goes, the calculations inside the parentheses are always done first, followed by powers, then exponents. As an aside, there is no power operator built-in to C# but we'll talk about that in another tutorial. Following that are the multiplication, division, addition, and subtraction, in that order – that is the exact same order that C# follows.

We can use parentheses to indicate that a specified bit of the equation is to be done before any other. For example:

```
// A piece of easy code to calculate the area of a trapezoid
// If you don’t remember this, don’t worry.
double side1 = 5.5;
double side2 = 3.25;
double height = 4.6;

double areaOfTrapezoid = (side1 + side2) / 2 * height;
```

If you have a complicated formula in math, one that has lots of sets of parentheses, you may find yourself using the square brackets ([]) or the curly braces ({}) as parentheses that are a bit more powerful. In C#, we don’t do it like that. Instead, you just use as many sets of parentheses, nested or otherwise, as you need, so long as each set is a complete set – remember, every opening bracket or parentheses requires a matching closing one.

Here’s an example:

```
// This isn’t a real formula; it is made up for the purposes of this
guide...
double a =5.1;
double b = -6.5;
double c = 87;
double d = -56;
double e = 5.487;

double result = (((a + b) * (c - 4)) + d) * e;
```

As an aside, I want to go over something I mentioned earlier why the equal sign is not used to mean that one thing is equal to another. Not directly, anyway. The = sign is the assignment operator and what it actually means is that whatever is on the right side of it is calculated and the result placed into a variable that is on the left side.

Let’s see how it works:

int a = 7;

a = a + 2; // the variable called a will have a value of 9 once this line has been run.

Mathematicians will look at that and think it looks strange. To all intents and purposes, a variable cannot possible equally itself plus 2. Just as an exercise, subtract a from both sides and it reads o = 2. From your point of view, this does not make sense.

From a programming point of view, it makes perfect sense. What we are saying is, take the value that is in a (7), add 2, making 9 and then assign that new value back to a. All we’ve done is taken the a variable and added two to it.

Compound Assignment Operators

What we talked about there, taking a value, doing something with it and then putting it back in the original variable, is actually very common in C#, so much so that a whole set of operators have been set aside just for it. These are known as compound operators – they can do the math function and assign values at the same time.

Let’s see how they work:

int a = 7;

a += 2; // This is the equivalent of a = a + 2;

There are also similar operators for the subtraction operator, division, multiplication, and modulus:

There are also equivalent ones for subtraction, multiplication, division, and the modulus:

int b = 8;

b -= 4; // This is the equivalent of b = b – 4; at this stage, b would be 4.

*b *= 5; // This is the equivalent of b = b * 5; at this stage, b would be 20.*

b /= 4; // This is the equivalent of b = b / 4; at this stage b would be 5.

b %= 2; // This is the equivalent of b = b % 2; at this stage, b would be 1.

Got it? We'll be discussing more math equations and calculations in a later chapter.

Quick Primer

- Addition works like int a = 4 + 5;
- Subtraction works like int b = 6 – 3;
- Multiplication has the * as its sign and is used like this – int c = 5 * 5;
- Division has the / as its sign and is used like this – int d = 8 / 4;
- The modulus operation uses a % as its sign and is for determining the
  remainder after a division operation, like this – int e = 22 % 7; (int e
  would have a value of 1).
- When doing math in C#, any of the standard types for numbers can be
  used – double, int, float, short, etc.
- Multiple operations can be combined into one and you can also use a
  variable you created earlier in your statement – int f = 3 * b – 4;
- Order of operations is very important and it works the same as you learned
  in school. Multiplication and division come before addition and
  subtraction, working left to right. Parentheses come before anything so use

them to make a calculation happen in the order you want, for example,

double m = (y2 – y1)/ (x2 – x1)

- The compound assignment operators are +=, -=, /= and %=. They do the operation, assigning the value to the variable on the left. For example,

  a += 3; is the equivalent of a = a +3; and is doing nothing more than adding 3 to the value already in a.

What's Next?

We discussed basic math here but there is much more to learn. We're going to take a slight detour next and look at user input using the command prompt, just so you know how it all works before we go back to our math class.

User Input

This is just a short section; we're going to take a quick look at a couple of pieces of code that we need to use for creating a program with some value. First, we'll look at the console window to see how we can use it to get some user input and then we'll look at how it can be converted into the correct data type. Lastly, we'll put it all together with everything else we've learned in one program that will tell us what the volume and the surface area of a cylinder is.

User Input from the Console

We can use the next statement to get some input from our user:

string whatTheUserTyped = Console.ReadLine();

This uses a method (we'll talk about these in a later chapter) that will read a text line supplied by the user. The text from the user is placed into a variable on the left named whatTheUserTyped. Remember that any variable can go there, this one is nothing special.

Converting

Most of the time, just getting the input from the user and putting it into the string variable isn't enough to give us the correct data type for our needs. What if, for example, we wanted a number from the user? When it gets put into the string, we can't do any math calculation with it unless we convert it to the correct data type first. Don't misunderstand me; on occasion, string will be the correct type so you won't always have to change it.

Converting to another type requires a method that will convert a data type into another type. I won't go into details of this just yet; that comes later. Suffice to say, for now, if we wanted to convert our string to an integer, we would need this:

int aNumber = Convert.ToInt32(whatTheUserTyped);

Each possible data type has a corresponding similar method. For example, ToBoolean converts to the bool type; ToInt16 converts to the short type, ToDouble is for the double and so on. There is one tricky one – the float type. For this, you must use ToSingle because the float is a single-precision value. You will see this in operation in the program.

A Complete Program

Great, we've got some more bits in place and now we can create a program that actually does something. What we are going to make is a program that lets a user input two values – cylinder radius and cylinder height. Then a little math with all formulas provided for you to work out what the cylinder volume and surface area are. The results then get printed to the user.

Now, the best way for you to learn would be just to get on and see what you can come up with. Seriously, the best way to truly learn is to struggle through and try to put all the pieces together. That's not going to happen, not this early in the guide. Besides, this is a tutorial, designed for you to do at your own pace, not a school lesson.

Of course, you could go off now and use what you learned to build your program and then compare it with what I'm going to tell you. One thing you need to understand is this – if your program and mine don't exactly match, it doesn't matter – so long as the output is correct and the code is correct.

Anyway, this is the code:

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

namespace CylinderCalculator
{
class Program
{
```

```
static void Main(string[] args)
{
// Print a greeting message.
Console.WriteLine("Welcome to Cylinder Calculator !");

// Read cylinder's radius supplied by the user
Console.Write("Enter the cylinder's radius:  ");
string radiusAsAString = Console.ReadLine();
double radius = Convert.ToDouble(radiusAsAString);

// Read cylinder's height supplied by the user
Console.Write("Enter the cylinder's height:  ");
string heightAsAString = Console.ReadLine();
double height = Convert.ToDouble(heightAsAString);

double pi = 3.1415926536; // We'll look at a different way to do PI
later on

// These are the standard formulas for cylinder volume and surface
area.
double volume = pi * radius * radius * height;
double surfaceArea = 2 * pi * radius * (radius + height);

// Now the results can be output
Console.WriteLine("The cylinder's volume is:  " + volume + " cubic
units.");
Console.WriteLine("The cylinder's surface area is:  " + surfaceArea +
" square units.");
// Before you close, wait for a response from the user...
Console.ReadKey();
}
}
}
```

Let’s break this down.

The first bit should be very familiar to you, the part that has the using statements.

What we did was provide a welcome message for the user.

```
// Print a greeting message.
Console.WriteLine("Welcome to Cylinder Calculator!");
```

Pretty much the same as our Hello World! example at the beginning of this guide.

Next, we ask the user to input the height and the radius of the cylinder and then we convert the values to the correct type using what we discussed at the start of this section:

```
// Read cylinder's radius supplied by the user
Console.Write("Enter the cylinder's radius:  ");
string radiusAsAString = Console.ReadLine();
double radius = Convert.ToDouble(radiusAsAString);

// Read  cylinder's height supplied by the user
Console.Write("Enter the cylinder's height:  ");
string heightAsAString = Console.ReadLine();
double height = Convert.ToDouble(heightAsAString);
```

The first line is nothing more than simple input and, after that, we read the user’s text in and stored it inside a string object. The next bit turned it into the right data type, in our case, the double type and then we did the same with the height of the cylinder.

Next, the value of PI was defined. Then we use some of the math operations we looked at in the last section to calculate what the cylinder surface area and volume are.

```
double pi = 3.1415926536; // We’ll look at a different way to do PI later on.


// These are the standard formulas for cylinder volume and surface area.

double volume = pi * radius * radius * height;

double surfaceArea = 2 * pi * radius * (radius + height);
```

Last, the result is output to the user:

```
// Now the results can be output

Console.WriteLine("The cylinder's volume is:  " + volume + " cubic units.");

Console.WriteLine("The cylinder's surface area is:  " + surfaceArea + " square units.");


// Before you close, wait for a response from the user...

Console.ReadKey();
```

That is our very first useful program written in C#.

What’s Next?

Now we’ve learned enough to create a useful program, we can kick things up a notch and take things further. Next, we’re going back to math but in much more detail this time. Once we’ve done that, we can get stuck into logic, into getting our computer to make some choices about what it does.

Math Part 2

Now, this is quite an involved section so don't panic if you don't understand it all first time around – go over it as many times as you need to.

As I explained earlier, computers absolutely love math, which is great, because that is pretty much all they can do. We will be moving on to more exciting stuff soon, I promise but this is a very important subject to learn.

One thing I do need to warn you is that the sections here are separate, not really related to one another at all. This is good because, if you need help on one specific thing, you can just jump straight to that section.

What are we going to discuss?

We'll start off doing some division using integers, covering a very interesting problem that arises from it. Then we'll move on to data conversion in a process called typecasting before we cover the subject of division by zero. After that, some of the special C# numbers, such is NaN, infinity, pi, and e, and then we'll cover over and underflow. To finish, we'll look at incrementing and decrementing, a very special C# feature.

Even if you think you know the subject, please don't be tempted to skip a section; it never hurts to have a refresher and especially do not skip the final section – you really will regret it!

Above all, although there is so much to learn here, do try and have fun. Programming doesn't have to be strait-laced, nose-to-the-grindstone theory.

Let's get started.

Integer Division

A good way to begin is with a thought problem. Work out, in your head, what the result of seven divided by two (7/2) is. Done it? Seriously, don't skip this and move on. Get that answer in your head

– it is a simple one. You should have one of two answers – 3.5 or 3 with a remainder of 1. Both of these are right.

Now, open C# and input the following code in the window – look at the result to see what happens:

int a = 7;

int b = 2;

int result = a / b;

See it? 3. Just 3. Not 3 with a remainder of 1 and not 3.5. Just plain and simple, 3. Why? This is not your standard division, not the stuff you learned in school anyway. Instead, it is called integer division and, in that, fractional numbers do not exist.

How does integer division work?

It takes the biggest number that it can evenly divide and, regardless of how close it may be, it will ignore the fractional bit. For example, 99/100 would be 0, even though the actual answer is 0.99.

When you do any division using the int type, integer division comes into play, as it does when you use the long, short or byte type, or an unsigned equivalent – ulong, ushort, or ubyte.

When you mix up the data types, things start to get a bit tricky:

int a = 7;

int b = 2;

float c = 4;

float d = 3;

float result = a / b + c / d;

In this, a/b is 3 as it was before. However, c/d takes care of the floating point division, which is the division you are familiar with, providing a result of 1.33333. Adding both together provides a result of 4.33333.

One thing that is very important is that you understand this is what is happening when you use integer types. It may not always be the right thing to do, but you can use it to your advantage as long as you understand it.

Typecasting

Usually, when you use two things of the same type in a math calculation, like two ints, for example, the result will be of the same type. But what about doing a math calculation using two types that are not the same? What if, for example, you wanted to add an int and a long type? The answer lies in typecasting or casting for short.

Casting is a great C# feature because it lets you convert one type to another type. C# has two types of casting – implicit and explicit. Implicit casting is automatic, magic if you like, where the conversion happens without you needing to do anything. Explicit, on the other hand, is where you have to indicate that you want the conversion or it won't be done.

In general, implicit casting happens automatically when you are going from a narrow to a wider type. What does that even mean? Go back to the section on types and look over the byte size. In C#, an int will use 32 bits and a long will use 64 bits. C# will implicitly cast ints to longs wherever it thinks they need to be cast without you telling it to do it.

So, you could do this:

int a = 8521;

long b = 159842965;

long sum = a + b;

When the adding is done, a is taken and cast to a long; it is added to b, and then put back into the variable called sum.

A floating point type is wider than an integer type, so, where needed, C# converts or casts ints to floats.

For example:

```
int a = 7;
float b = 2; // this converts the int value of 2 to a float value of 2.0

// because the value of b is a floating point value, the value of a is
implicitly
// cast to a float, and, rather than integer division, we now have
floating point division
// and this means that the result will now have 3.5 as a value,
// rather than 3 as the integer division would do, even though
// an int type was used.
float result = a / b;
```

Still with me?

Explicit casting is also incredibly important. There will be times when you want to go from a wide to a narrow type and doing this is a simple case of putting the required type to cast to inside a set of parentheses before what you are casting. Let’s see this in an example, to make it clearer; here, we cast a long to an int:

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

What is worth bearing in mind is that casting really isn’t magic and it can’t turn anything into anything. Not all types can be converted to another type and casting does have its limits. Try to explicitly cast something that can’t be cast and the compiler will throw up an error. Later, we’ll look at defining explicit casts but that is a bit more advanced for right now.

It should also be kept in mind that C# will always do casting first. That means if you want a math operation result cast, it must be included in parentheses with the cast in front of them. Here’s an example:

```
int a = 7;
int b = 2;
int c = 4;

// The answer is 7.5 because int be is converted to a float straight away,
// and then int a is implicitly cast to a float so the division can be done,
// then int c is cast to a float so the addition can be done.
float result1 = a / (float)b + c;

// Th answer is 7; a/b is done with integer division,
// which results in 3; the addition is them done with int c, providing the result of 7, and then
// lastly, the result is cast to a floating point value of 7.
float result2 = (float)(a / b + c);
```

Division By Zero

We all know that nothing good can come of dividing by zero; in terms of math, it just doesn’t work. What will happen in C# when you try to divide by zero? It is actually quite strange.

If you try dividing by zero with ints, an exception is thrown. Now, we’ll be discussing these in detail later but, for now, what it means is that your program will just stop, right there in its tracks if you are not running it in debug mode. You may see an error message, you may not. If you are in debug mode, Visual Studio activates at the line with the error so you should be able to see what went wrong and, hopefully, fix it. You will see a message that says “DivideByZeroException was Unhandled”.

What is interesting is if you are using a float or double, or any other floating point type, the program won’t crash. Instead, you get a value of Infinity which we will talk about right now.

C# has several special numbers – infinity, NaN, pi, e MaxValue and MinValue.

Infinity

The two types, float and double, define special values for infinity, both positive and negative, representing, as you would expect, positive and negative infinity. Remember, if you do any math with a number that is infinity, some strange things will happen. A positive infinity plus 1 will not be anything other than a positive infinity, as will positive infinity minus 1.

Pick one of these two options to use:

double a = double.PositiveInfinity;

float b = float.PositiveInfinity;

or:

double a = Double.PositiveInfinity;

float b = Single.PositiveInfinity;

NaN

NaN stands for Not a Number and this is another one of the special values. This can also come up when you do something a little mad, such as trying to divide infinity with infinity. There is also a specific way to access the value:

double a = double.NaN;

float b = float.NaN;

double c = Double.NaN;

float d = Single.NaN;

E and PI

These are both special numbers that may be made use of a lot, depending on what you are doing. It is worth understanding a little about them right now although we will touch on them again later.

You can create a variable to hold these two values, but why bother? There is already one built-in that will do this for you. We’ll use a class called Math (more about classes later, at which point, we’ll be back to math again) and we use code like this:

```
double radius = 3;
double area = Math.PI * radius * radius;

// pi is more likely to be used than e.
double eSquared = Math.E * Math.E;
```

Do you remember that we created a pi variable a while back? There isn’t any need to do that anymore because, as you can see, there is already one.

MinValue and MaxValue

Let’s look at these two values. Most numeric types in C# will define both a MinValue and a MaxValue inside them and these are used to hold the minimum and maximum values the type is able to hold. These are accessed in the same way as infinity and NaN for the floating points:

```
int maximum1 = Int32.MaxValue;
int maximum2 = int.MaxValue;

int minimum1 = Int32.MinValue;
int minimum2 = int.MinValue;
```

Overflow and Underflow

So, let’s think about this. The short type has a value that can go up to 32767. What would happen if we did this:

```
short a = 30000;
short b = 30000;
```

short sum = a + b; // The sum is far too big to go into a short so what would happen?

When a math operation makes something go over the range for the specific value, it's called overflow and it is important that you understand what happens. With the integer type – short, long, int, and byte – the most significant of the bits are dropped which is strange because the computer will interpret this as wrapping around.

Look at this:

int aNumber = Int32.MaxValue;

aNumber = aNumber + 1;

Console.WriteLine(aNumber);// This prints a large negative number.

As far as floating point types go, things are a little bit different. Because these types both have PositiveInfinity and NegativeInfinity defined, they will become infinity, rather than wrapping around.

If you find overflow too much of a concern, you can run it in checked mode. With this, integer operations will not wrap around; instead, they throw exceptions. Again, we'll come back to that later.

Underflow is a similar thing that can happen with floating point types on occasion. Let's assume that you have an extremely large floating point number, let's say, 1,000,000,000,000,000,000,000,000, for example. As you know, this size of number can be stored by a float. Let's assume that we have an extremely small number, such as 0.0000000000000000001. That can also be stored as a float but if you were to add these two numbers together, you could not store the result as a float. The result would be 1,000,000,000,000,000,000,000,000,000.0000000000000000001.

Floats cannot be that large and that precise – numbers that large are rarely precise! Instead, when you add them together, the result would be 1,000,000,000,000,000,000,000,000,000 and that is probably close enough for many things. However, we are still losing information that is potentially valuable.

Imagine that you had a load of extremely small numbers and you were adding them. This is where underflow may be a problem. It isn't as common as overflow though and you will probably never encounter it, but you should still be aware of it.

Incrementing and Decrementing

Before we leave Math behind, at least for now, we need to discuss a very cool C# feature. You might have noticed that often I've tried to add 1 on to a value. You've seen two ways of doing it already:

int a = 3;

a = a + 1;

and:

int a = 3;

a += 1;

Well, here's another way:

int a = 3;

a++;

This is known as incrementing and the ++ is increment operator. You will see quite a few places to use this as you work through this guide.

It has a counterpart, --, which is the decrement operator and is used for subtracting 1 from a value.

There is one more thing to mention with these operators. They can be written in two ways. For example, you've seen a++ but you can also write it as ++a. Likewise, a -- can also be written as -- a. We call it a postfix notation if the operator is at the end and a prefix notation if it is at the beginning.

Lastly, you should understand that there is a difference between the two - the operation does something we will see often in this guide, and that is it returns a value. With the postfix notation, the more common one, the original value is returned. So, for a++, the returned value would be a. With the prefix notation, which is not quite so commonly used, the new value is returned. Here’s an example to help you understand it better:

int a = 3;

int b = ++a; // int a and int b will both be 4.

int c = 5;

int d = c++; // the original value, 5, is assigned to int d while int c is now 6..

Quick Primer

- When you do C# division with integers, it will follow the standard rules for integer division.
- Many times, implicit casting is done from narrow to wider types.
- Explicit casting from one type to another can be done by inserting the type you want to cast to inside a set of parentheses in front, for example, (float a = (float) 3.4948393;).
- Division by zero will result in an exception being thrown for int types and, for the floating point types, in infinity.
- Positive Infinity, NegativeInfinity, and NaN are all designed for the double and float types, for example, float.NaN and Double.Positive.Infinity.
- MinValue and MaxValue are for almost all numeric types and define the minimum and the maximum values for each type.
- Pi and e are both defined in the class called Math and can be accessed using float area = Math.PI * radius * radius;

- Some math operations can result in numbers that go over the range for the
  data type the number is stored in. For the int values, this will result in
  wrapping around and, for floating points, the result is either
  PositiveInfinity or NegativeInfinity.
- The increment operator (++) is used for adding 1 to a value and the
  decrement operator (--) is used to subtract 1 from a value. For example,
  int a = 3; a++; will result in a having a value of 4 while int a = 3; a --; will
  result in a having a value of 2.

What's Next?

That was quite a lot of math and you might well be feeling a little overwhelmed. Don’t forget, you don’t need to understand all of this thoroughly right now. You can come back at any time.

Next, we’ll look at decision-making, one of the most important parts of writing a computer program.

Decision-making

All computer programs that do anything considered as real work will need to make decisions somewhere along the line. The decisions will be based on whether specified conditions are met or not. In other words, specific parts of the code will only be run when a state or condition is met. The core of this decision-making process is a C# statement known as the ‘if statement’ as well as a few related ones. We’ll start with the basic if statement, look at the related ones and then move on to look at how we can use them to compare values. We’ll finish by looking at logical operators, special operators used for making more complex conditions.

The if Statement

Imagine a situation where you are a teacher; you have grades to assign and you want to know what grade each student should be given based on their test scores. This is easy to do in the real world and also provides us with an easy example that uses decision-making.

The process the teacher follows is to consider the score for a student; if it is 90 or above, an A grade is awarded; 80 to 90 is a B grade and so on, right down to outright fail. To do this, decisions must be made, and those decisions are based on specific conditions. When we say something is done conditionally, we mean that it is only done some of the time.

Some of the students get an A, some will get a B, some will get a C, and some will fail.

What we are going to do is create a program that takes the student score and uses it to determine the grade. We’ll start at the very beginning.

In C#, the if statement is used to let you check if two things are equal and it will look like this:

```
if(score == 100)

{
```

```
// The code in between the set of curly braces is executed when
// the condition within the set of parentheses is true.
Console.WriteLine("Perfect score!  You win!");
}
```

That is quite straightforward. An if statement has several parts, starting with the ‘if’ keyword. A set of parentheses follows containing the condition we want. Next, the set of curly braces is used to show the block of code that will run only if the condition is true.

As a complete program, this is what it might look like:

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

namespace DecisionMaking
{
class Program
{
static void Main(string[] args)
{
//Everything to this point, before we get to the if statement
// always be executed.
int score;

Console.Write("Enter your score: ");        // The text is written without
moving to the net line.
string scoreAsText = Console.ReadLine();
score = Convert.ToInt32(scoreAsText);

if (score == 100)
{
// What is here is the code block of the if statement and it will
// only be executed if the condition is met--
// in our case, if the score equals 100.

Console.WriteLine("Perfect score! You win!");
}

// Everything from here onwards, following the if statement
// will also be executed all of the time.

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

The else-statement

Suppose you want to do something but you also want to do something else if a condition is not met. Let's say that you want "You win!" printed for perfect scores of 100 but "You lose!" printed for any other score? We can do that by using the 'else' keyword:

```
if(score == 100)
{
// This code is executed if the condition is met.
Console.WriteLine("Perfect score! You win!");
}
else
{
// This code is executed if the condition is not met.
Console.WriteLine("Not a perfect score.  You lose.");
}
```

What's important here is that one of those code blocks will be executed while the other one won't be, so try using this to your advantage.

else-if-statements

We can do more with if and the else statements; we can string them together and use them in different ways:

```
if(score == 100)
{
Console.WriteLine("Perfect score!  You win!");
}
else if(score == 99)
{
Console.WriteLine("Only just missed it!.");
}
else if(score == 0)
{
Console.WriteLine("Seriously, you really must have tried hard to get
that score!");
}
else
{
```

```
Console.WriteLine("Ah, that’s just a little bit boring.");
Console.WriteLine("Seriously.  Next time come up with a nicer number.");
Console.WriteLine("OK, I am doing this just make sure that you are aware ");
Console.WriteLine("that you can use as many code lines as you need.");
}
```

Just one of the code blocks is going to be executed and which one it is will depend entirely on what the score is.

Relational Operators

==, !=, <, >, <=, >=

You now know the decision-making basics so we’ll move on to how you can specify conditions (the part that says score == 100).

The == sign is called a relational operator. That is just another way of saying it’s an operator that can compare two things.

The != operator looks to see if two values are not equal to one another and it works in much the same way as the == operator:

```
if(score != 100)
{
// Whatever is here is executed IF the score is //NOT// 100
}
```

Then we have the > and < operators. These are used to see if a value is greater than or less than another value and these work much the same way as you would use them in math:

```
if(score > 90)
{
// This only gets executed if the score is more than 90.
Console.WriteLine("You got an 'A'!");
}

if(score < 60)
{
// This only gets executed if the score is less than 60.
Console.WriteLine("You got an 'F'.  Sorry.");
}
```

You might have spotted that things have turned out a little different to what we originally wanted to do. We wanted an A grade to be assigned if the core was at over 90. Exactly 90 will not result in an A grade because it is not more than 90. That brings us neatly to the final two relational operators - >= and <=. These mean, respectively, greater than or equal to and less than or equal to:

```
if(score >= 90)
{
// This only gets executed if the score is 90 or above...
// Slightly different from the previous example because this one
// also picks up on 90.
Console.WriteLine("You got an 'A'!");
}
```

What you have seen here is enough to create the program that we wanted to write in the first place:

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

```
namespace DecisionMaking
{
class Program
{
static void Main(string[] args)
{
int score;

Console.Write("Enter your score: ");        // The text is written without
moving to the net line.
string scoreAsText = Console.ReadLine();
score = Convert.ToInt32(scoreAsText);

// This if-statement is separated from the others.
// This has nothing whatsoever to do with the blank line
// in between the statement and the line that reads 'if(score >= 90)',
// because that begins a new if statement.
if (score == 100)
{
Console.WriteLine("Perfect score!  You win!");
}

// This will check on each of the conditions in turn,
// until it finds the first one that evaluates true;
// at this point, it will execute the relevant code block and then jump
// to the first point after the end of the if else code block.
if (score >= 90)
{
Console.WriteLine("You got an A.");
}
else if (score >= 80)
{
Console.WriteLine("You got a B.");
}
else if (score >= 70)
{
Console.WriteLine("You got a C.");
}
else if (score >= 60)
{
Console.WriteLine("You got a D.");
}
else
{
Console.WriteLine("You got an F.");
}

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

Using bool in Decision-making

Remember a while ago I mentioned one type called the bool and how useful it would be. That’s what we’re going to discuss here. We can use the bool type with variables in the decision-making process and this is, in fact, very common. Look at the next code block where we determine if a player has sufficient points to pass the game level:

```
int score = 45; // At some point, this can change as the player goes through the game.

int pointsNeededToPass = 100;


bool levelComplete;


if(score >= pointsNeededToPass)

{

levelComplete = true;

}

else

{

levelComplete = false;

}


if(levelComplete)

{

// Later on, as we learn more C#, we can do more here.

Console.WriteLine("You've beaten the level!");

}
```

It is worth noting, that the relation operators all return bool values, which means that relational operators can be used to assign values directly to bools:

```
int score = 45; // At some point, this can change as the player goes through the game.
int pointsNeededToPass = 100;

// You don't really need the set of parentheses but it does make the code more readable.
bool levelComplete = (score >= pointsNeededToPass);

if(levelComplete)
{
// Later on, as we learn more C#, we can do more here
Console.WriteLine(“You’ve beaten the level!”);
}
```

The ! Operator

The != operator is used to check that two values are NOT equal, but there is another way to use the ! sign and that is for negating conditions. We’ll take a look now but when we talk about conditional operators shortly, you will see just how powerful they are.

The ! operator simply takes the opposite value for what it gets used on, for example:

```
bool levelComplete = (score >= pointsNeededToPass);

if(!levelComplete)
{
Console.WriteLine("You haven't won yet.  Keep trying...");
}
```

This can also be combined with all the conditional operators although the ! operator is higher than the relational operators in precedence. This means it is done first, before any comparison. If you want the comparison done first then negated, you will need to use parentheses, like this:

```
if(!(score > oldHighScore))
{
}

// that's pretty much the same as:
if(score <= oldHighScore)
{
}
```

Over time, you will come to learn that there are quite a few ways of doing the same thing – you just need to make sure that you use the method easiest for you and easiest to understand.

Conditional Operators: && and ||

There are quite a few ways where your conditions could be significantly more complicated. For example, let’s say that you are creating a game in which the player is controlling a spacecraft. This spacecraft has armor and shields and the player loses a life when both are gone. In this instance, you would need to write a code that checks both of these and not just one of them.

This is where we can play with the conditional operators. In C#, we have the AND operator (&&) and the OR operator (||) and these are what we use to check several things at once:

```
int shields = 60;
int armor = 30;

if(shields <= 0 && armor <= 0)
{
```

```
Console.WriteLine("Life lost.");
}
```

Of course, this means if the shields are less than or equal to zero and the armor is less than or equal to zero. Both of these conditions must be true for the life to be lost.

The || operator works much the same with the exception that only one condition has to be true:

The || operator works in a similar way, except that *only one* of the two sides needs to be true:

```
int shields = 60;
int armor = 30;

if(shields > 0 || armor > 0)
{
Console.WriteLine("You're alive!  Keep going!");
}
```

When you use either operator, a computer will do what’s known as “lazy evaluation” and this means that, unless it has to, it will not check the second part. So, in the previous example, the computer always checks the shields to see if they are greater than 0 but the only time it will check armor greater than 0 is when shields is equal to 0.

You can also put several of these together and, using parentheses, come up with some unique conditions. Whatever you want to do is possible; you just have to keep an eye on readability and that leads us to an alternative way of writing code to make it more readable.

Nesting if Statements

The last thing I want to discuss is the nested if statement. What this means is that if statements and if-else statements can be placed inside other if statements in a process called nesting.

For example:

```
if(shields <= 0)
{
if(armor <= 0)
{
Console.WriteLine("Your shields and armor are both zero!  You lose a life!");
}
else
{
Console.WriteLine("Your shields are gone, but you have armor.  You are still alive – just!");
}
}
else
{
Console.WriteLine("You have shields left.  The world and you are safe.");
}
```

That’s not where it ends though. You can nest an if statement inside an if statement inside an if statement, and so on. However, just because you can do it, it doesn’t mean that you should. While intelligent nesting can make your code easier to read, go too far and you won’t be able to read it at all. I guess what I’m trying to say is, use it when you need to but don’t overdo it.

Quick Primer

- The if statement in C# works the same as it does in Java and C++:

```
if(condition)
{
statement;
statement;
statement;
}
else if(another condition)
{
statement;
statement;
}
else
{
statement;
statement;
}
```

- The relational operators, ==, !=, <, >, <= and >=, also work much the same as in other programming languages – they are used to check if values are equal or not equal, if they are less than or greater than, less than or equal to, or greater than or equal to respectively.
- Boolean types are negated by the ! operator.
- The AND (&&) operator and the OR (||) operator are used to compare several things at once and work exactly the same as they do in Java and C++.
- You can nest if statements – but don't overdo it.

What's Next?

The if statements are a vital part of programming because they are the core of decision-making. All computer programming languages have some kind of if statement in them and this is why, despite what some people may tell you, HTML is NOT a proper programming language.

Next, we are going to look at a switch statement, which is a construct used for doing much the same as an if statement but in a different way.

Switch Statements

In the last section, we went over decision-making and using if statements. What we are going to discuss now is a construct that is similar to these if statements; it’s another statement called a switch statement.

To be fair, whatever you use a switch statement for can just as easily be done with an if statement. There won’t be any other switch statements used throughout this guide but there is a good chance you will come across them on your programming journey so you do need to understand them.

Switch Statement Basics

Variables are a common part of C# programming and it is normal to want to do different things with these variables depending on what their value is. For example, let’s say that we have got a menu and it has five different options in it. The program user will input their choice and we put that value into a variable. Whichever option they choose on the menu will result in something different being done.

Using what you learned already, you could write a complex if else-if statement that would be something like if else-if, else-if, else-if else. It would look something like this:

```
int menuChoice = 3;


if (menuChoice == 1)

{

Console.WriteLine("You chose the first option.");

}

else if (menuChoice == 2)

{

Console.WriteLine("You chose the second option. That’s a good one!");
}
else if (menuChoice == 3)
{
Console.WriteLine("I can't believe you chose the third option.");
}
else if (menuChoice == 4)
{
Console.WriteLine("You can do better than choosing the fourth option....");
}
else if (menuChoice == 5)
{
Console.WriteLine("5?  You really want the fifth option?");
}
else
{
Console.WriteLine("Hey! That option doesn’t exist!");
}
```

That does the job, all right, but we could also use a switch statement to do the same job. This requires the use of the switch keyboard and, for each case or choice in the menu, a case keyword. Also included are two other keywords that we’ll discuss later – default and break.

As a switch statement, the above if statement would now look like this:

```
int menuChoice = 3;
```

```
switch (menuChoice)
{
case 1:
Console.WriteLine("You chose the first option");
break;
case 2:
Console.WriteLine("You chose the second option. That’s a good one!");
break;
case 3:
Console.WriteLine("I can’t believe you chose the third option.");
break;
case 4:
Console.WriteLine("You can do better than choosing the fourth option....");
break;
case 5:
Console.WriteLine("5?  5?  You really want the fifth option?");
break;
default:
Console.WriteLine("Hey! That option doesn’t exist!");
break;
}
```

As you see, we began with the switch statement and the enclosed the variable to be switched in the parentheses. Think of the switch as a railroad switch; when it’s used, it changes where the trains go.

Next, we have our case statements in a sequence. These are also called case labels and are used for indicating whether the variable matches the case statement value and that the block of code is where the execution flow will go.

One important thing to note is that the execution flow will go into one case label exactly so there will never be a situation where two or more case blocks are executed.

At the end of every case block, there must be the break keyword; this is what sends the flow outside the switch statement and on to the next code section. Notice that we also have a label of default. This is used to indicate that if no other case label works, the flow should be directed here. You don't have to have the default label last but it tends to be the place where it is used most commonly and it is the best practice. This is because the default block catches anything that isn't a specific situation for the case labels. In addition, this also takes the place of the last else block in the original long if statement.

Note that case conditions in switch statements do not take general logic. An if statement could be this:

if(x < 10)

However, you can't do this:

case x < 10:

and you can't do this either:

case < 10:

If you want complicated logic, go back to using an if statement instead.

Types Used with Switch Statements

In the example above, we used the int type. While many types can be used in switch statements, not all of them can. Those that can include:

- bool
- char

- string
- int
- byte
- short
- long
- ulong
- ushort
- ubyte
- uint

Enumerations can also be used in switch statements – we haven't got to these yet.

Implicit Fall-Through

If you have been using Java or C++ you might already know the little trick of when the break statement is left off, a case block will "fall through" to the next one. So, in those two languages, you could do this:

```
// This won't work in C#

switch (menuChoice)

{

case 1:

Console.WriteLine("You chose option 1.");

case 2:

Console.WriteLine("You chose option 2.  Or maybe you chose option 1.");

break;

}
```

Because we don't have the break with the first case, the code is executed and then carries on down to the second case, until a break statement is reached. In C#, this cannot happen because every case block has to have a break statement.

The reason this is required is that, in many cases, the break statement would be accidentally omitted and a bug would be the result. That bug is not easy to fix; to stop this happening, C# will not allow the break statement to be left off, eliminating the fall-through.

However, this can be done in C# if you have several case blocks that do not have any code in between:

```
// This will work in C#
switch (menuChoice)
{
case 1:
case 2:
Console.WriteLine("You chose option 1 or 2.");
break;
```

What this does is lets you do the same thing whether the value is 1 or 2. Although there is no break statement with case 1, it really doesn't need it; all we are saying is that both case 1 and 2 are the same.

You could do the exact same thing with an if statement.

Quick Primer

- Switch statements are an alternative way of writing multiple if else-if
  blocks. Rather than writing this:

```
int menuChoice = 2;
if(menuChoice == 1)
{
Console.WriteLine("You chose the first option");
}
else if(menuChoice == 2)
```

```
{
Console.WriteLine("You chose the second option. That’s a good
one!");
}
else
{
Console.WriteLine("I don't know what to do with that number.");
}
```

You would write this:

```
int menuChoice = 2;

switch (menuChoice)
{
case 1:
Console.WriteLine("You chose the first option");
break;
case 2:
Console.WriteLine("You chose the second option. That’s a good
one!");
break;
default:
Console.WriteLine("I don't know what to do with this number.");
break;
}
```

- You can use bool, char, string, int, uint, long, ulong, short, ushort, byte and ubyte types in switch statements.

- Enumerations can also be used – more about those later.
- There is no implicit fall-through in C# because the break statements
  cannot be left off, thus allowing code to fall from one block down to the
  next. The only exception is where several case labels are used together
  without any code in between them.

What's Next?

Things are looking good; you have the decision-making process in hand so we can move on to another important part of C# and that is looping.

Looping

After looking at the if statement, which is a very powerful part of programming, it seems suitable that we move onto another powerful construct – the loops. C# loops let you repeat one piece of code over and over again and we're going to be looking at three types. There is a fourth but that fits better with the next section, arrays, and it will make far more sense to you.

The While Loop

The first loop is the while loop. This is used to repeat a specified section of code repeatedly, for as long as a certain condition remains true. The while loop is constructed in a manner that makes it look very much like the if statement.

Here's an example:

```
while( condition )
{
// This code will be repeated until the condition evaluates false.
}
```

We'll begin with a simple example; all it does is counts to 10:

```
int x = 1;
while(x <= 10)
{
Console.WriteLine(x);
x++;
}
```

This begins with x = 1. After that, the program will check the while loop condition to see if it is true or not – and it is. The code in the while loop will now be executed. The value of x s written, which is 1, then x is incremented (1 is added). When it gets to the final curly brace that is the end of the loop where the code will then return to the beginning of the loop and repeat.

This time, x now equals 2 (remember, we incremented by 1 last time). However, the condition still evaluates true because x is still less than 10. The loop continues and 2 is printed, with x being incremented to 3. This happens 10 times in total; when x is incremented to 11 the program will check and the condition will finally evaluate to false. At this point, the flow jumps straight to the first code after the end of the loop.

Keep in mind that you can very easily wind up having a bug in your code and this can make it so that the loop condition is never met. Just imagine for a moment that we didn't add in the x++ line – in fact, take it out of the code now. All your program will do is repeat the loop endlessly; x would still equal 1 at the end of every loop and that is so common that it has a name – infinite loop. I guarantee that, by the time you have written a real program for yourself, you will have done this at least once. All you can do is kill the program, fix it, and then start again.

While I've got you focused on the infinite loop, it is worth mentioning that these are, on occasion, deliberately created. Sounds odd but it is incredibly easy to do:

```
while(true)

{

// Depending on what you put here, the loop will never end...

}
```

When it's done deliberately, it isn't called an infinite loop. Instead, it is known as a forever loop and, depending on what goes in the loop, a break statement for example (more on that later), you can get out of it.

This is a more complicated code that continues to repeat until a number between 0 and 10 is entered by the user:

```
int playersNumber = -1;
```

```
while( playersNumber < 0 || playersNumber > 10 )
{
// This code is continually repeated until the player enters a number
// between 0 and 10.

Console.Write("Enter a number between 0 and 10:  ");
string playerResponse = Console.ReadLine();
playersNumber = Convert.ToInt32(playerResponse);
}
```

With a while loop, you need to remember that the condition is checked before the loop is entered into. If the condition isn't met at the start, the loop is never entered. In the above example, we needed to initialize playersNumber as -1. If we had initialized it to 0, for example, the flow would have gone over the inside of the loop and the player would never have had the chance to input a number.

The Do-While Loop

The do-while loop is a variation on the while loop. Remember, with the while loop, the condition is checked before the loop is entered; if it isn't met, the loop could be skipped over. This isn't always a bad thing, it is just something you need to keep in mind.

The do-while loop will always be executed once and usually more than once. This is very useful if you need to set something up the first time the loop goes through and you know that it has to be executed a minimum of one time.

Let's reuse our last example because it was perfect for the do-while loop. We needed the player number to be initialized to -1 so it was forced to go through at least one loop. If you use a do-while loop instead, you don't need to do that:

```
int playersNumber;
```

```
do
{
Console.Write("Enter a number between 0 and 10:  ");
string playerResponse = Console.ReadLine();
playersNumber = Convert.ToInt32(playerResponse);
}
while (playersNumber < 0 || playersNumber > 10);
```

A do-while loop is formed by adding the do keyword at the beginning of the loop with the while keyword at the end. Note that a semicolon is required at the end of the line with the while keyword. Everything else remains exactly the same but the playerNumbers variable does not need to be initialized.

The For Loop

The for loop is a little bit different and very common in computer programming because it is an easy way of adding a counting loop type; we will see more of them in the next section on arrays but we'll have a basic look now.

They are a bit more complex to set up because three components are needed in the parentheses; the previous loops only needed one. A for loop looks like this:

```
for(initial condition; condition to check; action at end of loop)
{
//...
}
```

The three parts to the loop statement are separated by semicolons. The first bit sets the initial state while the second bit is the actual condition – this is what was in the other two loops. The third is the action that happens when we get to the end of the loop. Here's an example using a for loop and we'll do our count to 10 again:

```
for(int x = 1; x <= 10; x++)

{

Console.WriteLine(x);

}
```

This time, the variable was both declared and initialized in the for loop, as with int x = 1.

One of the reasons this is a nice kind of loop to work with is because the loop logic is separated from what you want to do with your number. Instead of the x++ command being inside the loop and having the variable declared before the loop, we just put it all in one place so the loop body is that much easier to read.

There is something else cool to think about. Whatever you can do with one of the loop types, you can easily do it with the other two. Sometimes, one of the loops will suit your purpose better; it will look cleaner, it may be easier to understand or just make better sense. At the end of the day, each of them is a loop and you can do the same thing with any one of them.

Breaking Out of Loops

Something else you can do with a loop is break out of it. You might get to a point that you come to the conclusion there really isn't any point in carrying on with the loop – any of the loops. With the break keyword, you can get out of a loop whenever you want:

```
int numberThatCausesAProblem = 47;


for(int x = 1; x <= 100; x++)

{

Console.WriteLine(x);
if(x == numberThatCausesAProblem)
{
break;
}
}
```

This code will continue until it gets to 47; the if statement will catch it and push it out of the loop. This is a basic example but it shows you when you could use the break keyword. When the loop starts, we expect to go all the way to 100 but, at some point, we determine that there is a problem and we need to get out of it.

Going back to the forever loop we mentioned earlier, this is an example of what would make one:

```
while(true)
{
Console.Write("What do you want to do?  ");
string input = Console.ReadLine();

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

The Next Loop Iteration

In much the same way as we used the break keyword, we can also use another keyword that will take you back to the beginning of the loop and start over, rather than breaking out of it. In other words, it goes to the next loop iteration before it finishes the one it is on. For this, we use the continue keyword:

```
for(int x = 1; x <= 10; x++)
{
if(x == 5)
{
continue;
}

Console.WriteLine(x);
}
```

In this example, every number will be printed with just one exception – 5. This is missed because we added a continue statement. When it gets to that part, it will go straight back to the x++ part of the loop and check the condition, x < 10, once more. Then it will just carry on with the next loop cycle.

Nesting Loops and Some Practice

As with the if statements, loops can also be nested. And you can put if statements in loops and loops in if statements! You can almost do whatever you want.

Let's see a couple of examples first, and then do some practical work.

This is a dead simple example, one that every new C# learner will do. We are going to write a loop that uses loops to print the following:

```
**********

**********

**********

**********

**********
```

So, the code we need is this:

```
for(int row = 0; row < 5; row++)

{

for(int column = 0; column < 10; column++)

{

Console.Write("*");

}


Console.WriteLine(); // This will make it wrap back around to the start
again.

}
```

This example is a little harder:

We want this:

```
*

**

***

****

*****

******

*******

********

*********

**********
```

So our code is:

```
for(int row = 0; row < 10; row++)

{
```

```
for(int column = 0; column < row + 1; column++)
{
Console.Write("*");
}

Console.WriteLine();
}
```

Did you spot that we got a little tricky there? We used the row variable inside the for loop condition with the column variable. I should also point out that most programmers just love 0-based indexing. This means indexes start at 0 and not 1. For example, if you have 5 rows, they would be numbered as 0, 1, 2, 3, 4 (not 1, 2, 3, 4, 5). You can probably see that I've done something here that I do all the time – row and column begin at 0 and go the number we want, 10 in our case, but not including that last number. That will do it 10 times.

Now it's your turn.

Have a go at writing the code that will display this:

```
*
***
*****
*******
*********
***********
```

It isn't going to be easy but you should have a go. Why? This is because now you are at the stage where you are ready to start writing your own code. Sure you can learn a certain amount just be reading this guide but you will learn a whole lot more by actually doing it.

Need a bit of a hint?

You'll want three loops. Each line has one big loop which will have two more loops inside – one printing "" and the other printing "*".

I will show you the solution at the end of this section.

Remember that I said there were four loops? We'll be discussing the for-each loop in the next section.

Quick Primer

- C# has a while loop, a for loop and a do-while loops and they work the
  same way as they do in Java and C++:

while(condition) { /* ... */ }

do { /*... */ } while(condition);

for(initialization; condition; update) { /* ... */ }

- The break keyword is used to get out of a loop at any time, moving you to
  the loop iteration using the continue keyword.

What's Next?

We've looked at several ways that we can create a loop. Loops are very powerful and, as you learn more you will come to understand them better. They do take some getting used to but you will get the hang of them.

Next, we look at arrays, a great way of storing things together.

Before I forget, here's your solution:

for (int row = 0; row < 6; row++)

{

```
    // Counting backwards!
    for (int spaces = 5 - row; spaces > 0; spaces--)
    {
        Console.Write(" ");
    }

    for (int column = 0; column < (2 * row + 1); column++)
    {
        Console.Write("*");
    }

    Console.WriteLine();
}
```

Arrays

Arrays are an incredibly powerful feature in C# because they let us store groups or collections of objects that are related. We'll be talking about how arrays are created, and how we use them. Then we'll look at multi-dimensional arrays, otherwise known in math as a matrix, and then finish by looking at the for-each loop.

What is an Array?

Arrays are a neat way of keeping track of collections or lists of multiple objects that are related. For example, let's say that you have a game with a high scoreboard and it holds up to 20 different scores. Using what we learned, we could create a variable for each score that we want to track. It would look like this:

int score1 = 98;

int score2 = 92;

int score3 = 85;

// ...

That's not a bad way to do it if you only have a few scores. What if you want to track thousands of scores? Suddenly, creating one variable for each score becomes just too much.

That's where arrays come in. These are ideal for tracking things like this because you can use them to store 20 scores or 20,000 scores; they are easy to create and, once you have them, incredibly easy to work with.

Working with Arrays

When you create an array, it is much like creating a variable. You provide it with a type, you give it a name and you initialize it; you can do all of that in one line if you want. Arrays are created using square brackets – […]:

int[] scores;

In that one line, you create an array of ints and you called it scores. If you want to create another array and assign that to a variable, the new keyword will be needed and you will have to specify how many elements will be stored in the array:

int[] scores = new int[20];

In the first example, we didn't give our array a value. We have this time; it has 20 items inside it.

Now we can assign a value to a specified point inside the array. When we reference a specific point, we use the square brackets and an index (sometimes called a subscript). The index is a number that lets the computer know which element to access in the array. If you wanted the first item in the array, this is the code to use:

int firstScore = scores[0];

It is vital to remember one thing – array indexes are 0-based. The first array item will always be numbered 0 – we will be talking about this later because there is an excellent reason for it and most computer languages do it. If you get this wrong, you will get an "off-by-one" error – this is because you are not on the correct number.

Use one technique to access any array value:

int thirdScore = scores[2];

int eighteenthScore = scores[19];

If you attempted to access a value that is outside the scope of the array, for example, scores[29] in an array with 20 items, the program just crashes and it tells you that the index was out of the array bounds.

Array values are also assigned with one simple technique:

int[] scores = new int[20];

scores[0] = 98;

scores[1] = 92;

```
scores[2] = 85;

// ...
```

There are other ways to initialize an array. The first is to create the array by providing specific values right at the start. You do this by enclosing the values you want to be assigned inside a set of curly braces, each separated by a comma:

```
int[] scores = new int[20] { 100, 92, 90, 87, 84, 80, 79, 75, 71, 62, 58, 56, 48, 40, 35, 29, 21, 19, 12, 10 };
```

To be honest, when you use this method to create your array, you can omit the number between the square brackets – the number of items in the array has already been stated:

```
int[] scores = new int[] { 100, 92, 90, 87, 84, 80, 79, 75, 71, 62, 58, 56, 48, 40, 35, 29, 21, 19, 12, 10 };
```

Another way is to use the length property. Yes, I know we haven't gotten to properties yet; we will do soon but, for now, just know that it really is quite easy to do – here's how:

```
int totalItemsInArray = scores.Length; // See, I said it was easy...

Console.WriteLine("There are " + totalItemsInArray + " items in the array.");
```

Let's have a look at a few examples, now that you know the basics:

Minimum Value in an Array

For the first one, we will use a for loop to work out what the minimum value is in a specified array.

The process will involve looking at each array item in turn. A variable is created to store the minimum value that we found so far and, as we go through the list of values in the array, the value of the variable will change every time we find a lower value than is already stored:

```
int[] array = new int[] { 4, 51, -15, 33, -87, 85, -9, 25, 92 };
```

```
int currentMinimum = Int32.MaxValue; // We start with the highest number so that every other number in our array will be the lowest.

for(int index = 0; index < array.Length; index++)
{
if( array[index] < currentMinimum )
{
currentMinimum = array[index];
}
}

// By now, the currentMinimum variable has the smallest or minimum value from the array.
```

Average Value in an Array

This is similar but different – we are going to find the average value from an array. The same pattern from the previous example is used but, this time, we will add all the array numbers. When that's done, we divide the result by the number of items in the array. As you would have learned in math, the average is the sum of all numbers divided by how many numbers there are:

```
int[] array = new int[] { 4, 51, -15, 33, -87, 85, -9, 25, 92 };

int total = 0;

for (int index = 0; index < array.Length; index++)
{
total += array[index];
}
```

float average = (float)total / array.Length;

Arrays of Arrays and Multi-Dimensional Arrays

Arrays can be of any type – floats, ints, bools, and so on. You could, if you wanted, have an array of arrays, or an array of arrays of arrays, or however far you want to go. This is one of the ways that a matrix is created – a grid that has a specified number of columns and rows (think of an Excel spreadsheet).

Creating an array of arrays is done like this:

int[][] matrix = new int[4][];

matrix[0] = new int[4];

matrix[1] = new int[5];

matrix[2] = new int[2];

matrix[3] = new int[6];

matrix[2][1] = 7;

Each array inside the main array has a different length. You can make them all identical if you want (we'll look at a better way of doing that shortly). When each individual array inside one main array has different length properties, it is known as a jagged array. If all the properties are the same, it is called a square or rectangular array. You can put any number of these together, as many as you want.

Assuming that we want a rectangular array, there is another way to work with them – the multi-dimensional array. To achieve this, several indices (the plural version of index) are placed inside a set of square brackets:

int[,] matrix = new int[4, 4];

matrix[0, 0] = 1;

matrix[0, 1] = 0;

matrix[3, 3] = 1;

C++ and Java do not have support for the multi-dimensional array although they do support jagged arrays.

The Foreach Loop

To finish arrays, at long last we come to the fourth loop – the foreach loop. This works very well with arrays and what it means is that the task you want to be done is done on every array element.

To use one, the foreach keyword is used with the array and the name of the variable that is to be used in the loop is specified:

```
int[] scores = new int[20];

// Populate the data and keep it maintained while your program is running – this is just an example:

scores[0] = 42;

scores[5] = -1;


foreach (int score in scores)

{

Console.WriteLine("Someone has this score:  " + score);

}
```

Inside the loop, the variable called score can be used. There is one important thing of note – inside the loop, there is no way to know what specific index you are on at any given moment. Most of the time that will not be a problem; the index number isn't an issue because you want something done with every element anyway.

If, however, you needed to know the current index, you would need to use the for loop:

```
int[] scores = new int[20];
```

```
// Populate the data and keep it maintained while your program is
running – this is just an example.
scores[0] = 42;
scores[5] = -1;

for(int index = 0; index < scores.Length; index++)
{
int score = scores[index];
Console.WriteLine("Score #" + index + ":  " + score);
}
```

Quick Primer

- Arrays are used for storing collections of objects of the same type and that are related.
- Creating an array requires you to use a set of square brackets along with the new keyword, i.e. int[] scores = new int[20];
- Arrays can also be created by supplying the values at the start, i.e. int[] scores = new int[]{1, 2, 3, 4, 5};
- Array values can be accessed and modified using the square brackets too, i.e. int firstScore = scores[0]; *and* scores[0] = 44;
- Array indexing is 0-based so the first element is index 0.
- Arrays of arrays can be created.
- Multi-dimensional arrays can also be created.
- The foreach loop can be used for looping through arrays and for doing something with each individual element.

What's Next?

That completes our look at arrays; these are useful in many different situations and every good programmer will make use of them. For our final subject, we look at enumerations. These are cool constructs that let you make your own kind of variables – sort of.

Enumerations

Enumerations are refreshingly easy to grasp. Enumerations are often called enums for short and they are a great way of defining a variable of your own – the first of several ways that you will learn.

Enumeration is derived from enumerate and that means “to count off, one after another” and that is pretty much what we will be doing. We’ll start by looking at how to use enumerations, then, we’ll discuss why they are valuable. It makes sense to look at what they are before learning how helpful they are – you will see what I mean. Lastly, we’ll recap how to use them.

The Basics of Enumerations

Knowing when to use an enumeration could be helpful.

Let’s assume that you want to track something that has a known set of values, albeit a small and specific set, that we can use. The most obvious one would be the days of the week and you could do this by defining Sunday as 1, Monday as 2, and so on.

Your code could look like this:

```
int dayOfWeek = 3;

if(dayOfWeek == 3)
{
// Do something because this is Tuesday.
}
```

What would be better to make things easier to read, you could create a variable for each day:

```
int sunday = 1;
int monday = 2;
int tuesday = 3;
int wednesday = 4;
int thursday = 5;
int friday = 6;
int saturday = 7;

int dayOfWeek = tuesday;

if(dayOfWeek == tuesday)
{
// Do something because this is Tuesday.
}
```

This is exactly what the enumeration is for – creating an enumeration and then listing every value that it could possibly have.

Creating an enumeration must be done outside of any classes that you are using (we'll talk about classes in another part of the guide). For now, we have just one class in our file, the Program class. The enum keyword is used, you name your enumeration, and then you have all the values it can have in between a set of curly braces:

```
public enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

Remember, this must be placed outside of the other classes, so your program is going to look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;
using System.Net;

namespace EnumerationTutorial
{
// enumerations are defined outside of other classes.
public enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday };

public class Program
{
static void Main(string[] args)
{

}
}
}
```

Inside the Main method, variables can be created with the new type, DaysOfWeek, rather than using an int or any other type. Using your own type to create a variable works exactly the same as all variables:

```
DaysOfWeek today;
```

Assigning a value to it requires the use of the dot operator (.) which we will talk about later on. For now, the dot operator is used for member access, which means that it is used when you want to use something that is already a part of something else. The values that we have in the enumeration are already a part of the enumeration so we can use the dot operator:

```
today = DaysOfWeek.Tuesday;
```

Again, this works the same as any other variable, even if you want to do something like a comparison in an if statement:

```
DaysOfWeek today = DaysOfWeek.Sunday;


if(today == DaysOfWeek.Sunday)
{
// ...
}
```

Why Enumerations are Useful

There is an excellent reason why you should use enumerations. At the beginning, we used the int type with the days of the week. We defined dayOfWeek = 3 and that was Tuesday. What if we had put dayOfWeek = 17. As far as the computer is concerned, this is perfectly valid because dayOfWeek has been defined as a variable of int type. Why shouldn’t it be 17? However, that doesn’t really make any sense because we only want 1 through 7.

With an enumeration, you can force both programmers and computer to use specified values only, values that you have defined. Using them prevents so many errors and it makes things so much easier to read. For example, if you saw dayOfWeek = DaysOfWeek.Sunday, you would understand it better than dayOfWeek = 1.

More Details

Before we move on, there is one more detail we need to look at. All C# is doing is wrapping the enumeration around the ints. So, essentially they are nothing but numbers. When an enumeration is created, it starts by providing them all with values, beginning at 0, and increasing by 1 for each one after. With our enumeration, Sunday is valued at 0, Monday at 1, Tuesday at 2, and so on.

This means that we can cast to an int or another number, and cast from it, like this:

int dayAsInt = (int)DaysOfWeek.Sunday;

DaysOfWeek today = (DaysOfWeek)dayAsInt;

DaysOfWeek tomorrow = (DaysOfWeek)(dayAsInt + 1);

Note that you need implicit casts for either direction – from enum to int and int to enum.

When an enumeration is created, you can assign your own values (numeric) aside from the default, if you need to:

public enum DaysOfWeek { Sunday = 5, Monday = 6, Tuesday = 7, Wednesday = 8, Thursday = 9, Friday = 10, Saturday = 11 };

Quick Primer

- Enumerations are a quick way of defining your own variable type with specific values.
- You define an enumeration like this: public enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
- An enumeration must be defined outside of any class that you already have in your program.
- Because enumerations can take on only the specific values that you choose means that there is no way of ending up with meaningless or bad data.
- It is possible to assign numeric values of your own to enumeration items: public enum DaysOfWeek { Sunday = 5, Monday = 6, Tuesday = 7, Wednesday = 8, Thursday = 9, Friday = 10, Saturday = 11 };

What's Next?

Enumerations are just one of several ways to create your own variables. You can use them in many different ways as you will find out as you gain experience.

We are now finished with the beginner's guide and it’s time to move on to the more intermediate topics in C#. We’ll start off with the methods.

Part 2: Intermediate Guide

Methods

Just think for a minute about programs like Microsoft Word or games like Gears of War. How much code do you think is involved in creating programs of that size? Let’s face it; they are massive programs, worlds away from the tiny ones we’ve been looking at.

How would you manage and keep track of that code? How would you know where to look if a bug arose?

As a programmer, you will learn that, when you have large programs like this, the easiest way to deal with big programs is to break them down. Like you would any large task, taking it down into smaller chunks makes it easier to manage. What’s even better, is you get the chance to reuse each chunk too. This is an approach you will see very often and it’s known as “divide and conquer”.

C# has a neat feature that lets us break our programs down and it’s called a method. You might know these as functions or procedures from other languages and the way they work is really quite simple. You can place code inside a method and then you can access it whenever and wherever you want. A method can be reused as many times as you want, which means you don’t need to keep on writing the same piece of code over and over again. Think of the time this can save you!

We’re going to look at how methods are created, and how they are used – a process known as calling a method. Then we’ll look at how to retrieve data from a method and how to send to it as well. That will be followed by a discussion on a process known as overloading – creating many methods with the same name – followed by yet another look at our Math class and how it works with methods. Lastly, we’ll look at a very powerful trick that can be used with a method, called recursion.

Creating a Method

We'll start by creating a method. When you open a new project you will see code that looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;
using System.Net;

namespace MethodsLesson
{
    public class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Remember our very first code – the Hello World! program. We used the template code for that and it already has a method defined in it – the Main method. Looking at the code you can see that this method is in a class named Program – we'll start discussing classes in the next section. As you can see, you've already had a little practice using methods without realizing it.

In a nutshell, every method belongs to a class. When you create a method you will add it to a class – we're going to add to the Program class, the same place as the Main method is found.

Add the following code to the template:

```
static void CountToTen()
{
    for(int index = 1; index <= 10; index++)
    {
        Console.WriteLine(index);
    }
}
```

Your code should now look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;
using System.Net;

namespace MethodsLesson
{
    public class Program
    {
        static void Main(string[] args)
        {
        }

        static void CountToTen()
        {
            for (int index = 1; index <= 10; index++)
            {
                Console.WriteLine(index);
            }
        }
    }
}
```

At the moment, this code isn't going to do anything. The method still needs to be called and we'll do that shortly. First, a quick look at what we added.

The code is quite simple. First, we added the static keyword. We will be discussing this in the next section on classes but, for now, just consider it as a bit of code that always has to be placed there. As it happens, most methods don't really need it but you still need to understand it and what you are going to do with this piece of code requires it.

Next, we added the void keyword. We'll come to this shortly when we get on to talking about returning data from a method. In short, this keyword is used to tell us that the method will not return anything; it will only do what it has to do and then finish.

Next, is to name of the method; in our case that is CountToTen. As with the variables, methods can be named anything but you really want to give it a name that actually means something and is related to what the method does. You should also get into the habit of adding comments above the method to describe what it is doing. The difference between the variable names and the method names is that method names start with a capital letter – the first word of a variable name is in lowercase.

Next, a set of parentheses is needed. We'll discuss this when we talk about handing something to a method. What is inside the parentheses are the arguments to the method – we’re not using any this time so there is nothing there.

This is followed by a set of curly braces and these contain the method body – this is the code that dictates what the method does. Last, we have a for loop. It is worth mentioning that all the code we have written to this point can go inside a for loop.

It does not matter what order you add the methods to a class. In our example, CountToTen could easily have been placed before Main and it would have made no difference whatsoever.

Calling Methods

We’ve created the method so now it would be good if we could use it. This is simple – add the following code to the Main method:

```
CountToTen();
```

So, now your code looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;
using System.Net;

namespace MethodsLesson
{
   public class Program
   {
      static void Main(string[] args)
      {
            CountToTen();
        }

        static void CountToTen()
        {
            for (int index = 1; index <= 10; index++)
            {
                Console.WriteLine(index);
            }
        }
    }
}
```

You can run the program now and it will start from the Main method automatically. Watch it jump over the method called CountToTen and do everything that the method states. When it gets to the end, it will return to where it started. Look at this code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.IO;
using System.Net;

namespace MethodsLesson
{
    public class Program
```

```
  {
     static void Main(string[] args)
     {
        Console.WriteLine("I'm going to go into a method.");

        DoSomethingGreat();

        Console.WriteLine("I’ve returned from the method.");
     }

     static void DoSomethingGreat()
     {
        Console.WriteLine("I’m in the method, doing something
great!");
     }
  }
}
```

The output from this is:

```
I'm going to go into a method.
I’m in the method, doing something great!
I’ve returned from the method.
```

Methods can be called from inside other methods so you can knock yourself our creating methods and calling them all over the place!

Returning Something From a Method

A method is always created for the purpose of doing something. Sometimes, though, we want the method to do something and then give us the results. This is called returning. So far, we haven’t used a method that returns anything. Look back over your code and you'll see we used the void keyword in front of our method names. As mentioned earlier, the void keyword tells us that the method won't return anything. However, we can change that; instead of using the void keyword, we use a variable type instead (bool, int, float, string, etc.). The method will then return a result of the specified type.

To return a value, the return keyword is used in the method, then the value that is to be returned and finished with a semicolon. One simple example would be:

```
static float GetRandomNumber()

{

   return 4.385f;

}
```

We began by specifying the return type – we used float which means that the result is of the float type. Inside the method, we added the return keyword together with the value we wanted to be returned.

Let's see a more complex example. In the previous code, we have asked a user to input a number into the program. With this example, we will make a method that requests a user to input a number between 1 and 10; it will continue to make the request until the user gives the number that works. Then, we return that number:

```
static int GetNumberFromUser()

{

   int usersNumber = 0;


   while(usersNumber < 1 || usersNumber > 10)

   {

      Console.Write("Enter a number between 1 and 10:  ");

      string usersResponse = Console.ReadLine();
```

```
        usersNumber = Convert.ToInt32(usersResponse);
    }

    return usersNumber;
}
```

If the method doesn't return anything (the void keyword was used) then a return statement is not required. If a return is expected, we must have that statement.

When something is returned from a method, we can use a variable with the same type to catch it. Alternatively, typecasting could be used, implicit or explicit, to convert it. As an aside, the word 'catch' is used lightly here. In a future part of the guide, we will look at the catch keyword and how it is used with exceptions - it does have a specific meaning.

Back to this, to make use of the result that the method returns, you can do this:

```
int usersNumber = GetNumberFromUser();
```

Do you recognize it? If you've been paying attention, you should because we've been doing things like this all through this guide. We used lines like string usersResponse = Console.ReadLine() – this calls a method called ReadLine from the class called Console and has the value that the method returned.

Another note: while you will almost always see the return statement as the final line in a method, it doesn't necessarily have to be. You can have an if statement in the method which will return a value only when the method meets a specific condition, for example:

```
static int CalculatePlayerScore()
{
    int livesLeft = 5;
    int underlingsDestroyed = 19;
    int minionsDestroyed = 8;
    int bossesDestroyed = 2;

    // If the player loses all their lives, they also lose all their points.
    if(livesLeft == 0)
    {
        return 0;
    }

    // Otherwise, 10 points are awarded for every underling the player
destroys,
    // 150 points are awarded for each minion destroyed, and 1500
points for every boss
    // destroyed.
    return underlingsDestroyed * 10 +
            minionsDestroyed * 150 +
            bossesDestroyed * 1500;
}
```

If the method does return void, although the return keyword isn't required, you can add it into your code on its own:

```
static void DoSomething()
{
    int aNumber = 1;

    if(aNumber == 2)
    {
        return;
```

```
   }

   Console.WriteLine("This will only be printed if the 'return'
statement isn’t executed.");
}
```

Note that whenever we hit a return statement, the flow goes straight back to the point the method was called and nothing more will be executed from within the method.

Handing Something Off to a Method

On occasion, we will want a method to do something with specific information. We can place what information we want the method to work with inside parentheses in a process called handing off.

```
static void Count(int numberToCountTo)
{
   for(int current = 1; current <= numberToCountTo; current++)
   {
      Console.WriteLine(current);
   }
}
```

Where the method is defined, you add the variable type and provide it with a name that it will use in the method. In our example, numberToCountTo is a parameter. When a method is called and it has a parameter in it, a value can be passed to the method or by placing the value inside parentheses, you can hand off the value so the method can work with it:

```
Count(5);
Count(15);
```

In conjunction with the code before, these two lines will print the numbers 1, 2, 3, 4, 5 and then it will print 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.

Passing in Multiple Parameters

It is also possible to create a method that can pass in multiple parameters. The example I am going to show you is somewhat overkill but it does show you how to pass in two numbers to a method called Multiply. This method multiplies the numbers and returns the result:

```
static int Multiply(int a, int b)

{

   return a * b;

}
```

You can also see from this that parameters can be passed in and a result returned all in one method something that is common.

Method Overloading

There is a cool feature with methods that can also be quite confusing. It’s called method overloading. This is when you create several methods that all have the exact same name.

The key to this is, while methods can share the same name, they cannot share a signature. The signature to a method is the combination of the name and the types of parameter and order – parameter names are not taken into account, only the type. The reason for this is that when a method is called, the compiler works out the overload to be used based on the name of the method AND what data types are passed in. Variable names don’t really matter either when methods are called so they too are ignored.

Below is an example of a method signature:

```
static int Multiply(int a, int b)

{

   return a * b;

}
```

The signature here is static int Multiply(int, int).

Methods can only be overloaded if a different signature is used so the following piece of code would work:

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


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

This will work because of the methods called Multiply have differing numbers of parameters and that means they each have different signatures. It would also be possible to define another method called Multiply that didn't have any parameters at all (int Multiply()) or that just had a single parameter (int Multiply(int)) although it's hard to think how methods like that could possibly do any multiplication with zero or with one number – it is only an example. In the same way, a Multiply method could be defined with 12 or 13 parameters but then people would think you were just a little on the crazy side. That said, there are rare cases where there could be a need for that many parameters.

You can also have the exact same number of parameters but with a different type:

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

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

Again, this will work because both of the Multiply methods have different signatures – Multiply(int, int) and //Multiply(double, double).

This next one will NOT work:

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


static int Multiply(int c, int d) // This will not work because it has an identical signature.
{
   return c * d;
}
```

The real magic in overloading is that you can create multiple methods to do similar things in different data types without needing to come up with different names for them all. It makes it easier for you and anyone else who calls the method. Because the signatures are different, the compiler can also easily work out which of the overloaded methods should be used.

Convert and Console Classes

Now that you know roughly how a method works, we should go back and look at two classes that we already used.

In the Console class, for example, we have done things like:

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

Now that we can apply our methods knowledge, we can see in the class called Console, a method called WriteLine has been defined and this is a method we can call. It has just one parameter, a string. That string, ‘Hello, World!’ is passed in and the method does what it is meant to do and then it goes.

This is actually a great example of why methods are used. Because the WriteLine method has already been created, there is no need to worry about how it does what it’s meant to do; all we need to be concerned with is that it does it and that it's easy.

The Convert class, which we also used a number of times, has something similar:

```
int number = Convert.ToInt32("42");
```

This class contains lots of methods, all of which you can use for converting types to other types. In the above example, the method, ToInt32, will take a parameter of a string. The class uses method overloading an a lot, not just because it has the ToInt32 method taking a string but because it has a method that takes a double, one that takes a short and so on.

A Brief Look at Recursion

One last trick used with methods is somewhat complicated and it is often used incorrectly. It’s called recursion and, although you don’t need to know too much about it now, you can start giving some thought to how you can use it.

Recursion is where a method is called from within itself. Here’s an example and, when you run it, it will break:

```
static void MethodThatUsesRecursion()
{
   // This makes use of recursion; the method is calling itself.
   // But although this is a simple example, it will cause problems.
```

```
    // All it will do is continue to go deeper, further on down the line.
    // A bit like the movie called Inception.
    // When it gets so deep that C# cannot handle it because it has far
too many layers
    // it will just give up and show you a "StackOverflowException".
    MethodThatUsesRecursion();
}
```

As I said, this will break because all it is doing is going into deeper methods. The problem here is that we don’t have a base case so it can never reach a point where it is finished and can return from the method call, going back to where it started.

A base case is nothing more than a situation whereby there isn’t any need to continue and the method just returns without any further recursion.

One classic example of when recursion could be used is the factorial function (mathematical). Do you remember your Math classes? Factorial is written with an exclamation mark next to a number. As an example, 7 *6 * 5 * 4* 3 * 2 * 1.72! would actually be 72 * 71 * 70 * 69 * … * 3 * 2 * 1. Factorial can very quickly lead to the overflow problems we talked of earlier. In this example, we already know that 1! is just 1. All the other numbers are numbers that will be multiplied by the factorial of the number that is one less than itself, for example, 7 * 6!.

And this creates the right opportunity for recursion:

```
static int Factorial(int number)
{
    // The base case is established here and, when we reach this
point, we are done. Both 0
    // and 1 are defined with a factorial of 1. There is no factorial
defined
```

```
    // for negative numbers, and there shouldn't be either, but we don't have the
    // right tools to account for that properly – more int eh Exceptions section.
    if(number == 0 || number == 1)
    {
        return 1;
    }

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

There is just one more thing to mention – every single step must take you closer to the base case otherwise you are never going to get to the end.

Quick Primer

- Methods are a good way of taking one piece of code that does something specific and put it together in ways that it can be reused.
- We can return a result from a method simply by specifying what the return type is in the method definition. We then add the return keyword inside the method where you want the value returned.
- You can pass a parameter into a method.
- You can create several methods with the same name so long as they all have different signatures.
- Recursion is the process of calling a method from inside itself. A base case is required otherwise the flow will just continue digging into more method calls and will kill itself in a StackOverflowException.

What's Next?

That takes care of a basic introduction to methods. They are powerful things and you will be using them a lot in this guide. You now know how to return a value from a method and how to pass in parameters which provide information.

We also looked at some more advanced topics, overloading, and recursion but don’t worry if you didn’t understand them – they are not really for this section. You just need to be aware that they do exist and that you will run into them from time to time.

Next, we are going to start on those all-important classes.

Classes Part 1

I’ve split this section into two because there is a lot to learn about classes – we’ll start by looking at how to use them.

C# is an object-oriented programming language, OOP for short. All this means is that any code you write is going to belong to an object. To kick off this section, we’ll look at objects, what they are, and how we define them. Then, we will create and use them in C#.

What is an Object?

People have been programming for years and, over time, it has become apparent that the best and the easiest way to structure a code is to emulate the real world. The real world has objects. Let’s take an example of a game player. The objects associated with this contain specific information – the player’s score, the number of lives they have left, and so on. In the real world, the objects can do some things by themselves or you can do some specific things with them, such as taking your turn in the game.

With OOP, we attempt to emulate this. We have objects which have certain data associated with them and we can do specific actions or operations with them. We covered variables earlier and that is how data is stored in C#. As we go through the process, you will see how variables can be used inside an object to store the data. We also covered methods and we know that they are a way of doing specified code and, again, you will see how methods can be used for representing object actions – either what the object can do or what can be done to the object.

Before we look at how all the pieces come together, we need to have a discussion about classes.

What is a Class?

We need a way of defining in C# what a type of object will have or the actions it can do. In other words, we want to know what the whole class of objects can store inside variables or what can they do.

We'll start by creating a class. This is a template, a blueprint if you like, for all objects of the specified type, and it will say what the object can do and what it can store. Rather than diving straight into this subject and creating classes and objects, we'll start by playing with some that are already created as part of the .NET framework.

Creating an Object

Let's begin by using the class called Random. This is a great class that lets you create some random numbers. This class can be used for all sorts of things – determining what a die roll will be, shuffling a card deck, even making some random things happen.

There is one thing I need to say about generating random numbers – in general, on a computer a random number isn't actually random; they are called "pseudo-random". These numbers are picked using an algorithm that makes them look random. The programmer needs to choose a starting number to get things started and this number is known as a 'seed'. Start with the same seed each time and you will get the same random number sequence each time.

To stop this, the random number generator may be seeded with the current time, right down to the millisecond – that way, the game will play out differently every time.

We're going to use the Random class and this will automatically seed the algorithm with the current time – although you do have the option of specifying the seed if you want.

The process of creating objects is much like that of creating a variable. We use the following:

Random random = new Random();

The first thing we do is specify what type the variable will be. However, unlike the variable where we used types such as int, char, string, etc., this time we use the Random type which is also the Random class.

Next, the variable is named – we've called it random but you can call it whatever you want. It is common to name variables after the type.

Then, we use the assignment operator (=) to assign the variable with a value, just like we did with any other variable. Now, you might have noticed two things about the statement, specifically the right-hand side. The first thing is we used the new keyword. This comes from Java and C++, both of which use keywords much the same. What this keyword indicates is that we are creating a new Random object and we use the Random class template for this. You might think that the Random() looks much like a method call and, in a way, it is. It is actually a special method type that we call a constructor, which we'll be discussing in a while.

Right now, what we have is a new Random object that we can now do something with.

Using an Object

Now we can do something with our working object. In the Random class, there are some methods that you can call – to start with, we have a method called Next. This is called by using the dot operator (.) with the method name:

Random random = new Random();

int nextRandomNumber = random.Next();

No doubt you spotted, with your newfound knowledge, that the Next operator has been overloaded. This means we have many versions of this method and that includes having one that allows us to pick a range of numbers for the random number to come from:

Random random = new Random();

int dieRoll = random.Next(6) + 1; // the Next method will return a number between 0 and 5, inclusive.

The class also has a method called NextDouble and this is used for returning numbers between 0 and 1 inclusive, so we can do random probabilities and other similar operations.

Using these classes and objects, we can do quite a few things. First, the object will let us track the data inside the object's variables. What this means is that nothing from outside this class can access or modify the internal data unless the class provides a method to do so. That includes being able to call methods that ensure any changes are allowable, for example.

The Random class tracks the seed value but there is no way of accessing it externally, which is as it should be. Responsibility for the seed value lies with the Random class, and only with that class. This is called encapsulation – one object is controlling its own data, and there can be no modification from external sources unless a specific method is provided by the object that sets out the rules by which it can be modified.

Both classes and object are nothing more than pieces of code that can be reused. The Random class, for example, has already been created for us and now we can reuse it when we want and where needed, without having to write the code ourselves. That is a cool feature and, as you will come to see, it will save you a ton of time.

Constructors

All classes are able to define several ways of creating an object instance. An instance is nothing more than another version of that object rather than referring to the entire class. In the examples above, we already created a Random class instance.

We call these methods of creating objects constructors. They work a little bit like a method with one difference – instead of returning something, they simply create a new version of an object – we will go over this more in the second part of our Classes section.

Like a method, the constructors can also have several parameters. In our last example, we had a constructor that had no parameters. So, let’s look at one that does have a parameter:

The Random class contains a constructor that must have a seed value. If you wanted to use this constructor instead, you would just use this code:

```
int seed = 5000;

Random random = new Random(seed);
```

This is very much like a standard method but we create new objects instead of returning values.

Before we discuss more about classes, there are a couple of other important things to discuss, starting with Stacks and Heaps.

Stack vs. Heap

Although we don’t really need to worry too much about actively managing memory and garbage collection with the .NET framework, we do need to keep both in mind to ensure that our applications are running at optimal performance. It doesn’t hurt to have, at the very least, a basic understanding of the way memory management works because it will help you understand how the variables we use in every program actually work. What I’ll be talking about here is Stack and Heap basics, variable types and why some variable behave the way they do.

In the .NET framework, there are two places where items are stored in the memory while your code is being executed. Those two places are the Stack and the Heap. Both are useful to help the code run and they both live in the operating memory on your computer containing the data needed to make everything happen as it should.

Stack vs. Heap: What's the difference?

The Stack is responsible for monitoring what gets executed or called in your code while the Heap is responsible for monitoring the objects, or most of the data.

You can consider the Stack to be nothing more than a pile of boxes, each stacked on top of one another neatly. Whenever a method is called, another box is stacked on top and we call that a Frame. At any given time, we can only make use of what is in the box at the top of the stack. When we are finished with it, i.e. the method has been executed, the top box is discarded, and we move to the next box.

The Heap isn’t much different but its purpose is to store data and not track the execution. This means that we can access anything in the Heap at any time regardless of where it is. Think of it as being a heap of laundry that needs to be put away but hasn’t been – when you want something, you grab it, no matter where in the heap it is.

The Stack will look after its own memory management; the top box is automatically discarded when no longer needed. By contrast, the

Heap has to consider Garbage Collection or GC; this is what deals with keeping the Heap clean and tidy.

What Goes On The Stack and Heap?

There are four types of things that go in the Stack and Heap as the code we have written executes:

- Value types
- Reference types
- Pointers
- Instructions

We'll look at each one in turn.

Value Types

In C#, anything that is declared with any of type declarations listed below is a Value Type because they come from System.ValueType.

bool

byte

char

decimal

double

enum

float

int

long

sbyte

short

struct

uint

ulong

ushort

Reference Types

Anything that is declared with the declaration types listed below are Reference Types. They will inherit from System.Object with the exception of the object itself – this System.Object.

class

interface

delegate

object

string

Pointers

The third type that goes into memory management is a pointer, which is a reference to a type. These are not used explicitly; the CLR (Common Language Runtime) manages them. Pointers differ from reference types in that anything that is a reference type is accessed via a pointer, which is a space in the memory that points to another space in the memory. Pointers take up space, the same as anything else that goes in the Stack and the Heap and it has a value of a memory address or of null.

Instructions

Lastly, instructions and we will see how they work later on in this section.

What Goes Where?

One last thing, there are two rules that determine what goes where:

- Reference types always go on the Heap.
- Pointers and Value types go to where they were declared. This is a bit more complicated and, to fully understand it, you need to fully understand how a Stack works so that you can work out where the value type or pointer was declared.

We know that the Stack has the responsibility for monitoring each thread while the code is executing. Think of it as being a thread state, with each thread having its own Stack. When a call is made by the code to a method, the thread will begin to execute the instructions that were JIT (Just In Time) compiled and that live in the method table. Plus, it will also place the parameters for the method onto the Stack for the relevant thread. Then, as the code runs through and we come up against variables in the method, they also go onto the Stack.

Here’s an example to make this clearer:

Take this method:

```
public int AddFive(int pValue)
{
    int result;
    result = pValue + 5;
    return result;
}
```

So, what goes on at the top of the Stack? Just bear in mind that what we are currently seeing is what goes to the top of the Stack, on top of all the other items already in there. Once the method begins executing, the parameters for that method go to the Stack (we’ll discuss passing parameters later). The method itself will not live on that Stack, only the parameters.

After that, control, which is the thread that is executing the current method, goes to the instructions which for our method are called AddFive() – this method is on the method table for the type. If this is the first hit on the method, a JIT compilation will be carried out.

As the method continues to execute, we are going to require memory for the variable called result and this will be allocated on the Stack. The method execution completes and the result is returned.

The memory that has been allocated on the Stack is cleaned – this is done by placing a pointer in the memory address where our AddFive() method began and we move to the previous method stored on the Stack.

In our example, the result variable goes on the Stack – it is worth noting that, whenever a Value Type gets declared inside a method body, it will automatically go to the Stack.

Occasionally a Value type will go on the Heap. Remember – value types will always go where declaration took place. If a Value Type is declared external to a method but internal to a Reference type, it goes inside the Reference Type onto the Heap.

Have a look at another example:

We have a class called MyInt – because it is a class, it is also a Reference Type:

```
public class MyInt

{

  public int MyValue;

}
```

This method is currently executing:

```
public MyInt AddFive(int pValue)

{

    MyInt result = new MyInt();

    result.MyValue = pValue + 5;

    return result;

}
```

As we saw earlier, the thread will begin executing the method and the method parameters go to the Stack for that thread.

Now things start to get a bit more interesting.

MyInt is a Reference Type and that means it goes on the Heap and the pointer on the Stack references it.

After the AddFive() method has stopped executing, cleanup takes place and we get left with an orphan MyInt on the Heap – because the Stack has been cleaned, there isn’t a reference to MyInt there anymore.

This is where a subject we’ll be discussing in more detail later comes in – Garbage Collection (GC). When the program gets to a specified threshold of memory and more Heap space is needed, GC starts, and GC stops any threads from running, known as a Full Stop, and then it looks in the Heap for all objects that the main program is not accessing; these are deleted. GC then reorganizes the remaining objects in the Heap so more space is available and adjusts pointers to the remaining objects in the Heap and in the Stack.

You can probably see that this is going to have an effect on performance so it is really important when you write code that is high-performance; you keep an eye on what is going in the Stack and the Heap.

Ok, so how does this affect you?

When you use Reference Types, you deal with pointers to the specific type, not to the object itself. When you use Value Types, you use the actual object.

Here’s an example.

The following method is executed:

```
public int ReturnValue()

{

    int x = new int();

    x = 3;

    int y = new int();

    y = x;
    y = 4;
    return x;
}
```

And the result is a value of 3. Got that? Simple, yes?

But, what if we were using our MyInt class, the one we used earlier:

```
public class MyInt
{
     public int MyValue;
}
```

And the method below is being executed:

```
public int ReturnValue2()
{
    MyInt x = new MyInt();
    x.MyValue = 3;
    MyInt y = new MyInt();
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```

The result is 4, not 3.

Why? How do we get 4?

Let's look a bit closer and see if things make more sense.

In our first example, everything went as it should:

```
public int ReturnValue()
{
    int x = 3;
    int y = x;
    y = 4;
    return x;
}
```

However, in the next example, both the x variable and the y variable are pointing to the exact same object on the Heap.

```
public int ReturnValue2()
{
    MyInt x;
    x.MyValue = 3;
    MyInt y;
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```

Hopefully, that has cleared things up and given you an understanding of the differences between C# Reference Type variables and Value Type Variables. Now, we'll look at method parameters.

Passing by Reference and by Value

Now that you know what the difference is between a Reference and a Value type, we can look at one very important difference in how both are used by methods.

Let's say that you are using a method like this:

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

When this is called from the Main method, or anywhere, using something similar to this:

```
int number1 = 3;
int number2 = 5;
int result = Multiply(number1, number2);
```

These numbers have been passed in by value and this means that all we send to the method are the variable values. If the value inside the method was modified, it would only have an effect on the method. Let's say that we have a statement of a *=2; inside the Multiply method – the value in the method is changed but the same value in the calling method remains unchanged. This is because all we did was send the value stored in it.

This will make a bit more sense if we look at it in contrast to passing references.

We have a method that needs an object as one of its parameters:

```
static int RollDie(Random random, int numberOfSides)
{
    int result = random.Next(numberOfSides) + 1;
    return result;
}
```

The object has been passed by reference and this means that the method refers to the original object called Random. If it is modified in the method, those modifications are reflected in the original calling place. This is because the object reference has been passed over and the method works with it on the Heap – both of these places refer to the same object on that Heap.

Therefore, with an object that is passed by reference, any modifications in the method will have an effect on the original object.

As far as value types are concerned, any changes made do NOT affect the original variable.

Garbage Collection

When objects are created in C#, CLR will allocate that object memory from the Heap. Every time a new object is created, that process is repeated. Everything has its limits, though and that includes memory. We don’t have an unlimited amount of memory and every now and again we need to have a cleanup so we have room for some more objects. That is where Garbage Collection enters the picture. GC or the Garbage Collector is responsible for managing the memory, both allocating it and reclaiming it. It does this by going to the Heap and picking up all the objects that are not in use by the program and removes them, freeing up memory.

Memory Facts

When a process is triggered, it is allocated a piece of virtual space. This comes from the physical memory, the memory that every part of your system uses. A system deals with physical memory, and programs deal with virtual space and GC also deals with that virtual space by allocating and deallocating it.

In basic terms, virtual memory has free-blocks which are also called holes. When a request for memory is made for an object, the allocation manager looks for these free-blocks and assigns them to the object.

There are three blocks in virtual memory:

- Free, which is empty space
- Reserved, which is already allocated
- Committed, which is kind of reserved for physical memory and cannot be allocated

How Does GC Work?

Garbage collection works on the managed Heap. This is nothing more than a memory block used for storing objects. When GC starts, it will check for any dead objects and those that are no longer being used; it removes them and then compacts the rest to free up more memory.

The Heap is actually managed by different generations, storing and handling objects, both long and short-term ones. The generations are:

- 0 Generation – Zero. Holds short-term objects, such as temporary objects that don’t last long. In this generation, GC is done on a very frequent basis.
- 1 Generation – One. The buffer between the short and long-term objects
- 2 Generation – Two. Holds the long-term objects, such as static variables and global variables that last for much longer. If an object is not collected on 0 Generation, it is moved to 1 Generation and becomes known as a survivor. In the same way, if an object is not collected in 1 Generation, it is moved to 2 Generation and will remain there.

How Does GC Determine Which Objects Are Live?

To determine if an object is live or not, GC will check the information below:

- All object handles are collected if they are not allocated by CLR or by user code
- Static objects are tracked because they are being referenced by another object
- Uses the Stack provided by JIT and Stack Walker

When Does GC Get Triggered?

That is difficult to answer because there is no set timing. When the following conditions are met, GC will be triggered:

- When virtual memory is running low on space
- When the allocated memory is suppressed to an acceptable threshold. If GC has found a high level of living objects it will increase the allocation.
- When the GC.Collect() method is explicitly called but, as GC is always running, it is rare that this would be needed.

What are the Managed/Unmanaged Objects or Resources?

Put simply:

- A managed object is one that is created, managed or otherwise within the CLR scope; is pure .NET code that is runtime-managed; is within the .NET scope; any .NET framework classes – string, bool, int variables (all referenced as managed code).
- An unmanaged object has been created externally to the .NET library control, is not CLR-managed and examples include COM objects, connection objects, file streams, Interop objects, pretty much any third-party library that is referenced within the .NET code.

Cleaning up Unmanaged Resources

When unmanaged objects are created, GC cannot clean them; it is up to the programmer to explicitly release the object when they are no longer needed. On the whole, unmanaged objects tend to be wrapped around or hidden around resources in the operating system, such as database connections, file streams, network-related instances, class handles, pointers, registries, etc. GC takes responsibility for tracking the lifecycle of every managed and unmanaged resource but it is still up to you to release the unmanaged objects. There are a few ways this can be done:

- Implementation of the Dispose method and the IDisposable interface
- Through a “using” code block

Implementation of the Dispose method is done in two ways:

- Use the SafeHandle class. This is an abstract class that is built-in and contains the IDisposable interface and CriticalFinalizerObject, implementing both.
- Override the Object.Finalize method, which will clean the unmanaged resources that a specific object uses before the object is destroyed.

Let’s have a look at some code which is used to dispose of the unmanaged resources:

Implement Dispose using the 'SafeHandle' Class:

```
class clsDispose_safe
  {
      // First check if the object has already been disposed of using a
flag.
      bool bDisposed = false;

      // Create an object of the SafeHandle class
      SafeHandle objSafeHandle = new SafeFileHandle(IntPtr.Zero,
true);

      // Dispose method (public)
      public void Dispose1()
      {
         Dispose(true);
         GC.SuppressFinalize(this);
      }

      // Dispose method (protected)
      protected virtual void Dispose(bool bDispose)
      {
         if (bDisposed)
            return;

         if (bDispose)
         {
            objSafeHandle.Dispose();
            // Free up any managed objects at this point.
          }

          // Free up any unmanaged objects at this point.
          //
          bDisposed = true;
        }
    }
```

Implement Dispose by overriding the 'Object.Finalize' method:

```
class clsDispose_Fin
    {
        // Flag: check whether Dispose has been called already
        bool disposed = false;

        // Public implementation of the Dispose pattern taht is
consumer-callable.
        public void Dispose()
        {
            Dispose1(true);
            GC.SuppressFinalize(this);
        }

        // Protected implementation of the Dispose pattern.
        protected virtual void Dispose1(bool disposing)
        {
            if (disposed)
            return;

          if (disposing)
          {
             // Free up any other managed objects at this point.
             //
          }

          // Free up any unmanaged objects at this point.
          //
         disposed = true;
       }

       ~clsDispose_Fin()
       {
          Dispose1(false);
       }
   }
```

The 'using' Statement

The ‘using’ statement is used to ensure that the object is disposed of, and is a better way of using IDisposable objects. When objects fall out of scope, the Dispose method is automatically called. It does pretty much the same thing as the Try…Finally, block and to show you how it works, I’ve created a class that has the IDisposable implementation; the ‘using’ statement will call Dispose, even if an exception is thrown:

```
class testClass : IDisposable
```

```
{
    public void Dispose()
    {
        // Dispose of the objects here
        // clean resources
        Console.WriteLine(0);
    }
}

//call class
class Program
{
    static void Main()
   {
        // Use the using statement with the class that implements
Dispose.
        using (testClass objClass = new testClass())
        {
                    Console.WriteLine(1);
         }
         Console.WriteLine(2);
    }
}

//output
1
```

```
0
2

//it is exactly the same as the TRY...Finally, code below
{
   clsDispose_Fin objClass = new clsDispose_Fin();
   try
   {
      //the code goes here
   }
   finally
   {
      if (objClass != null)
      ((IDisposable)objClass).Dispose();
   }
}
```

In this example, once 1 has been printed, the ‘using’ code block is finished and it will call the Dispose method, followed by the statement that comes after the ‘using’ block.

Quick Primer

- Object-oriented programming, or OOP, is where we create pieces of code that can be matched to real-world objects.
- C# classes are used to define what a whole class of objects does, the type of data it will store, what it can do, and what can be done to it.
- New objects can easily be created using something like Random random = new Random(); this will create a brand new

Random object which is then used for the generation of random numbers.

- Destructors are generally not needed in C# because, if an object is no longer needed, it is Garbage Collected.
- Stack and Heap are used for memory management.
- Garbage Collector will manage the allocation and de-allocation of the memory
- GC works on the managed Heap, which is a memory block used for storing objects.
- GC does not have any specific timing to be triggered; it starts automatically when a set of conditions are met.
- CLR is used to create and manage managed objects.
- Unmanaged objects are hidden or wrapped around the operating system resources.
- We can use the Dispose method and/or the ‘using’ statement to clean up unmanaged resources.

What's Next?

We now know what an object is, how it is created, how to use them, and we also learned what goes on behind the scenes – Stack, Heap, and GC.

The next step is to go back to Classes and learn how we can create our own objects and classes – this is where we really start to learn just what C# can do.

Classes Part 2

We now know what a class is and we know what an object is and how they are used. It’s time to look at how to create classes. This is a very important part of C# programming to learn and, by the end of this section, you will have an understanding of how to create a class – this is something you will do a lot of in programming so take your time to learn this.

What we are going to do is create a simple class called Player and this would be used for representing a player in a game.

Now, there is quite a lot to go through here so don’t worry if you struggle. Go back over whatever you need to and take your time over each section. With what you already learned about methods and what you will learn about classes, you will soon have the power of C# in your hands.

Creating a New Class

It’s time to make a class. Remember that classes are blueprints for the actions that a specific object type needs to track and do.

While you can place several classes in one file (as opposed to Java where each class can have only a single file), it is good practice to do just that – put each class into a file of its own. This keeps your files in more manageable sizes.

The first thing we need to do is create a brand new file in our project. Open Project Explorer in Visual C and right-click your project – the top-level item you see is the solution which is nothing more than a collection of projects related to one another. Choose the second item in Project Explorer – this should be your project – and, when the menu appears, click on Add -> Class.

A dialog box, ‘Add New Item”, will load and you will see the Class template has already been selected. Go to the bottom of it and type in a name for your class – mine is called Player.cs – the class is called Player. Now click on Add and watch your file be created.

Now, because you told your new file that you wanted a new class made, you should see some default code in your Player.cs file – it will look like this:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace CreatingClasses // Your namespace will probably be something different.

{

   class Player // If you named your class/file with a different name, that's what will be here instead of  Player

   {

   }

}
```

That's put us where we want to be, with a brand new class.

Adding Instance Variables

Right now, there is little in this class so the first thing we do is add a few variables. Remember, one of the main points of an object is that they track and monitor their own data. These are stored as variables for the object. Because each class object or instance has its own data, we call the variables by the name of instance variables. Let's say that your game has two players; each can track their own scores by themselves inside those instance variables.

For our simple class, we want to track three things – the name of the player, the number of lives they have left, and their score. We want to create one instance variable for each of these so, inside the curly braces of our class, we add the following code:

```
private string name;
private int score;
private int livesLeft;
```

Right now your code should be looking something like:

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

namespace CreatingClasses
{
   class Player
   {
      private string name;
      private int score;
      private int livesLeft;
   }
}
```

The first thing is the private modifier, which we will be talking about in a minute, and the rest is the same as you would create any variable – we declare the variable type, and the variable name. Up to now, most of the variables we have created have been called 'local' variables, apart from those in a method definition; these are called parameters. That makes the instance variable the third type we have looked at – are you keeping a note of them all?

Access Modifiers: *Private* and *Public*

Now, back to that private keyword. Everything that is in a class will have an accessibility level, which is what indicates where it can be accessed from. With the private keyword, we are saying that the variable may only be accessed, which means changed or used, from within the class itself. Nothing that is outside of the class will even recognize that the variable exists.

You will also see the public type. With the public access modifier, we are saying that the variable can be seen from outside the class. What making a variable public is that it can be modified by anyone from anywhere and that is not always a good thing.

In general, classes should be responsible for all the changes made to their own instance variables – that job should not be done by anything that is external to that class. This way, everything that the class has responsibility for is kept inside the class and protected.

As an aside, if you do not include one of the access modifiers, the variables are made private by default. This is the default level of accessibility for all class members. While you don’t necessarily need to specify this in our case, it is good practice to use them, not only to get into the habit but so that anyone who sees your code knows immediately that it is private.

There are other access modifiers but we don’t need to discuss those now as they are a bit more advanced, we can look at them later. For now, we will concentrate on the public and private ones.

Adding Constructors

Ok, we have our instance variables so now we need to add some constructors. When we make these, we want to look at what instance variables we have and give some thought to how we want new objects created and what should be set immediately. For example, it doesn’t make any sense to create players without giving them names. So, what we want here is a constructor with a parameter of a name. We may also need to specify how many lives a player starts with but, where the score is concerned for a new Player, it will almost always be 0 to start with.

For our class, we are going to want a constructor that takes one parameter as a name only and maybe one that has a name and a number signifying the number of lives. Below your instance variables, add in this code:

```
public Player(string name)
{
    this.name = name;
}

public Player(string name, int startingLives)
{
    this.name = name;
    livesLeft = startingLives;
}
```

Your code will now look like this:

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

namespace CreatingClasses
{
    class Player
    {
        private string name;
        private int score;
        private int livesLeft;
        public Player(string name)
        {
            this.name = name;
        }

        public Player(string name, int startingLives)
        {
            this.name = name;
            livesLeft = startingLives;
        }
    }
}
```

Notice that the public access modifier was used. Why? Because we want people to be able to create Player objects from outside of the class. You may, at times, want to make your constructors private but, for our purpose, for now, they will all be public.

All of the constructors that we are going to create are going to have the same name that the class does in our case, they will be named Player. That is how the compiler is going to know that you are creating a constructor rather than a standard method. As well, we don't yet have a return type, i.e. we don't use something like public void Player(); this is another way of telling the compiler that we are using constructors not methods.

Next, the constructor parameters are placed inside parentheses, just like they are for a method.

Two other things come out of this example that we need to discuss – variable scope and the 'this' keyword.

Variable Scope

What we haven't discussed properly yet, where variables are concerned, is scope. Variable scope is referring to points in your code where a variable is valid, where it exists. In other words, where it can be used and referred to.

Let's say we have a method that we created. Inside that method, a variable is created, such as int aNumber = 3. We can use this variable anywhere within the method but it can't be used inside of any other method. This variable only has scope inside the method in which it was created, nowhere else.

Here's another example:

```
public void DoSomething()

{

   for(int x = 0; x < 10; x++)

   {

      Console.WriteLine(x);

   }


   x = -5;  // This will not compile because the variable called x is not
in scope.

}
```

What we have here is a for loop and inside is a variable with a name of x. That variable is only valid within the for loop. Once we exit that loop, the variable can't be used anywhere else or at any other time.

What we can do is create another for loop and have another variable inside it, a different one also called x but neither will conflict with each another:

```
public void DoSomething()

{

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


   for(int x = 50; x < 60; x++)
   {
      Console.WriteLine(x);
   }
}
```

The first scope type we looked at was called method scope but this one is called block or inner block scope.

The key is to allow the curly braces to guide you. Whatever is created inside the curly braces is usable anywhere within them.

The next logical step from here is the class scope. Take a look at the variable instances; they are set inside the curly braces and are usable within the class and that includes our constructors and the methods that we will be creating shortly.

Name Hiding

There is one thing that C# allows you to do – if you have a local variable with block or method scope or a parameter, you can give it the same name as something with class scope. When we created our class, we created an instance variable that we called name – this has class scope because it can be used throughout the class. Then we had a parameter within the constructor that we also called name and this has method scope because it can only be used in the method or constructor.

This creates something known as name hiding. While we can use the instance variable called name within the class, including our constructor, there is now another one on the constructor that has the same name and is also usable. Which one are we referring to when we reference it within the constructor? It would be the one that was created with method scope because it has the smaller scope.

What we have is a situation where a variable is hiding another one, based just on the variable name. That’s fine,but what happens if we want to access the instance variable that has the larger scope?

There are a couple of ways we can do this. First and simplest way would be to give each variable a different name so that one wouldn’t hide the other. There’s nothing wrong with that. We could have given the parameter a name of playerName rather than just name but trying to think of different names all the time will get tiring quickly because the names would still have to be similar enough that you know what they were.

A second way of doing things is the more common approach. It involves naming your instance variables in such a way that you know exactly what they refer to while maintaining consistency. For example, you could add m_ to the start of all the names of your instance variables, so you would have m_livesLeft, m_score, for example. Using this, your names are easily identifiable but are not the same as the method scope variables.

Let’s be clear – this is a common approach but it really isn’t the best one. Why? Because it does not add to the readability of your code. At the end of the day, we’re all about making that code easy to read. All of the em’s dotted throughout mean you have to sit and think about what the variable actually does. What about when you read your code out loud? You find yourself having to actually say “em underscore and then the variable name. However, at the end of the day, it all comes down to which approach you find easier.

There is another one that we haven’t discussed yet and, personally, I think this is the best one. Because of that, this is how I will be demonstrating to you throughout the code examples. If you want to use a different way, make sure you are consistent in changing things throughout your code.

So, what is it? Simply allow the name hiding to take place. Name your local variable or parameter and your instance variable exactly the same and use a specific keyword to refer back to your instance variables. That keyword is 'this' and is what we discuss next.

The *this* Keyword

Whenever you are in a class, there is one easy way to access everything in the class – the methods and the instance variables. The 'this' keyword is used to reference whatever object you are in at the time. So, to carry on with the example we started, in your Player class, whenever you want you can use this.name to access the instance variable called name. The same goes for methods; by using this.MethodName(), you could call any other method belonging to the class.

Most of the time you won't need to do this; C# will automatically assume that you are already referring to what is in the class if there isn't anything in the block scope of method scope by the name you are referring to.

Putting it all together, if you have name hiding in operation, using the 'this' keyword will let you get to the instance variable that has class scope regardless of whether there is a parameter or variable with method scope that would be hiding that instance variable.

Back to the code that we added:

```
public Player(string name)
{
   this.name = name;
}


public Player(string name, int startingLives)
{
   this.name = name;
```

```
    livesLeft = startingLives;
}
```

In both methods, we have a parameter that has an identical name to the instance variable. Using the ‘this’ keyword, we access the name variable for the class. When we only say ‘name’, it is the parameter that we are referring to, So, using this.name = name; takes the name parameter value, passes to the constructor, and assigns the value to the name instance variable.

Ok, that’s enough of that for now, let’s get to our class.

Adding Methods

What we need to sort out now is which methods we might want to be defined by the class. We could decide that we want to get the name of the current player; we might want their current score, and we might want to add points. We might also be looking for a way of making that player lose a life, lose all their lives so they are ‘killed’ off, and to determine how many lives they have left.

Using the code below, we can add a method for each of these:

```
public string GetName()
{
    return name;
}


public int GetScore()
{
    return score;
}
```

```
public void AddPoints(int totalPoints)
{
    score += totalPoints;
}

public void Kill()
{
    // We need to ensure the player lives can't go into a negative
number.
    if (livesLeft > 0)
    {
        livesLeft--;
    }
}

public int GetLivesLeft()
{
    return livesLeft;
}
```

Ok, so your Player class should now look like this:

So now our completed Player class should look like this:

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

```
namespace CreatingClasses
{
    class Player
    {
        private string name;
        private int score;
        private int livesLeft;

        public Player(string name)
        {
            this.name = name;
        }

        public Player(string name, int startingLives)
        {
            this.name = name;
            livesLeft = startingLives;
        }

        public string GetName()
        {
            return name;
        }

        public int GetScore()
        {
            return score;
        }

        public void AddPoints(int totalPoints)
        {
            score += totalPoints;
        }

        public void Kill()
        {
            // We need to ensure the player lives can't go into a negative
number.
            if (livesLeft > 0)
            {
                livesLeft--;
            }
        }

        public int GetLivesLeft()
        {
            return livesLeft;
        }
    }
}
```

There are a couple of things that we need to talk about here. Previously, the methods we have created have all been marked as static. First, you probably noticed (and if you didn't, you should have) that all the methods here are public. What that means is that the methods can be called from outside the class, by anyone who has created instances of the class. If we wanted, these could have been made private and, if we had done that, the method would only have been callable from within another method that is within the class. In our case, we want anyone to have the ability to call the methods, which is why we made them public.

Second, there are a number of methods that begin with ‘Get’. One of the most common things in programming is to have some methods that will just return whatever the instance variable value is and to have another that sets the variable value and checking that the value to be assigned is a valid one. Here we have a lot of these GetSomething methods and some SetSomething methods. For the time being, that is what we will go with. However, as this is common, there is a feature in C# that makes all of this so much easier to read and that feature is called Properties, which is what we will be discussing in the next section.

For now, let’s go back to our static keyword.

Static Methods, Variables, and Classes

Up to now, we’ve been doing a lot of work with variables, specifically instance variables, and with methods belonging to class instances. What this means is that several instances of a class can be created and each will have its own set of data. When you call one of them, it will do the work it needs to by using a set of variables specific to it.

However, previously, all of our methods were static. What does that mean? Any class-level method or variable that has the static keyword belongs to the entire class, not just one instance of it.

As far as variables go, any instance that uses the variable will have the exact same values. For example, if an instance changes the value of a static class value to, let’s say 4, another instance will be able to see the new value and this is because the variables are shared with every class instance.

For methods, there is no need to have a class instance to call it because it will already belong that that class.

To make this clearer, a method that is not marked static would be accessed like this:

```
Player player = new Player("Rob Roy");
player.AddPoints(100);
```

Methods that have the static keyword are directly accessed with the name of the class. There is no need to create class instances to use the methods in a class and we saw that very clearly with our Console class:

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

WriteLine is static so there isn’t any need for a Console object to be created (this would be an instance of the class) to use that method.

One last note – the static keyword can also be used on classes. If you do, it will mean that the class is not able to have methods or instance variables, though. All of them have to be static and you can be sure that the compiler will check this. The Console class is only one example – it is not possible to create instances of it but then, you don’t need to because it only has static methods.

There is one last thing: we want to make sure we can see how the Player class is used. So, in the Main method, create a Player class instance and use it like this:

```
Player player = new Player("Jekyll", 3);


while (player.GetLivesLeft() > 0)
{
   player.AddPoints(100);
```

```
    player.Kill();
}
```

Quick Primer

- Classes usually go into a file of their own, even though C# doesn’t require this to happen – it’s just easier.
- You can add variables to classes and they are known as instance variables.
- You can add however many constructors you want to a class.
- Methods can also be added to classes and these will work on the instance variables in the class.
- When you use the private keyword with instance variables or methods, you are saying that it can only be used from within the class.
- When you use the public keyword with an instance variable or a method, you are saying that it can be accessed from anywhere, in or out of the class.
- We also looked at name hiding, variable scope and the ‘this’ keyword.

What's Next?

Wow, that was quite intense and there is quite a lot there but, between all the sections so far you now have a good understanding of the core of C#. From here on, everything else could be considered more fluff. Well done for making it this far!

In the next section, we look at properties.

Properties

In the last section, we talked about how on many occasions, you will want to ‘get’ or ‘set’ an instance variable value. The result of that is that you will likely be adding lots of Get() and Set() methods in your code.

In C#, there is a cool feature that makes it very easy for you to create the ability for another programmer to get or set that value and that feature is properties. We’re going to look at what these can do and how to create them in multiple ways before looking at how to set multiple properties right at the start of creating an object instance.

Why Properties?

Let’s assume that you are in the process of creating a class. So far it looks like this:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace Properties

{

    class Player

    {

        private int score;

    }

}
```

Now we want to let something that is outside of that class set a score value and get the current score value. The usual way would be to create a method that will return the score value and then create another one that sets the score value; you might add in some additional logic to make sure that the score is a valid value, i.e. isn't negative.

This is how that would look:

```
public int GetScore()
{
    return score;
}

public void SetScore(int score)
{
    this.score = score;
    if(this.score < 0)
    {
        this.score = 0;
    }
}
```

This is common in programming but one of the most frustrating things is having to keep typing SetScore(). In fact, it would be very tempting to set score as public rather than private; all you would need to do then is type score = and the value. That's a convenient way of doing things but there is a problem with it. Now that your instance variable is public, anyone can access it and modify it any way they want.

Properties are the way to solve this. Let's look at how to create them.

Creating Properties

Properties are nothing more than a simple way of creating get and set methods that are easy to read. Rather than using two methods as we did in the last example, we would do this:

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

Now we have a property called score. It is public, which means anyone can use it – you can set it to private if you want. We’ve also stated what type the property is, in our case, it’s an int and then we named it.

Next, the get and set keywords are used to indicate what happens when anyone attempts to read or set the property value. Inside the curly braces, you see code that is pretty much the same as what was in our GetScore() and SetScore(int) methods but there is a difference.

The normal SetScore() method lets us pass in and set a variable value; with a property, we make use of another keyword, value instead and this will take on whatever value is being set when the property is called.

Now, when we call our property, we can do it like this:

```
Player player = new Player("Player");

int currentScore = player.Score;

player.Score = 50;
```

If you tried to read what's in the Score property, whatever is inside the property's get block would be executed and the value returned. If you tried to assign the Score property with a value, whatever is in the set property will be executed. In our case, the value keyword in the set block would be assigned a value of 50.

Properties make it easier to get and set variables because they can also do the validation work when a user attempts to set something, all without messing up the readability of the code.

It is worth knowing that properties really aren't anything more than "syntactic sugar" and all C# does is turns them into methods that do exactly the same thing.

There is one more thing to talk about here to clear things up. It isn't necessary to have your properties belong to any specific instance variable. In the code above, that is exactly what we have done, but you don't need to do that.

Here's an example that doesn't have the property belong to an instance variable:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace Properties

{

    class Time
    {
        private int seconds;

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

        public int Minutes
        {
            get
            {
                return seconds / 60;
            }
        }
    }
}
```

What we have here is an instance variable named seconds and, with it, a property named Seconds. Just a note - it is very common to use lower-case names for your instance variables and upper-case for the matching property.

We also have a property called Minutes and that one stands alone, not belonging to an instance variable. If you look, you will see that there is no instance variable called minutes; what we have done instead is returned the result of the number of seconds after being divided by 60.

This also shows us that there isn't really a necessity to provide a get and a set accessor for our property. You need to use one of them although which one is down to you and your code. If you use a getter, the code is marked as read-only – it has not been set. If you only use a setter, that would be odd as you would be setting a value that you would never be able to read, so what good is it?

Accessibility Levels

Up to now, our properties have all been set with public access which means that anyone can access the getter and setter (get block and set block). And the example above shows only a getter being set.

There will be times when you have a property and you want the getter and setter to be set to different levels of accessibility. For example, you might want the getter set as public and the setter as private. We can do this very easily simply by specifying an access modifier in front of the get and set keywords – like this:

```
public int Score

{

   get  // This is public because we set the property as public

  {

      return score;

   }

   private set // This one is now set to private

   {
        score = value;
    }
}
```

Here, anyone can use the getter but the setter can only be accessed from within the class it is in.

Auto-Implemented Properties

You will also find yourself creating quite a lot that looks like this:

```
private int score;


public int Score
{
    get
    {
        return score;
    }
    set
    {
        score = value;
    }
}
```

With C# v3.0, a new feature was introduced which allows the ability to make auto-implemented properties that do the same things. Instead of using that code above and that includes our private instance variable called score, you would just do this:

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

Behind the scene, a default instance variable is created along with the basic code for get and set. What you won’t have is access to the instance variable marked private because it simply doesn't exist. It is, however, an easy way of creating a simple property.

Setting Properties at Object Creation

When we talk about properties, we should also take something else into account. Let's take our Player class where we have instance variables for the name of the player, the score, and how many lives are left. Let's assume that we want to create a property for each one and these will be called, respectively, Name, Score, LivesLeft.

When a new Player object is created, the syntax below could be used to assign values to each property at the time the object is created:

```
Player player = new Player("Jekyll") { Score = 100, LivesLeft = 5 };
```

This means we don't have to do this:

```
Player player = new Player("Jekyll");

player.Score = 100;

player.LivesLeft = 5;
```

Instead, it can all go neatly on one line.

Quick Primer

- Properties give us an easy and quick way of creating our getters and setters
  for the instance variables.
- You can use something like the code below to create a property:

```
public int Score

{

   get

   {

      return score;

   }
```

```
    set
    {
        score = value;
        if (score < 0)
        {
            score = 0;
        }
    }
}
```

- You won’t need both a getter and a setter for all properties.
- You can set different access modifiers for the getters and setters.
- You can create auto-implemented properties to define simple properties that have default behavior very quickly (public int Score { get; set; })

What's Next?

That pretty much covers everything important that you need to know about properties. Next, we’ll look at something that is much like a class in appearance but has very different implementation – the struct.

Structs

Now that you understand classes, we can look at a construct that is very similar – the struct. This word comes from C/C++ and is from the word ‘structured’. A struct is representing a record or a set of data that has been structured – in C language, this marked the beginning of OOP and lead to classes in C++.

Structs are much like classes and you could be forgiven, at first glance, for mistaking one for another. However, there are some significant differences. To really understand structs, it is worth going back over the section on the Stack and Heap and on passing references by value – if you need a brush-up, now is the time to do it.

Creating a struct

This is much like creating a class and the code below will define a struct and a class that is identical and does the exact same thing:

```
struct TimeStruct
{
  private int seconds;

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

```
        }
    }

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

class TimeClass
{
    private int seconds;

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

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

As you can see, they are fairly similar and we have used the same code in both of them – the only difference is in the keywords – struct to create the struct and class to create the class.

What's the Difference?

As they are much the same, no doubt you are asking yourself what the real differences are.

The first difference is that, where we have a constructor inside a class, we don’t need to assign all the data in the class for it to take a value. For example, our TimeClass in the above example might contain a constructor looking similar to this:

```
public TimeClass()
{
    // We don’t need to put anything here; the instance variables will simply be initialized
    // to their default values which, in many cases, will be 0.
}
```

Where the struct is concerned, on the other hand, all instance variables must be assigned a value:

```
public TimeStruct() // This part broken. See below.
{
    seconds = 0; // This is required, or the program will not compile.
}
```

Another major difference is that there is no way of creating a constructor without parameters for the struct.

Remember classes – if we didn’t put any constructors in, a default one without parameters was provided. With classes, it is possible to create any constructor you want and you can provide an implementation for those without parameters. When you add a constructor, the compiler will not add the default one for you.

With the struct, there is already a default constructor with no parameters but it cannot be replaced with the one you create. Even if you added new constructors, that default one will not go away – it is always going to be there.

So, from the viewpoint of writing the code, there are a couple of minor differences and there is a major difference,

A class is a reference type but a struct is a value type. Cast your mind back to when we talked about variables, about how they were containers for storing information. With reference types, the container will hold a reference to a location in memory where the full data is stored. With the value type, all of that date is in the container.

That raises one or two side effects.

For a start, when two variables are assigned with identical values, in a value type like the struct, there will be a complete copy of the relevant data in each variable. When you modify one it won’t change the other. However, with reference types like the class, each variable has a copy of the reference but both will be pointing at the same memory location. When one is modified, those changes can be seen in the other one’s reference.

It will take a bit of getting used to and I don’t expect you to grasp it straight away. However, you need to always keep in mind whether you are using reference or value types because each has its own behavior.

Most of the types built-in to C#, such as the int, bool, long, short, char, decimal, float, and byte, along with enumerations and anything that has been defined as a struct will be a value type. Arrays, strings and anything defined as a class is a value type.

Quick Primer

- Structs and classes are very similar to one another.
- The struct is created with the struct keyword rather than the class keyword.
- Structs are created on the Stack, not on the Heap and this makes them perform faster but it also leads to them being passed by value and not reference.

What's Next?

While the struct and the class are similar, they do have differences, with the main one being allocation of the struct on the Stack and not the Heap. As this means it is passed by value, the method gets a copy of the struct, not the original.

It is a nice feature and you do need to know about them but the likelihood is you will use classes way more than you ever will use a struct.

Next, we are going to delve into inheritance.

Inheritance

Let’s imagine that you have many geometric polygons that you want to track. That would include squares, triangles, rectangles, pentagons, and so on. You could create a class called Polygon that contains the number of sides on the polygon and perhaps also has an array in it that contains the vertices or corner positions of the polygon.

You could use this Polygon class to represent any one of the polygons but, using a square as an example, you might want to do something else with it. For instance, you might want to create a square using a single size – the height and width of a square are the same lengths – or you might want to create a method to return the square area.

One way of doing it would be to create a class called Square and this class would contain everything from the Polygon class but everything else you need. So, what’s the problem?

The problem is that you now have a class that is completely different and you if wanted to change how polygons were worked with, you would also need to change how squares are worked with. On top of that, while you could have an array of polygons (Polygon[] polygon) there is no way of adding squares to it - yes, I know that a square is a polygon and we should be able to treat it as such.

This leads us to the section on inheritance.

A square is most definitely a polygon but it is a bit more specific than that – a square is a special polygon type that has special attributes. Whatever a polygon can do, a square can do it too because, yes, that square is a polygon.

This is known, in programming circles, as an ‘is-a’ type of relationship and it is such a common thing that most programming languages facilitate it by creating constructs.

The C# construct that lets us do this is known as inheritance and this lets us create that class called Polygon and create a class called

Square based on our Polygon class; from this, we can reuse everything in the Polygon class. So, any changes made to Polygon would be reflected automatically in our Square class.

Base Classes

Base classes are often also called parent classes or super-classes. We'll stick with calling them base classes for now. A base class is much like a standard class but it is used for the purposes of inheritance by another class. In our case, the base class is the Polygon class.

For example, our class could look like this:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace Inheritance

{

    class Polygon

    {

        public int NumberOfSides { get; set; }


       public Polygon()

        {

            NumberOfSides = 0;

        }


        public Polygon(int numberOfSides)
        {
            NumberOfSides = numberOfSides;
        }
    }
}
```

Derived Classes

Derived classes are also known as subclasses and are based on other classes. Our Square class is a class that was derived from the Polygon class. Derived classes are created in much the same way as a standard class but with one difference – that difference is what tells the program which of our classes is the base class.

A standard class is created like this:

```
class Square
{
    //...
}
```

The colon (:) is used as an indication that the Square class has been derived from our Polygon class; this is followed by the base class name:

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

All we do in the class is to indicate anything new for the Square class that is not in the Polygon class. Our Square class will look like this:

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

namespace Inheritance
{
    class Square : Polygon
    {
        public float Size { get; set; }

        public Square(float size)
        {
            Size = size;
            NumberOfSides = 4;
        }
    }
}
```

A square now has a property for size but it will also inherit all the methods, properties, and instance variables from the Polygon class, and that means it will inherit a property called NumberOfSides.

Another thing to mention is that classes can inherit from classes that are already inheriting from another class. Basically, the number of layers you have is unlimited.

Using Derived Classes

We need to talk about something important with regards to using a base class. You already know that you can create two objects in a program and the same applies here – you can have a Polygon object and a Square object:

```
Polygon polygon = new Polygon(3); // a triangle
```

Square square = new Square(4.5f); // a square, which is a polygon that has 4 sides of length 4.5.

However, our Square is already a Polygon so you could also include a variable with a type of Polygon that stores a Square. Are you still with me?

Polygon polygon = new Square(4.5f);

While the program is running and while the Polygon variable is being used, we can only access what is inside Polygon. What we could do is check what NumberOfSides is but we would not be able to find out if a Square or a Polygon, or any other type, perhaps a parallelogram, is being tracked by a variable of Polygon type. All we know is that it is a Polygon and all we can work with is what the Polygon has.

Checking Type and Casting for Objects

As we just discussed, we can work with derived types, such as Square, but we can only know what it is – a Polygon. There will be times that we will want to work out what an object type is so we can work with it in a different way. We can do this by using the 'is' keyword and using casting:

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


if (polygon is Square)

{

   Square square = (Square)polygon;

   // Now we can do whatever we want with the square.

}
```

Using Inheritance in Arrays

One more thing that we should talk about is creating arrays of the base class. An example of that would be Polygon{} lotsOfPolygons = new Polygon[5];. Then we could put a derived class in it – lotsOfPolygons[2] = new Square92.1f);.

Derived classes are able to function the same way as their bases classes do.

Constructors and Inheritance

All the properties, methods, and instance variables in a derived class are inherited. The constructor is somewhat different.

Constructors from a base class cannot be used to create derived classes. For example, the constructor in the Polygon class is able to take an int as the value of the number of sides but a Square cannot be created from that constructor.

There is a good reason for that – it makes no sense. If you could use something along the lines of }}Square square = new Polygon(6);}} to create a new Square, setting it as 6 sides, it would all be a bit messy.

Object instances can only be created from constructors defined by the derived class.

There's more to the constructor - because derived classes have been based on other classes, we would also need to get a constructor from the base class and the default is the construction with no parameters – the Polygon(), for example.

Don't forget, whether you have declared constructors or not, the compiler will create a constructor with no parameters by default.

You don't need to use this if you don't want to or you may have a base class that does not have one of these parameterless constructors.

Either way, you have the choice of any constructor for the base class in the constructors for the derived class and for that, you use the base keyword. The constructor we created in the Square class originally would be defined like this:

```
public Square(float size)
        : base(4)
{
    Size = size;
}
```

We add a colon and the base keyword together with the base class parameters that we want to make use of. By doing this, we can get rid of the parameterless constructor from the Polygon class – this constructor is responsible for letting programmers create Polygons with no number of sides specified!

Protected Access Modifier

Previously, we talked about the public and the private access modifier; public means that the variable or method is available to anyone, while private means it can only be accessed from within the class to which the variable or method belongs.

Inheritance gives us a third option – the protected access modifier. If you mark a method, variable, or property as protected rather than public or private, it can only be accessed from within the class or any class derived from that class.

So, if the NumberOfSides property in our Polygon class was protected rather than public, it could be accessed from the Polygon class, the Square class, and any other derived class but not from anywhere else.

The Base Class of Everything

Even if you didn’t specify a class for inheritance purposes, every class you create is already derived from a base class. This is the base class of everything and it is the class called object.

There is also an object keyword that works much the same as the float or int types we’ve used but the object keyword always represents objects. And, because it is a class, it is a reference type and not a value type. However, the inheritance properties dictate that it can be used for representing any object.

Remember earlier, when I said that you could store derived classes in base types, like this:

Polygon polygon = new Square(4.5f);

Well, it can also go in the object type:

object anyOldObject = new Square(4.5f);

Because object is the base class of every class, you can store any object type in it.

On top of that, the object class defines several methods that all objects will have. There are about four of these methods but the one that means the most is a method called ToString(). Every object you ever create will have a ToString() method and that means it can be called whenever you work with any object.

Preventing Inheritance

You might come to a point where you don't want a specific class to be inherited from, which means you don't want that class derived from.

To facilitate this, there is another keyword to use – sealed. Adding it the class definition prevents that class from being inherited from:

```
sealed class Square : Polygon
{
   // ...
}
```

Plus, as an added bonus, this can provide a boost in performance.

By default, a struct is sealed and there is no way to unseal it; this means it is not possible to do inheritance with any struct.

Quick Primer

- Inheritance helps us to reuse a class by expanding it into another specific
  class type. For example, the Square class can be inherited (derived)
  from the Polygon class and that means it will have all the features from

the Polygon class.
- You can use any class as a base class.
- To specify a base class from another class, the colon is used followed by
the name of the class, i.e. class Square : Polygon {/*…*/}
- Using the protected access modifier on something means it can be
accessed from within the class and from a derived class.
- The sealed keyword stops a class being inherited from, i.e. sealed class
Square {/*…*/} means the Square class cannot be inherited from.

What's Next?

Inheritance is a powerful thing in programming. Not only does it help us to get our code more organized, it means we can reuse our classes. It is, however, quite complicated and we’re going to carry on with it by discussing abstract classes and polymorphism next.

Polymorphism, Virtual Methods, and Abstract Classes

In the last section, we looked at inheritance but there are still many more things to go through regarding this feature. In this section, we'll be looking at some of the more advanced things, starting with polymorphism. We'll look at what it is, how to use it with virtual methods and with overriding methods before taking a look back at the base keyword.

We will then move on to abstract classes (those that cannot be created; instead, derived class instances are needed) and we'll end by looking at the new keyword again and another use for it.

What is Polymorphism and Why Do We Need it?

You will hear the word, polymorphism, quite a lot as a programmer. It is a Greek word and it literally means "many forms". Now that we are looking at inheritance, it's time to discuss what it is and why it is such a useful feature.

Let's assume that we are creating a game. We have lots of different player types; one is human and can use the arrow keys to navigate around the screen. You might also have a player controlled by the computer that can also navigate the screen but in a completely different way. To keep things simple, we'll say it moves randomly.

We could start by creating a new Player class and giving it a method named MakeMove(). This method would return a direction the player is to move in. Then, taking what we learned in the last section about inheritance, we could make a derived class from this and call it HumanPlayer. This will take care of making moves by seeing which keys are pressed by the user and another derived class, called RandomAIPlayer would simply pick a random direction in which to move.

All of this describes the core of polymorphism. We've got our base class, which is called Player, and this class knows that moves need to be made; for our point of view, how that happens specifically is not particularly important. We also have two derived classes – RandomAIPlayer and HumanPlayer – and we want these to implement the method by using whatever means is required. These two classes can each have completely different methods for doing this and that leads to many forms of achieving the end result. That is exactly what polymorphism is.

If we also had an object called Player, we could call the MakeMove() method from outside both of those classes and the player will learn how a move is to be made using what it needs to and returning a result. How it reaches the decision is not important; all we need to know is that the decision on how to move was reached.

With C#, polymorphism is quite simple to do.

Virtual Methods and Overriding

We'll begin by creating the Player class:

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

namespace Inheritance
{
   public enum MoveDirection { None, Left, Right, Up, Down };

   class Player
   {
      public virtual MoveDirection MakeMove()
      {
         return MoveDirection.Left;
```

```
        }
    }
}
```

What we have done is define an enum called MoveDirection; this is used to define all the directions in which a player can move. Next, we have the class called Player, made the same way you make any class in C# and then the MakeMove() method is defined – this uses the enum that we made as the return type. Next, we added the virtual keyword.

What this keyword means is that our derived classes can change how that method has been defined and provide a definition of their own and this is what lets us do the polymorphism.

Let’s see what our HumanPlayer class looks like:

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

namespace Inheritance
{
    class HumanPlayer : Player
    {
        public HumanPlayer()
        {
        }

        public override MoveDirection MakeMove()
        {
            ConsoleKeyInfo info = Console.ReadKey();


            if(info.Key == ConsoleKey.LeftArrow) { return
MoveDirection.Left; }
            if(info.Key == ConsoleKey.RightArrow) { return
MoveDirection.Right; }
            if(info.Key == ConsoleKey.UpArrow) { return
MoveDirection.Up; }
            if (info.Key == ConsoleKey.DownArrow) { return
MoveDirection.Down; }


            return MoveDirection.None;
        }
    }
}
```

We created a class in a normal way and we derived another class from it. The MakeMove() method is created and it will do what we need it to do for our HumanPlayer class. To do this, the override keyword is added to the MakeMove() method.

Now, if we happened to be working with an object called Player that is a HumanPlayer and MakeMove() was called, the new piece of code would be executed rather than the code that was in the base class. In this example, the code will read whatever input comes from the keyboard, i.e. the keys that the user presses. The new definition for the method in this class will override the definition in our base class.

Similarly, the RandomAIPlayer class can be created and we implement the MakeMove() method in yet another way:

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

namespace Inheritance
{
    class RandomAIPlayer : Player
    {
        private Random random;

        public RandomAIPlayer()
        {
            random = new Random();
        }

        public override MoveDirection MakeMove()
        {
            int choice = random.Next(4);

            if (choice == 0) { return MoveDirection.Left; }
           if (choice == 1) { return MoveDirection.Right; }
            if (choice == 2) { return MoveDirection.Up; }
            if (choice == 3) { return MoveDirection.Down; }

            return MoveDirection.None;
        }
    }
}
```

To see how this works, we need to look at the code that you could put somewhere else in the program to make use of it:

```
Player player1 = new HumanPlayer();
Player player2 = new RandomAIPlayer();

MoveDirection player1Direction = player1.MakeMove();
MoveDirection player2Direction = player2.MakeMove();
```

At this point, how the player decides the direction to move is not important; all we need to know is that the move was made and the game can progress. Each of the player types will work the details out by themselves leaving the code that uses the player type to focus elsewhere.

In our case, where a definition was provided for the method by the base class, Player, the derived class doesn’t have to override that method; instead, you could just use the definition from the base class. Up to now, we have done this in such a way that we don’t have to override the method, but we can if we want.

Back to the Base Keyword

Let’s go back to our base keyword for a minute. Assuming that you are currently in a derived class, and you are in the process of overriding a specified method; the base class gave us a form of default implementation for that method and, in most situations, this would be perfectly valid. However, there will be times when you want something different done and that is where method overriding comes in.

But wait a minute; that base class has done loads of work and we want to reuse some of it. To do that, within any of your methods, the base keyword can be used to have access to the base class methods:

```
public override MoveDirection MakeMove()
```

```
{
   MoveDirection moveDirection = base.MakeMove();
   if(moveDirection == MoveDirection.None)
   {
      return MoveDirection.Up;
   }
}
```

When you use the base keyword like this, you can still access the original implementation of the method while, at the same time, overriding it so something else can be done.

Abstract Base Classes

Back to our Player class example.

This class is defining what our player may do but we might not want just anyone to have the ability to create a Player. We may want them to make RandomAIPlayer or HumanPlayer instead, or some other type of player that we haven't yet defined.

There is a way of ensuring that instances cannot be created of specified class types, like the Player class, and that is to use the abstract keyword:

```
abstract class Player
{
   public virtual MoveDirection MakeMove();
}
```

When you use the abstract keyword, you make sure that a normal Player cannot be created – you would not be able to make use of Player player = new Player();. Instead, you would need to use Player player – new RandomAIPlayer(); or Player player = new HumanPlayer;.

Abstract Methods

Staying with the example above, we have a Player class defined with a method called MakeMove(). We only added this because, when a method is added, we need to implement it and it needs to know exactly what it does.

If we marked a base class as abstract (using the keyword), you might find that you have methods which don’t really make any sense to implement, such as the MakeMove() method. You might also find that, rather than a default definition, you want derived classes to have their own implementation.

In an abstract class, the abstract keyword can be attached to any method without a definition being provided. This is much like the virtual keyword but, rather than a method implementation being provided, the declaration is ended with the semicolon:

```
public abstract MoveDirection MakeMove();
```

With that, there is no need for the Player class to have a default definition; the RandomAIPlayer and HumanPlayer derived classes will have to have their own method implementation instead.

The *new* Keyword with Methods

One topic that seems to trip a lot of people up in relation to virtual and override keywords for methods is the new keyword.

The new keyword can be used on methods in this way:

```
public new MoveDirection MakeMove()
{
   //...
}
```

By using the new keyword, you are making it clear that you are creating a new method inside the derived class; that method has nothing to do with the base class method of the same name. The MakeMove() method here is nothing whatsoever to do with the MakeMove() method inside the Player class – this is not a good idea for most cases because now there are two methods that have the same name and the same signature but are totally unrelated.

Often, the new keyword is confused with overriding abstract and virtual methods and there are two reasons for that – first, it looks much the same. The new keyword is being used the same way that override is used. Second, if you omit the new keyword and the override keyword, when there is meant to be one of them there, the compiler warns you, telling you that one of those keywords needs to be used; if you leave the keyword out, by default it will just do what the new keyword already does.

Quick Primer

- Polymorphism is a Greek term that translates to "many forms". In OOP, it means that derived classes can be created to implement methods in different ways from one another and from the base class.
- Base class methods can be marked virtual so that the derived classes can use the override method and provide their own implementation.
- Anyone using the derived or base class can call the base class defined method and the compiler automatically calls the right method in the derived class. This means that the code doing the calling needn't worry about the implementation that is being run.
- You can mark a class as abstract so that an instance cannot be created of it. Instead, a derived class instance would need to be created instead.
- Methods in abstract classes can also be marked abstract and no method implementation would be provided. The abstract method must be implemented by the derived class though, or a compile-time error is thrown.
- When you attach the new keyword to any method, you are saying that you are creating a new method that is not related in any way to any base class method that has the same name.

The result of this is two methods with identical names, with the original method being hidden inside the base class.

What's Next?

All of this – polymorphism, inheritance, abstract base classes, and virtual method – can all be quite tricky to get the hang of so don’t worry if this went over your head. Come back to it later and read it again. There is a good chance that it will only really start to make sense when you use it in your own programming.

In the next section, we’re going to take a look at interfaces, the last topic in object-oriented programming.

Interfaces

In our last OOP topic, we are going to look at C# interfaces. These are much like the abstract classes we already used earlier so we'll start off by talking about what they are, and how to create one, before talking about multiple inheritance, the lack of support for it, and how we can use an interface to get a similar effect.

What is an Interface?

As a concept, an interface is nothing more than a list of things seen by the outside world, presented by the object, and allowing users/players to interact. Take your television, for example. No doubt it has buttons on it and they are how you interact with it. The interface for an object can be seen as a contract between the object and the outside world, which allows it to do specific things.

In C#, we have a built-in construct called an interface and this does exactly the same – provides a list of methods that a class that makes use of what the interface must provide.

In the last section, we looked briefly at abstract classes and the interface is very similar. In fact, if you had one of the abstract base classes and every method in it was abstract too, and your class didn't have any properties or methods, you would pretty much have an interface. That kind of abstract class is called a pure abstract base class in C++ because it has no interface construct built-in.

How to Create an Interface

If you can create a class, you can create an interface because they are incredibly similar. Let's assume that our program needs to write files out (we'll discuss that later) but you also want the ability to write multiple file formats. A different class can be created for each type of file writer but all of them will do pretty much the same as far as calling code goes. Each will have the exact same interface, so we create a file writers interface like this:

```
using System;

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

namespace CSharpTutorial
{
   public interface FileWriter
   {
      string Extension { get; }

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

As you can see, this is a lot like a class with a couple of differences. First, the interface keyword is used rather than the class keyword. Second, as you should have spotted, all the interfaces and methods look a lot like abstract methods with no method body. Did you spot that the virtual keyword is nowhere in sight? Nor have we marked them as public, protected, or private. Because we are creating an interface, it goes without saying that our methods will not be implemented and that they must all be public methods – otherwise, no one would be able to interact with them.

Using this interface, we can now create some classes.

We’ll start with a text file writer:

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

```
namespace CSharpTutorial
{
    public class TextFileWriter : FileWriter
    {
        public string Extension
        {
            get
            {
                return ".txt";
            }
        }

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

Making a class or implementing an interface is done using the same notation used for deriving classes from base classes. The colon (:) is followed by the interface name.

Within the class, a public method needs to be created with the exact same name, return type, and parameter list that the interface has.

What we can do is create some other properties, methods, and instance variables as we need them. There are no limits to what an interface can define; all we need to do is make sure that we have that definition as an absolute minimum.

In many ways, the interface works like a base class. For example, we could have a list of the FileWriter interface and add in different object types that will implement this interface:

```
FileWriter[] fileWriters = new FileWriter[3];

fileWriters[0] = new TextFileWriter();

fileWriters[1] = new RtfFileWriter(); // this type has not yet been implemented anywhere

fileWriters[2] = new DocxFileWriter(); // neither has this type


foreach(FileWriter fileWriter in fileWriters)

{

   fileWriter.Write("path/to/file" + fileWriter.Extension);

}
```

Most programmers use a capital letter to start the name of their interface and as you get used to using the standard C# libraries provided by Microsoft, you will see a lot is named this way, including IList and IEnumerable. Personally, I don't use that way of naming, simply because it doesn't make your code readable, but it is down to you and your own personal choice.

Multiple Inheritance

If you are familiar with some other programming languages, no doubt you have noticed that they may offer the ability to create one class derived from several base classes. Let's say that you have a class called Person; this provides age, name, and other things, and it also has a class called NoiseMaker. That class, in turn, has a method named RaiseTheRoof(). But you also have a class called Musician – this is both a noisemaker and a person and your first instinct could be to give your class both Person and NoiseMaker as base classes.

This is known as multiple inheritance and, sadly, C# does not support it. You can only derive from one base class.

What C# does do is it allows you to implement multiple interfaces. All you need to do is add every method that every interface will require. And you can inherit from one base class and implement the interfaces too.

There is a way around our Musician, NoiseMaker, and Person problem – simply create a NoiseMaker interface and a Person interface and let them both be implemented by Musician. You could also create Person as a class and have NoiseMaker be the interface – it would still work, and it would work the other way around too.

Implementation of multiple interfaces or deriving from one base and several interfaces simply requires you to list them with a comma separating each one:

```
public class Musician : Person, NoiseMaker
{
   // ...
}
```

The compiler looks to make sure the class is not attempting to derive from multiple classes.

Quick Primer

- Generally speaking, interfaces provide something that the external world can use for interaction.
- There is an interface concept in C# that works like a pure abstract base class – an interface is nothing more than a set of properties or methods that a class must provide if it implements an interface.
- Creating an interface is as simple as using the interface keyword instead of the class keyword.
- Interface properties and methods don’t need to be declared as public or abstract; it is a requirement that they already are both.
- If you want a class to implement an interface, the class is listed the same way that a base class is listed, i.e. class

ImplementingClass : Interface {/* the interface methods are listed here as public methods */}

- Multiple interfaces can be implemented or you can derive from one class and implement multiple interfaces – either work. What doesn't work is deriving from several base classes.

What's Next?

As far as classes go, there is one more thing to discuss – a powerful and pretty cool feature known as generics. This is a way of making classes that we can reuse at any time to hold any data type that we want, hence the name, generic. We'll kick off looking at why we would want to do this and look at using built-in classes that support and use generics before making our own.

Generics: Part 1

In this section and the next one, we're going to concentrate on a powerful C# feature called generics. If you have a background in C++, this is the same kind of concept as templates and, if you have a Java background, then you pretty much know generics already. If this is your first look, then fear not; they are not too difficult to grasp.

We'll talk about why we even have generics, what problems they are used to solve, and then we'll move on to the .NET framework classes that use them. These are called the List and Dictionary classes and both have a wide range of uses, the List class more so than Dictionary. As you learn to program, these are two classes you will put to good use, along with other generic classes.

Why Generics?

Before we talk about using generics, it would be helpful if we knew why we had them. So, let's start off by thinking about an underlying problem generics address.

We've been working with classes for a while now so you understand them. Let's say that we want to create a class that stores a list of numbers. I bet the first thing that pops into your mind is an array. Yes, ok, but this is a special kind of list; perhaps it's been sorted or something. Now, I know I also told you that there is a List class built-in but, for now, I want you to see how it all works so just go with the flow for a bit.

Now, let's think about how you would create this class to do what you want it to do. You could do this:

```
public class ListOfNumbers

{

   private int[] numbers;


   public ListOfNumbers()

   {
        numbers = new int[0];
    }

    public AddNumber(int newNumber)
    {
        // Add code here to make a new, slightly bigger, array,
        // bigger than it was before, and then add in your number.
    }

    public object GetNumber(int index)
    {
        return numbers[index];
    }
}
```

This is by no means finished but, hopefully, you get the gist of it. You simply create your class and you use that int type liberally.

Ok, but what if you now want a list of strings? It's sad that you can't use all that hard work, isn't it? You can't use your ListOfNumbers class and add strings to it so what do we do now? Think about it. Got any ideas?

We could probably make another class, similar, and name it ListOfStrings, couldn't we? It would look much the same as our ListOfNumbers class but it would have the string type and not the int type.

To be fair, you could go a little bit mad here and make all kinds of ListOf classes, one for each type. But wouldn't you find that irritating? You would have an almost unlimited number of these classes.

There could be another way. We could go down the route of making a list of objects. Why? Do you remember that the object is the base class of everything and of any object type?

All we would need to do is create one simple class. A List class that makes use of the object type:

```
public class List
{
    private object[] objects;

    public List()
    {
        objects = new object[0];
    }

    public void AddObject(object newObject)
    {
        // Add code here to make a new, slightly bigger, array,
        // bigger than it was before, and then add in your number.
    }

    public object GetObject(int index)
    {
        return objects[index];
    }
}
```

Look at that. Now we can add any object type we want into it. We have just one simple List class that will take any object type.

Now, at first look, this might look great. It works, doesn't it? Well, it isn't without its problems. For example, whenever you want to work with it, you need to cast whatever you are working with to the object type you want to add in.

If it were a list of strings and we were adding /strings to it, we would want to call our GetObject method so we do this:

string text3 = (string)list.GetObject(3);

Casting takes up valuable time because it isn't fast to execute and it will slow everything down. In addition, it is just a little bit irritating because you have to keep on doing it, throughout your whole program. That's fine if you only have a short piece of code but most are quite long.

There is another issue here, though, something much bigger. Sticking with adding strings to the list, we can do all the casting we want but, because we are working with a list of objects, we are not limited to just adding strings – we can put any type we want in there.

We could be extra careful and make sure that we don't, but accidents happen and even the best of programmers could accidentally add something else to the list, such as a Random class instance. There isn't anything to stop you from doing that. Really, what is wrong with using listAddObject(new Random()); even if we were only supposed to be adding strings? It might not even be you that adds something else – it could be anyone who is using that code.

In short, there isn't a way of knowing what object type is being pulled out of the list. You will always have to make sure you check that you are pulling out the type you think you are because it could be anything. As I'm sure you might have guessed, there is a name for this – programmers will say that it isn't "type-safe".

Type-safety is when you know, at all times, what object type you are working with. In our first example, it isn't something we would have had an issue with because all we had was a ListOf a specific type. We would know that we were working with ints or strings and we could make sure that we had the right object types all the time; we didn't even need to do any casting.

There are two bad things here. The first is that we make list classes that are type-safe, one for each type that we want and we call them ListOf whatever the type is. The second is that we create one class of objects that is not type-safe but eases the workload because we don't have to make a hundred different classes.

Obviously, this is leading to something isn't it? There is a better way to deal with all this and it's called Generics.

What are Generics?

This is a neat way of letting you make generic classes when you create a class – and that is created as in writing the code that defines it, not creating a class instance. Then, when you need to use a generic class, you just state the type you want, when you want it. So, you might want the generic class to be a list of ints and the next time you might want a list of Player objects.

Generics provide you with an easy way to make a type-safe class without having to state any specific type when the class is created.

One of the easiest ways to understand this is to see the classes that use this, so that's what we'll do. Starting with the List class, you will see it's quite close to what we already talked about and then we'll look at the Dictionary class which is a little more advanced and shows you some more of the generics features.

The List Class

Lists are one of the most common things in programs and C# made a neat List class to use. Let's look at how lists really work.

We have this class called List and you could think that this is the way to create your list:

List listOfStrings = new List(); // This doesn't work too well...

But you can't do that because it is using generics. If you want a List class instance you will also need to state what type is going into this list. To do that, you simply state the type you want inside a set of angled brackets (<>). So, if we wanted to work with a list of strings, it would look like this:

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

Now you have your list that contains strings and you can add things to the end of your list using a method called Add:

```
listOfStrings.Add("Hello World!");
```

Notice that because we are using the List<string> type, it is a requirement that you only add strings. The compiler will know that no other type can be added and that makes it type-safe. All we had to do was define one generic list class.

Really, that is the basics of generics covered, but this List class is one of the most useful so we'll take time to look at it closer.

There are a couple of ways of adding new items to your list. The first way is the Add method:

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

strings.Add("text1");

strings.Add("text2");
```

This method will add new stuff to the end of your existing list.

There is another way and it's called the Insert method. This lets you provide an index where you want the new item added and that will push all the items in the list back:

```
strings.Insert(0, "text3");
```

Remember, C# is zero-indexed so the first index is 0 – the line above adds the new item right at the beginning of the list.

You can also use a method called ElementAt to retrieve an item from the list.

```
listOfStrings.ElementAt(0);
```

This is an extension method and we will be looking at those later.

Another way of getting and setting items in a list looks something like an array – the square brackets. You could do this:

```
string secondItem = listOfStrings[1];
```

Or this:

```
listOfStrings[0] = "This will replace the current item at index 0";
```

There is also a method called RemoveAt to delete a specified item:

```
listOfStrings.RemoveAt(2);
```

If you want to delete everything from your List, you can use the Clear method:

```
listOfStrings.Clear();
```

To determine the number of items in a list, the array uses the Length property but with the List class, another property is used called Count – there is no Length property with the List class:

```
int itemsInList = listOfStrings.Count;
```

Let’s say that we have a List and we want to make it into an array. We can do this very easily using a method called ToArray. This will convert the generic class to an array, making sure it is of the right type.

```
List<int> someNumbersInAList = new List<int>();
someNumbersInAList.Add(14);
someNumbersInAList.Add(24);
someNumbersInAList.Add(37);

int[] numbersInArray = someNumbersInAList.ToArray();
```

There is one last thing to consider – looping over the items in a List class in much the same way as we do with an array:

```
List<int> someNumbersInAList = new List<int>();
someNumbersInAList.Add(10);
someNumbersInAList.Add(22);
someNumbersInAList.Add(35);

foreach(int number in someNumbersInAList)
{
   // ...
}
```

Keep in mind that, because List is a generic class, it is possible to create List classes for just about any class or type that you want to use. When you do that, methods such as ElementAt and Add will only work for the specific type in use; that is the whole point of generics.

The Dictionary Class

To finish this section on Generics, we'll take a quick look at the Dictionary class. This class uses Generics in a more complex way than the List class does but it isn't anywhere near as versatile. It does have its uses though and is a good one to learn.

What is the Dictionary class? If you already have a programming background, it is similar to a lookup table or a hash table. Let me explain.

To be honest, the class name gives it away; it works in much the same way as a dictionary. In any dictionary, you will have a list of words and each one has a definition, When you search for the word you want, you find the right definition and it's quite efficient. In short, you use a piece of text to find another piece.

The Dictionary class is like this – one piece of information, which is known as the 'key', is used to store information and to look up a piece of information, which is known as the 'value'. Basically, it is a map of key and value pairs. However, because we have a generic class, we don't need to only use strings; we can use whatever type we want.

For example, we can create a phone book using the Dictionary. The names of the people will be strings and their phone numbers will be ints. These may not be the best thing to use for the numbers but, for now, I just want to give you a simple example of how it works with two types.

We could create a Contact or a Person class that will store the names and all other information and a class called PhoneNumber to store the phone number details including area and country codes. However, we’re not going to do that because I want to keep it simple.

Like the List class, Dictionary is also generic, but instead of one, it has two generic types – the key used for looking up, and the value that we look up. Rather than putting one type between our angled brackets, we put both, one for each generic piece:

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

Now we can make use of the indexing operator to get a value or set a value in our Dictionary:

phoneBook["Bunter, Billy"] = 1526587;

phoneBook["Twain, Mark"] = 1625874;

int billsNumber = phoneBook["Bunter, Billy"]; // We're good friends...

As with our List class, we have several methods that can be used so take a bit of time and explore the Dictionary class, and see what can be done with it.

Quick Primer

- C# Generics are much the same as they are in Java and as C++ templates.
- Generics are used for creating type-safe classes without having to specify one specific type.
- When an instance is created of a generic class, you must indicate the type or types that you want to use for a specific

instance, using angled brackets (<>); for example, List<string> list of Strings = new List<string>();
- We also looked at the Dictionary class briefly – this is much like a hash or lookup table.

What's Next?

Generics is a powerful feature in C#, which gives the means to make 'container' or 'collection' classes that we can use for any object type. We looked at two classes that use generics – List and Dictionary – and, now that we have an idea of how they work, we can look at how to create them in our classes.

Generics: Part 2

We looked at the way generics work by using classes. Now it’s time to look at how you can make classes that will use generics.

We will create a version of the List class. Yes, it is already in existence but I want to show you how it all works and creating a List class is going to be the simplest way. One thing I want to say is that you mustn’t feel like you are limited to lists; generics work in many different places.

It is also worth mentioning that, while we are making a List class, you should really use the standard one, which already has all the methods and features you will need in a list and they have all been tested and work. Don’t make work for yourself by writing code that has already been written.

That said, that is exactly what we are going to do now.

To keep our list different from the standard List class, we’ll call it PracticeList, just so you don’t get confused.

Creating a Generic Class

Creating classes that make use of generics is much the same as creating a normal class. Begin by putting a new .cs file (class file) into your project and call it PracticeList.

On the code line where your class is declared, don’t forget to add the set of angled brackets and add int and name you are giving to your generic type (we’re using T) as you can see below:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace Generics
```

```
{
    public class PracticeList<T>
    {
    }
}
```

So far, this is pretty much what we would see with a standard class. The only exception is the addition of <T> and that is what makes this class use generics.

We now have our T type and, at this moment, it hasn't been determined so it could any type. This is the generic type and we can use it wherever we want in the class as a way of representing a generic type.

We could have as many types in the class as we wanted inside those <> brackets, so long as they are comma-separated, for example, <K, V> to denote the key/value pair of the Dictionary.

The T name is nothing special; it just happens to be common so that's why I have used it. K and V are also very common and, generally, single letters are used to indicate generic types. It stands out more and states that we are referring to generic types and not a class that has been called T – most classes have much longer names. Technically, you can call the generic type whatever you want.

Using Generic Types in Classes

We have our class that has the generic type and now we can use it wherever we want in the class. For example, because we are working on creating a list, we would most likely want to add some items to the list. We could add an instance variable:

```
private T[] items;
```

Now we have an array, marked as private, named items. Note that the array type is T, which is the generic type. Whenever the PracticeList is used, whoever uses it can choose which type they want and that will go into the list. If they wanted a PracticeList full of ints, they would do this – PracticeList<int> list = new PracticeList<int>(); and it would become an array of ints.

Let’s carry on building this class.

We will also need a constructor that sets the array to have 0 items to start with:

```
public PracticeList()

{

   items = new T[0];

}
```

This is easy enough; a new array has been created with the generic type and it has 0 items in it. Ok, it is quite worthless at this stage, but it is somewhere to start. As new items get added, we will make our array longer to accommodate them.

We also want a method that will return a specified item from our list. In this case, we want the generic type returned:

```
public T GetItem(int index)

{

   return items[index];

}
```

Again, straightforward enough; the generic type is returned. If this was used by someone else and turned into a PracticeList of ints, the return type would be ints. Once your list is created, the IntelliSense feature in Visual Studio will indicate that the method is returning the int type.

Let’s add another method; this one will add some items to the list. As far as generics go, this is as easy as everything else we have done but the code used to add items is more complex:

```
public void Add(T newItem)

{
   T[] newItems = new T[items.Length + 1];

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

   newItems[newItems.Length - 1] = newItem;

   items = newItems;
}
```

The parameter we used is our generic type of T that was defined for our class and this works as it does in all the other examples we have seen.

Adding an item to the array is not quite so straightforward. An array cannot increase in size; when you create an array, the size when you created it is the size it keeps forever. What we need to do is create another one that is larger than the first one. Then we copy everything from the original array into the new one and put the new item into the slot. Lastly, the instance variable that we called items that is tracking the original array is told to look at the new array. The example above does this.

Our complete class should now look like this:

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

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

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

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

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

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

            newItems[newItems.Length - 1] = newItem;
            items = newItems;
        }
    }
}
```

Generic Constraints

I'm not going into minute detail about this. I just want to mention that, as far as generic types go, it is possible to indicate that they have to derive from a specific base class or that a specific interface has to be implemented. Doing this requires the 'where' keyword and colon operator, like this:

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

Because we now know that our T type has to use the interface called IComparable, we make use of the methods in that interface wherever we want in our class.

If you are using several generic types, a type constraint can be specified for each one with that 'where' keyword:

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

You can specify constraints of the constructors too. This means you can specify that the generic list can be used only if the type in use has got a constructor with these specific parameters. For example, if you were to state that the used type should have a parameterless constructor, this is what you would do:

public class PracticeList<T> where T : new()

{

//...

}

Now we can see that the generic T type contains a parameterless constructor so, wherever needed in our class, we could say:

T newObject = new T();

This would refer to the parameterless constructor in the generic T type – we know it has it because we specified that it was required.

Quick Primer

- Creating classes that use generics is done by putting the generic type in a
  set of angled brackets that follow the class definitions – public class
  PracticeList<T> {/*…*/}.
- It is common to use one capital letter to indicate the generic types,
  although it is not a requirement.
- You can use several generic types in one class – public class
  PracticeDicionary<K, V> {/*…*/}.
- Once the generic type has been defined, it can be used anywhere in your
  class; either as a method return type, for declaring another type, or as a
  method parameter type – public T GetItem(int index) {/*…*/}.
- Your generic types can be given constraints, which is nothing more than a

way to indicate that any type can be used so long as it is derived from a
specific class implements a specific interface or contain specified
constructors. Them, the methods, constructors, interfaces, or base classes
can be used in your code wherever you want – you already know that the
specified type will have the specified items, even though it is generic.

What's Next?

Generics is a powerful feature in C# and can result in a huge reduction in how much code you need to write. You will find plenty of ways to use generics, not just by making use of generic classes that someone else already written, but by making your own as you need them.

That brings our intermediate section to a close and now it’s time to start looking at the more advanced subjects in C#. The first thing we will look at is reading to and writing from a file.

Part 3: Advanced Guide

File I/O

One of the most common tasks in programming is writing information or data to a file and reading it. This is known as file input and output and is often shortened to file I/O.

Writing to and reading from file is very easy to do the C# language. We are going to take a look at three ways of doing file I/O; the first is a very simple and very quick, the second is a more complex way of writing files that are text-based or readable by humans, and the third way is binary file I/O.

We are going to be making use of some files that are already in the System.IO namespace. Note that, to start with, this namespace is not included at the top of the list by default so we have to do that; this is done by adding a using directive at the top.

Put this first in your file followed by all the other using directives:

using System.IO;

The Easy Method

Writing the File

In C#, there is an incredibly easy way of reading from files and that is to use a class that is already created. It’s called File and it enables you to easily and quickly write to files.

Let’s look at two different ways we can use this method to write data or information to a file:

string informationToWrite = "Hello persistent file storage world!";

File.WriteAllText("C:/Users/RB/Desktop/test1.txt", informationToWrite); // Change the file path here to read what you want.

string[] arrayOfInformation = new string[2];

```
arrayOfInformation[0] = "This is line 1";
arrayOfInformation[1] = "This is line 2";
File.WriteAllLines("C:/Users/RB/Desktop/test2.txt",
arrayOfInformation);
```

In the first one, a single string has been prepared. This contains everything that we want to store in our file. After that we used a method called WriteAllText, which is in the File class, to write everything. That method requires that you include, not just the text you want to be written, but a file location too.

In the second one, an array of strings has been prepared. We use a method called WriteAllLines and this will output all the lines, one by one, each with a new line separating it. The file will look like this:

```
This is line 1
This is line 2
```

Reading the File

Reading the file back in is even easier:

```
string fileContents =
File.ReadAllText("C:/Users/RB/Desktop/test1.txt");


string[] fileContentsByLine =
File.ReadAllLines("C:/Users/RB/Desktop/test2.txt");
```

This does exactly the same but it does it in reverse. In the first bit, everything that is in the file is pulled into the Contents variable for the file called Contents. In the second bit, everything has been pulled into an array of strings; each of the lines in the file is a different and separate array index.

There is more to this.

While this is extremely easy to work with, it isn’t always as easy as you think it might be. On more than one occasion, you won’t just be working with single or multiple strings; you will be working with other data that you want to be written to the file too.

However, it isn’t that difficult to convert that data into one or more strings. For example, let’s assume that in your game you have a high scoreboard. On this board, you have several different scores and each one has a player name with it. We could make a file format where the name of each player is written on one line together with their score, with a comma separating the two. Like this:

```
Mary, 1000

Simon, 950

Anna, 925

Mike, 900

James, 840
```

If you think back to when we discussed how to create a class in C#, we could have an object called HighScore and it has two properties – one called Name and one called Score.

We want strings written to a file. So, to create them, we could do this:

```
HighScore[] highScores = new HighScore[10];


// populate the list of scores here as you need them and maintain the list...


string allHighScoresText = "";


foreach(HighScore score in highScores)

{

   allHighScoresText += score.Name + "," + score.Score + "\n";
}

File.WriteAllText("highscores.csv", allHighScoresText);
```

In case this is something you have not seen before when it comes to creating a string, you can use special characters inside the string. Special characters are easy to make in a string; you just start with a \ character which indicates that the next character to be typed is a special character and not a normal one. In the above example, we put \n and, although you see two characters, this is actually just one. It is a newline character which makes the line wrap to the next one. It’s the same as what happens when you press the Enter key on your keyboard.

The reverse of this is reading the file back in and you can do a lot of similar things with it.

You can start by reading the whole text of that file back in. Next, you can parse the file; that means you break up a text block into small parts that are more meaningful. After that, the file is turned back to a list containing the high scores.

The next example will read the high scores file we created and convert it back to a list of high scores:

```
string[] highScoresText = File.ReadAllLines("highscores.csv");


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


for(int index = 0; index < highScoresText.Length; index++)

{

   string[] tokens = highScoresText[index].Split(',');


   string name = tokens[0];
```

```
    int score = Convert.ToInt32(tokens[1]);


    highScores[index] = new HighScore(name, score);
}
```

As you can probably see, the method called Split is going to prove very useful when we read in from a file. This method breaks a string into several smaller ones splitting when it comes across a specified character. In our case, that character is the comma.

Reading Text-Based Files

There is another way of handling I/O for text-based files but it is a lot more complicated. What it lets us do is write and read files a bit at a time instead of all at the same time, which is what happens in the first example. What we will be looking at here is how to use this method to write and read text-based files.

Writing the File

To start, we can make use of the File class. This time though, rather than writing the whole file out at once, all we will do is open the file so it can be read. What this will give us is a FileStream object; we wrap this in a TextWriter object and then use to it to write things when we need to.

```
FileStream fileStream =
File.OpenWrite("C:/Users/RB/Desktop/test3.txt");
TextWriter textWriter = new StreamWriter(fileStream);


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


textWriter.Flush();
textWriter.Close();
```

What we have done is use a method called OpenWrite and that gives us a FileStream object back. It is possible to work directly with this FileStream object but it is a very low-level object. What we want is something that is more advanced, something that can make things much easier for us. What we do is wrap the FileStream object in another object, the TextWriter object. What we actually use is a StreamWriter, which is a TextWriter and that uses inheritance.

Now, using this TextWriter, we can write all kinds of ints, text, whatever we want, making use of the Write method overloads.

When the writing is done, the next step is to clean everything up. We can use a method called Flush to ensure that everything needed has been written to the file. TextWriter may not do this until there is enough text to write a lot of it at once, rather than a bit at a time. Lastly, the Close method is called to release the connections that we might still have to the file.

Reading the File

Reading the file in is more of a problem than writing it. Why? Because when we write information out to a file, we don’t have any real way of knowing the way it was structured, which makes it difficult to read it back in. One way we could do it is to go back to the File.ReadAllLines method. You can mix it up as you want but you should still look at reading in a file in a way that matches the way we wrote them out.

You do need to bear in mind that things are going to be a bit more complicated because we have to do this at a level that is lower than the level we used to write the file out.

```
FileStream fileStream =
File.OpenRead("C:/Users/RB/Desktop/test3.txt");

TextReader textReader = new StreamReader(fileStream);


char nextCharacter = (char)textReader.Read();
```

```
char[] bufferToPutStuffIn = new char[2];
textReader.Read(bufferToPutStuffIn, 0, 2);
string whatWasReadIn = new string(bufferToPutStuffIn);

string restOfLine = textReader.ReadLine();

textReader.Close();
```

As with the file writing, the first thing we do is open the file. This time, we use a method called OpenRead. Next, the FileStream object is wrapped in a TextReader and that puts us in a position to start reading.

If you want a single character read in, you can use the method called Read(). An int will be returned from this; the next step is to cast that into a char.

You could also read in a lot of text and doing this would require you to use another overload from the Read method. As you see from the above example, using this method requires an array of chars to be created and then passed into the Read method. You could also specify the index where you want the writing to start (we used 0) and how many characters need to be written in (we opted for 2). The example shows how the array of chars can be converted to a string.

In the TextReader, there is also a method called ReadLine. This lets you read text into the end of the line and that will return a string automatically. We haven't shown it in the example but another method called ReadToEnd will pull the rest of the file in starting from the point you are currently at.

As with writing, when we are finished with the reading, we want to call TextReader.Close() to ensure that there are no more connections to the file.

Reading and Writing Binary Files

Rather than writing text-based files, there is an alternative – we could write binary files. When we write these, you cannot open the files in a text editor. Well, you can actually, but it would look like complete and utter gibberish.

However, there are a couple of great advantages to binary files. First, a binary file will take up a good deal less space than a text-based file. Second, to a certain extent, the data in binary files are encrypted. Because it isn't text, it can't just be opened up and read and that provides a certain level of protection for the data. However, this isn't true encryption, not actual proper encryption; all it does is obscures the data. Any programmer worth their salt could still work out what is in it.

Writing the File

This is much like the text-based method we talked about earlier. Writing to a binary file requires the same code with one exception – instead of a TextWriter, we use a BinaryWriter.

```
FileStream fileStream =
File.OpenWrite("C:/Users/RB/Desktop/test4.txt");

BinaryWriter binaryWriter = new BinaryWriter(fileStream);


binaryWriter.Write(3);

binaryWriter.Write("Hello");


binaryWriter.Flush();

binaryWriter.Close();
```

As before, the first thing we need to do is open a FileStream that has a connection to the file we are writing using the OpenWrite method. This time, we are not wrapping it in a TextWriter. We are wrapping it in a BinaryWriter.

Next is the Write methods. We can call however many we need for outputting our file. We call the Flush method – this forces our BinaryWriter to write absolutely everything that it has and then the Close method to cut the connections to the file.

Reading the File

This time, reading binary files is much easier than it is to read text-based files.

```
fileStream = File.OpenRead("C:/Users/RB/Desktop/test4.txt");
BinaryReader binaryReader = new BinaryReader(fileStream);


int number = binaryReader.ReadInt32();
string text = binaryReader.ReadString();


binaryReader.Close();
```

If you noticed, this is quite a lot like what we looked at before. The file is opened and the FileStream object is wrapped in a BinaryReader. However, now we can just call the numerous versions of the Read***() in BinaryReader. Those include ReadInt32 for the ints, ReadInt16 for the short type, ReadString for the string type, and so on.

Then, as usual, the Close method is called on BinaryReader so our file connection is closed.

The File Class

While we are discussing the File class, it is worth mentioning a few of the things that it can do. There is a Delete method in the class; this lets you delete files, although you should be aware that this will not show you a prompt asking you if you are sure you want to carry on. There are also some methods that allow you to copy and move files; they are called Copy and Move as you would expect. Lastly, you also have the ability to see if a specified file is in existence, called the Exist method. If you attempted to read from a file that isn't there, you can expect to see errors.

Quick Primer

- Reading and writing to a file is sometimes known as file input and output but is more often called file I/O.
- File I/O is not difficult to do in C#, but not as complicated as other languages.
- Everything you do with file I/O will always begin with the class called File.
- If you want to write multiple lines to a file at the same time, use File.WriteAllLines(string filePath, string[] lines);.
- If you want to write the entire contents of a file out at the same time, you would use File.WriteAllText(string filePath, string allText);.
- When you do file I/O, there is a good chance that your data will need to be turned into a format that fits with the output methods; the text would then need to be parsed back in allowing your objects to be reconstructed. In simpler terms, there is much more to file writing that just putting the data into the file – it must also be in the correct format to work properly.
- Text-based and binary files are read and written in different ways. With the binary files, we can read and write a little at a time whereas, with the text-based files, it is all at once. Whichever way you do this, you always begin with File.OpenRead or with File.OpenWrite. The FileStream that is returned is wrapped in either a TextReader or TextWriter or a BinaryReader or BinaryWriter, depending on what you are doing. Then you can do all the reading and writing you want; when you have finished with the writing, always call the Flush method for the writer and call the Close method for the reader or the writer.

What's Next?

File I/O is incredibly powerful and it is one of the most important parts of programming that you can learn to do. Almost every program you come across will require this in one form or another.

Obviously, there are far more advanced things that file I/O can be used for, such as reading or writing an XML file, and in C# you will find all the tools you need to write an XML file much easier.

There is still a lot to this subject that we have not yet covered but we have touched on the most important parts of it. What you learned here will be sufficient to get you started and the best way to learn more is to get going and do it.

As you continue working through this guide, you will learn even more of the more complicated and advanced C# programming topics and the next logical place for us to move to is to look at error handling and how to use exceptions.

Error Handling: Exceptions

On occasion, things will go wrong. It happens even with the most seasoned of programmers. A user might have clicked a wrong button; they may have clicked a sequence of buttons in the wrong order. Perhaps a freak hurricane wind took out your network cable.

The point I am trying to make is that anything can go wrong and that's what we are going to talk about here – how to handle these errors properly in a way that keeps the negative effect on the user to an absolute minimum while the problem is fixed in the section of code that caused the error.

Let's start with an example. A user types in the wrong input, perhaps a piece of text or a letter when it was meant to be a number. The code we start with could look like this:

Console.Write("Type in a number: ");

string userInput = Console.ReadLine();

int number = Convert.ToInt32(userInput);

If you are not aware of what might happen if you were to input abc and not a number, try it and see what happens.

If your program is being run in debug mode, it will just freeze up, and Visual Express will indicate to you where the problem was – in this example, it will tell you that the error was on the line that reads Convert.ToInt32. If your program is being run in normal mode, that is the non-debug mode, all that will happen is your program will die.

We have an error here and it needs to be fixed.

Thankfully, there is a built-in way on C# that lets us do this – it's called exceptions.

Exception handling is a very powerful way of dealing with errors. Let's say that one of your methods is attempting to do something and it has gone badly wrong. Right now, it's all a bit of a mess and the method that found the problem might not know what bit caused the issue or how it should be fixed. All it knows is, there is a problem, and it just doesn't know how to move on.

Exceptions give methods a way to say that they ran into a problem and they cannot continue in an intelligent manner. The error, together with any available information about the problem, is sent back to where the code or method was originally called from, with the hope that someone else, somewhere, can come up with a way to fix it.

If something else can fix the problem, the message will be intercepted and that something will do something to solve it. It could be as simple as letting the user know that there is an issue and they need to try again.

If there isn't anything that can handle the error, it will continue to go to higher method calls, right back to the Main method. If the Main method can't handle it, the program will end and the user will be left wondering how to get their lost file back.

The idea is that, regardless of what the error is, something should be able to handle it so the program can continue in some way regardless of the problem because the exception was handled.

"Catching" Exceptions

You could think of an exception as being something like the Hot Potato game. Never played it? It has a very simple concept. A group of people standing a circle and they throw an object, the hot potato (it may not actually be a potato) from one person to the next as quickly as they can. There will be some music playing and, when that stops, the person who is left holding the object is out of the game. The idea is to throw it to the next person and get rid of it as quickly as you can.

When your computer program has a problem and something goes badly wrong, if the computer cannot fix it, you will get an error message created, all neatly packed into an object that has the Exception type. That message is then passed up the chain very quickly back to the method responsible for calling it. That method will not finish executing and it will not return a value; all it will do is stop suddenly, have the exception created, and return to where it was called from.

When it gets back there and if the method isn't able to handle the problem, it will quickly pass the exception to the method that called the method. This will continue until it gets to a point where something can handle it or it gets to the Main method; at which point, the Main method will throw the exception once more and your program will die. If your program is in debug mode, Visual Studio is opened at the point where the error happens.

What we need to do is work out who should handle it and then we need to work out how to 'catch' the exception and do something with it. In case you were wondering, all this talk of throwing and catching is not made up – they are the real terms used by programmers for this scenario.

How do we do this? What we need is a try-catch block. And that gives us two more new keywords to play with – try and catch. In basic terms, we are going to say that there is a bit of code that could end in a problem; try running it and, if an error or an exception does happen, we can catch it and use another code block to handle it.

Make sense?

We do it like this:

```
Console.Write("Enter a number:  ");

string userInput = Console.ReadLine();


try

{

   // The code that might cause a problem will go here...

   int number = Convert.ToInt32(userInput);
```

```
    // It is important to understand that if an error occurred in
    // the above line, the program would go down to the
    // catch block you see below, and the code here wouldn’t be
executed.
}
catch(Exception e)
{
    // The code that will handle the error or exception will go here.
    // Notice that if there isn’t an error, that piece of code
    // will never be executed.  gets executed.
    Console.WriteLine("You must input a number.");
}
```

Basically, your dodgy piece of code is put inside a try block; immediately following that is a catch block that contains the code needed to handle the exception. If anything goes wrong inside your dodgy code, the rest of it will cease to execute. Instead, it goes straight to the catch block where the error handling code resides. However, if it all goes as it should, that code in the catch block is never executed.

Look at everything we put inside the parentheses at the beginning of our catch block. It says Exception e. We need to talk about two things – first, we have created a variable in this line and named it e (we could have called it anything). That variable is then used inside the catch block. For example, the class called Exception has a property named Message so you could do something like Console.WriteLine(e.Message) and then print the message. The Exception object that we have here could be holding information about the problem and you can look to see if you can work out what went wrong.

```
}
```

The second thing is that when exceptions get thrown, they could be using that Exception class or they could be using a derived class that is specific to whatever problem type occurred. If we return to our example of bad user input, this hasn't thrown an Exception object; instead, it threw a different object, called FormatException and this is an object derived from our Exception class. I'd better clear things up a bit here. In technical terms, an exception has been thrown but the FormatException is what gets thrown and this is an Exception type.

In the code example we used, the method called ConvertToInt32 is able to throw two different exceptions – the FormatException or OverflowException. The latter will happen when you provide the method with text that is a number (anything else would cause the FormatException to be thrown) but it is too big a number for an int. A number like 12345678912345678912 34 could do that.

Inside the catch block, we could indicate that it should catch(Exception e) and catch whatever is that exception or that is an Exception class derivation (pretty much everything). Or, we could have catch(FormatException formatException) and that would only catch what is of the FormatException type. If an OverflowException is thrown, it gets sent on as unhandled to the method that called it and then to the next and so on until another try-catch block handles it or it gets to the Main method where the program will die.

It is up to you to determine what exceptions will be handled and how.

Not Naming the Exception Variable

Earlier, I said where the catch block is concerned, the exception that you are trying to catch can be used as a variable in the catch block. However, if you don't intend to use it, there is no need to provide the variable with any name:

```
try
{
   //...
}
```

```
catch(Exception) // this Exception variable does not have a name
{
   //...
}
```

On the one hand, you have no way of referring back to it from the catch block but, on the other, that isn't going to be a problem if you never intend to use it. What this does is frees up the name that it would have been called, so it can be used on another variable. It also ensures that the compiler doesn't kick up a warning telling you that the named variable was declared but was never used.

Different Ways to Handle Different Exceptions

It is possible for different code blocks to throw up different exceptions and any of the exception types can be caught or all of them can be caught. Alternatively, you can pick a derived type to catch.

What if you wanted to be awkward and catch every exception but deal with each different error type in a different way?

Well, that's not awkward at all because there is a very easy way of doing it.

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

All you do is add in as many catch blocks as you need after the try block.

However, only one of the blocks will be executed; as soon as that exception is handled, it goes right over the top of all the other catch blocks. If you get a FormatException thrown, it goes into the first of the blocks and runs the code that will handle the problem; what then happens is that the catch(Exception e) block will not be executed despite the fact that, technically, the exception was an Exception type – it's been handled already.

When you do this, you can handle every exception type. All you need to do is make sure that the more specific of the types, which is the derived type, is placed before the other general base types. If the first catch block is catch(Exception e), everything would be caught and nothing else would get to the Overflow Exception block or the FormatException block.

Order is incredibly important and it is a detail you must pay attention to.

Throwing Exceptions

Up to now, we have only discussed handling these exceptions. What if you were busy writing code and you became aware that there was a potential problem that might happen to stop your code dead? It's quite simple – you just create your own exceptions and throw them.

To do that, all you need is the 'throw' keyword:

So far, we've just talked about handling exceptions. However, what if you are writing some of your own code and you are aware of a potential problem that could occur that could stop you dead in your tracks? You can create and throw your own exceptions!

```
public void CauseAnIssueForTheWholeWorld()  //never up to any good...

{

   throw new Exception("Just my job!");

}
```

What do you think? Does it feel a bit like a return statement? It is, but rather than returning values, this one throws exceptions. The Exception object can be created in the same way as any other object by using the new keyword.

You don’t have to stay using just the Exception class; there are lots more options that are already defined. Look at some of the more common exceptions and what they are all for:

Exception Type	**What Does It Do?**
lementedException	This exception can be used if you have defined but not implemented a method. If methods are automatically generated by Visual Studio, this is what goes in the body.
utOfRangeException	This exception is thrown if you attempted to access an array or something else at an index that goes beyond the size of the array.
CastException	You attempted to cast one thing to another type but you tried to cast to the wrong type of object.

Exception	You have text that is in the wrong format to be converted to something else, such as letters contained in a string that you want to convert to a number.
portedException	You attempted to perform an operation that was not supported, for example, trying to call a method at a point where it was not allowed.
erenceException	A variable for an object type (any type) has null in it rather than an object but the operation you wanted to perform didn’t require a null; it wanted something else.
verflowException	If you run out of space (memory) on the stack because you called one or two methods too many, this exception will be thrown – it tends to be recursion that has gone very wrong.
yZeroException	You attempted division by zero and got caught.
ntException	An argument or a parameter that you sent to a method wasn’t a match to what the method required.
ntNullException	A parameter or an argument that you passed into a method was a null but the method needs something else.
ntOutOfRangeException	An argument had a value that the method couldn’t do anything within an intelligent way. For example, a method needed a number between 1 and 9 and you provided a number

	of -10 – this would throw this type of exception.

If you don’t happen to see the exception type that you want, you simply create one of your own. All it requires is creating a derived class from the Exception class or you could create a class that will derive another class that has been derived from the Exception class. For example, you could derive a class from the class called ArgumentNullException if you wanted to.

This would look like this:

```
// For those of you over the pond, if you really want to and it makes you feel better,
// they can still be called sausages if you want!
public class AteTooManyHotdogsException : Exception
{
   public int HotdogsEaten { get; set; }

   public AteTooManyHotdogsException(int hotdogsEaten)
   {
      HotdogsEaten = hotdogsEaten;
   }
}
```

Using this class, you are saying:

```
throw new AteTooManyHotdogsException(125);
```

And:

```
try
{
   EatSomeHotdogs(32);
}
```

```
catch(AteTooManyHotdogsException hotdogsException)
{
   Console.WriteLine(hotdogsException.HotdogsEaten + " is far too
many hotdogs.");
}
```

The *finally* Keyword

Before we move on, I want to talk about the finally keyword in relation to exceptions. At the end of any try-catch block, you could add a finally block and this would turn it into a try-catch-finally block. Inside the finally bit, your code would be executed regardless of anything else. If the code went through the try block in the normal way, it would be executed. If an exception was thrown, it would be executed.

Let’s see how this works:

```
try
{
   // Do something that could potentially throw an exception here
}
catch (Exception)
{
   // The exception gets handled here
}
finally
{
  // This code is always going to be executed, no matter what goes
on in the try-catch block.
}
```

Now, you are possibly wondering, why can't I just place the code from the finally block into the program all on its own after the try-catch block? Why couldn't you do something along those lines?

```
try

{

   // Do something that could potentially throw an exception here

}

catch (Exception)

{

   // The exception gets handled here

}


// This code is always going to be executed, no matter what goes on
in the try-catch block.
```

You are quite correct – this will always be executed regardless of what happens inside the try-catch block. However, there is a difference here, albeit a subtle one. What is in the final block does get executed whether the return keyword is used anywhere in the block or not.

Have a look at this code to see what I'm talking about:

```
public static void TestFinally(bool throwException)

{

   try

   {

      Console.WriteLine("in try");


      if (throwException)

      {
            throw new Exception();
        }


        return;
    }
    catch (Exception)
   {
        Console.WriteLine("in catch");
    }
    finally
    {
        Console.WriteLine("in finally");
    }
}
```

Try running the code and see what happens. You should find that, provided the flow of execution is as it should be and you don't come up against any errors, you will get to the return statement inside the try block. However, as it leaves the block before anything is returned, what is inside the finally block is executed.

What this means is that return statements cannot be included in finally blocks. They just don't work that way – by the time they are executed, they could be returning already.

This brings up something else that is interesting. Somehow, the finally block got itself affixed to exception handling and so many coding books simply treat it as a part of the exception handling purely as a way of making sure that code is executed whether an error occurs or not.

That isn't really what it's about, not exactly, anyway. If that is what you wanted to achieve, all you would need to do is place the code after the try-catch block has finished completely.

It is about running specific code, regardless of how a method returns. You might have a normal return, you might have an error return, or you might even find dozens of ways to return normally but by forcing the execution of a specified code. For example, you could do something daft like this:

```
private int numberOfTimesMethodHasBeenCalled = 0;


public static int RandomMethod(int input)
{
   try
   {
      if(input == 0) return 17;
      if(input == 1) return -2;
      if(input == 2) return -11;
      if(input == 3) return 8;
      if(input == 4) return 6;

      return 5;
   }
   finally
   {
      numberOfTimesMethodHasBeenCalled++;
   }
}
```

Whatever happens, whatever the method return is, the code inside the finally block is always going to be executed. Now we could write this in another way and not use the try-finally block; for example, you could just put numberOfTimesMethodHasBeenCalled++ right at the beginning of the method and that would do the job.

There are many different ways of writing the same code.

Quick Primer

- When something goes wrong in your program, an exception will give you a way of having the bit of your code that found the problem, tell you that something is wrong, and then send the execution back to another bit of the code that can deal with the reported problem.
- If you are running a piece of code that you think might cause a problem, you should wrap the code inside a try-catch block ensuring that the exception type is placed inside the catch block parentheses. For example, try {/* code that could throw errors goes here */} catch (Exception e) {/* the code that will handle the problem will go here */}.
- You are not limited to using the Exception class; you can make use of any type that was derived from that class too, such as the FormatException class. If you do this, you will only handle exceptions of a specified type while others get thrown higher up the call stack.
- Several catch blocks can be strung together, starting with the most specific exception type and going to the least specific type. For example, try {} catch(FormatException e) { /* format errors are handled here */ } catch(Exception e) { /* all other errors get handled here */ }.
- You can derive from the Exception class to create your own exception types
- To get the process started in your code, the throw keyword is used. For example, throw new Exception (“An error occurred”).

What's Next?

Using exceptions to handle errors might not seem particularly easy to use at first. There is an awful lot of try-catch bits floating about and, often, things look and feel more complex than they actually need to be. You don’t need to worry; you can learn this a little at a time.

Try it out for yourself. When you think a bit of code you are writing might be a problem, use the try-catch blocks. Do the same if your program crashes and throws an exception.

With error handling, you take care of many problems that arise and, probably, more importantly, it will let those methods that find the problems pass it on to where the problem occurred or to somewhere that can deal with it.

Next, we look at how methods can be treated like they are objects so that we can pass them about as we need to. That way is called delegates and they are a very important part of the next section – events. These let one bit of code tell another bit of code when something happens. First, we’ll discuss the delegates.

Delegates

Before we start on delegates in C#, it would help to understand what a delegate is in the real world. You probably already know this but a delegate is someone (in some cases, something) that does something or takes on responsibility for something on behalf of someone else (or something).

You'll have heard the terms, 'delegate' and 'delegation', which is a just a group of delegates usually in connection with politics and government. Most of today's governments run on a system of votes where a person or group of people is chosen to do some work. These people are the politicians, the representatives, or the delegates of the people who put their cross in the box and, in theory, it is down to them to take on responsibility for something that they were chosen to do, in whatever way is best.

Delegates in C#

In C#, we don't have people, we have our methods. These are nothing more than code blocks that do certain things. They do that task in their own way and, to all intents and purposes, we really don't care how it's done, so long as it is done satisfactorily. Not all methods will do the exact same kind of work. Some will print something for a user to see, others do the math, and then there are those that just cause the program to crash.

One concept in C# is delegates and, I will be the first to admit that it may not be all that easy to grasp – but that is why it is in the advanced section, after all. As basic as I can put it, a C# delegate is saying that it has some work that has to be done and it wants to palm it off to someone or something that can do it. A delegate will indicate what information they are handing off, i.e. the work they need to be done, along with an indication of what they want back.

They do this by stating that, if a method wants to take on the work, it must have specific parameters and it must return a specific type. However, unlike the interface, it really doesn't matter what the method name is; we don't care about that, only about the parameters and the return type.

Declaring a Delegate

Now you know what a delegate is and what it does. It's time to look at how you create one and this will help you to understand them a bit better.

The delegate we are going to create is one that will represent a method (any method) that can use two numbers in some kind of math operation. What we are going to be saying is: we have these numbers and we need somebody, anyone, doesn't matter who, to do some math with them and then give us back the result.

There may well be a method floating around that can add the numbers and return the result, or that multiplies them. There might be a method that takes those numbers, squares them, adds them together, and then returns the square root. Hold on, isn't the Pythagorean Theorem? The point I am making is that all we want is any method that can do something with our numbers and get a result.

We know what we want so let's make it happen.

We create delegates in the namespace in the same way that we do a struct, a class, and an interface. Just like those, you should get into the habit of placing each delegate into a file of its own – keep them separate for the sake of ease and readability.

So, create a new .cs file and call it whatever your delegate will be called. I'm calling mine MathDelegate.cs but you can call yours whatever you want. Now add the code below to create your delegate:

public delegate float MathDelegate(float a, float b);

Your MathDelegate.cs file (or whatever you called it) should look like this:

using System;

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

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

What we have now is the delegate keyword that says we are searching for a method, any method that can take a pair of floats and return a float.

Using Delegates

To use delegates, we need methods that are going to match what our delegate needs to be done. What we need are methods that can take floats as parameters (in our case, two floats) and then return a float as the result. Go back to where your Main method is in your Program class and create the following methods – all three of them will match what our delegate requires:

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

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

```
public static float Power(float baseNumber, float exponent)
{
   return (float)Math.Pow(baseNumber, exponent);
}
```

All three methods take a pair of floats as parameters and all three will return a float but each method works with the numbers in a different way.

In a more specific type of situation, you would most likely already have defined these methods a long time before you even thought about creating delegates for them, but in our case, we haven’t got that far yet.

Using delegates is much like using objects. In your Main method, you can add some code that will let you use any of these methods by using a delegate. You don’t have to put this in your Main method; you can put it wherever you want it. For the purposes of this, add the following into your Main method:

```
MathDelegate mathOperation = Add;


float a = 5;
float b = 7;


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

What we’ve done here is created a ‘variable’ that has the type of MathDelegate and then we named it, just as we do with any normal variable. A value is assigned, which is the name of one of our methods that does what the delegates needs – two parameters which are floats and float return type.

In the next couple of lines, we set up a and b and, by now, that should all make sense to you – all we have done is some standard setting up.

On the final line, our delegate is used and this looks a lot like a method call. In many ways, that is what it is; we are just going via the delegate.

Because we assigned mathOperation to the Add method, that Add method will be called and the result would be 12.

We don’t have to create the delegate and give it a value right away. We could do some logic to work out which possibility out of all of them we wanted to use:

```
bool subtractionIsWayBetterThanAddition = true;

MathDelegate mathOperation;

if(subtractionIsWayBetterThanAddition)
{
   mathOperation = Subtract;
}
else
{
   mathOperation = Add;
}

float a = 5;
float b = 7;

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

Depending on what the state of the Boolean variable, isSubtractionWayBetterThanAddition, is, we could end up by calling our subtract method.

Methods as First-Class Citizens

You might have realized that using a delegate is very much like using a variable or an object and this is done by design.

A delegate lets you assign a method as you do with a variable; it lets you pass a method as a parameter to another method and it lets you return a method from another method.

In computer programming, this has a name. A delegate will make it so that a method is a “first-class citizen”, much like the variables with a value type – the int, float, byte, etc. – and the structs and objects.

This is a very powerful ability but only if you know exactly how to use it. Later, in the next part of this guide, we’ll be discussing events and these are also very useful, especially to those who want to get involved in GUI programming and create much better interfaces than the console.

Aside from that, delegates are incredibly powerful all by themselves and that makes them worth knowing about.

A Better Example

Just to show you, once and for all, the power and usefulness that delegates have, I’m going to show you a more complex example; this is very powerful and it is likely to confuse you.

But, I will tell you this. If you have kept up with me so far and you have a basic understanding of everything to this point if you feel that you are truly ready to move on and look at events, then understanding this part really isn’t that critical. If you want, you can move on to events right now but I would suggest that you finish this section, if only to round off your knowledge.

Let’s start by looking at exactly what we aim to achieve with this example. Let’s assume that you have an object, anything, it doesn’t matter what it is, and you are using this object. You also have a List, full of other objects that you are working with. You also have lots of operations that you want to be done on each of the list items at different times.

For the sake of simplicity, the object we are going to use has been defined like this:

```
public class MagicNumber
{
    public int Number { get; set; }


    public bool IsMagic { get; set; }
}
```

This is a very simple class but I am attempting to keep this as simple as I can.

We also have a few methods that will do things with MagicNumbers, like these:

```
public void Increment(MagicNumber magicNumber)
{
    magicNumber.Number++;
}


public void Decrement(MagicNumber magicNumber)
{
    magicNumber.Number--;
}


public void Magicify(MagicNumber magicNumber)
{
    magicNumber.IsMagic = true;
}


public void DeMagicify(MagicNumber magicNumber)
{
    magicNumber.IsMagic = false;
}
```

I’m still trying hard to keep this simple!

Any of those methods could do many different things with MagicNumber.

But, all of these methods will take an input of MagicNumber but none of them will return anything and this is where a delegate is very useful.

Our delegate could be created in this way:

```
public delegate void MagicNumberModifier(MagicNumber
magicNumber);
```

Remember; somewhere in the main program is a kind of container that will hold all the MagicNumbers we have. It could be any container but we’ll stay with a List for now.

```
List<MagicNumber> magicNumbers = new List<MagicNumber>();


magicNumbers.Add(new MagicNumber() { Number = 4, IsMagic =
true });
magicNumbers.Add(new MagicNumber() { Number = 6, IsMagic =
false });
magicNumbers.Add(new MagicNumber() { Number = 9, IsMagic =
true });
```

```
magicNumbers.Add(new MagicNumber() { Number = 13, IsMagic = false });

magicNumbers.Add(new MagicNumber() { Number = 20, IsMagic = true });
```

We could do this:

```
public static void ApplyModifierToAll(MagicNumberModifier modifier, List<MagicNumber> magicNumbers)

{

   foreach(MagicNumber number in magicNumbers)

   {

      modifier(number);

   }

}
```

And, inside the Main method, or wherever you want it, this:

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

{

   List<MagicNumber> magicNumbers = new List<MagicNumber>();


   magicNumbers.Add(new MagicNumber() { Number = 4, IsMagic = true });

   magicNumbers.Add(new MagicNumber() { Number = 6, IsMagic = false });

   magicNumbers.Add(new MagicNumber() { Number = 9, IsMagic = true });

   magicNumbers.Add(new MagicNumber() { Number = 13, IsMagic = false });

   magicNumbers.Add(new MagicNumber() { Number = 20, IsMagic = true });
```

```
    ApplyModifierToAll(Increment, magicNumbers);
    ApplyModifierToAll(Magicify, magicNumbers);
}
```

This is an excellent example of something called a Visitor Design Pattern. You won’t know what these are at this stage but, in short, a design pattern is a generalized solution designed for problems that are similar. These are not used by all programmers, but with the Visitor pattern, you can do all sorts of operations on an object type without needing to worry about the structure they were contained in.

Quick Primer

- Real-world delegates are people or things that do something for people or things.
- C# delegates provide a way for methods to be passed like variables or objects giving you the ability to delegate methods as you go, as and when needed.
- The delegate keyword is needed to create them, i.e., public delegate float MathDelegate(float a, float b);. We could now use this to track methods that have the float return type and the same float, float list of parameters.
- Once created, delegates can be used in a similar way to a variable, i.e. MathDelegate mathOperation = Add; float a = 5; float b = 7; float result = mathOperation(a, b);. Of course, the implication here is that you have an Add method that already returns a float and takes a pair of floats as its parameters, thus matching what the delegate requires.

What's Next?

Delegates are more than a little confusing but, like everything, you don’t need to have a thorough understanding of them, to begin with. At the end of the day, understanding the basics is enough to get you started and practice will improve your ability and understanding.

In order to move on to the next section, you must have that basic understanding. If you haven’t, go back over this section until you feel ready to take on the next topic – events. These rely on delegates very heavily so make sure you know enough.

Events

Events are one of the coolest features in C# because when a specific event occurs, they let classes notify others. This is one of the most common things in applications that are GUI based; you might have things like forms, checkboxes, and buttons. All of these can indicate that something has happened – a button was pressed, a form was completed, a box was checked, and so on – notifying other classes of what has happened so they can handle things as they need to.

Delegates are a key part of the events features – if you still don't understand delegates, go back over the previous section because you need, at the very least, a basic understanding of them. Delegates are what we will use to receive the event notifications.

Defining an Event

The first thing we need to do is create our event. We'll assume that we have a class representing one point that is in two dimensions; thus it will have both an X and a Y coordinate or value. The class would look like this:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace Events

{

   public class Point

   {

      private double x;

      private double y;
```

```
        public double X
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        }

        public double Y
        {
            get
            {
                return y;
            }
            set
            {
                y = value;
            }
        }
    }
}
```

Defining an event in the class requires just one code line and this will be added as a class member:

```
public event EventHandler PointChanged;
```

Your code should look like this:

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

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

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

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

        public event EventHandler PointChanged;
    }
}
```

Like any other member – a property, a method, or an instance variable – the three access modifiers (public, private, and protected) can be used for the events; public is the most commonly used.

We also use a keyword called event, which tells the compiler that it is an event we are defining.

Next, the delegate type needs to be specified so that the other places can use it to handle the event. Remember how delegates work? What we are saying is that this is a specific return type and parameters that a method will need in order to be notified of an event. In our case, we are using a delegate that was already created for us by Microsoft and it is nothing more than a basic one that has been called EventHandler. It has a void return type and it has two parameters – the first is an object, which ‘sends’ the event, and the second is an EventArgs object, which is really very simple.

We could create a delegate ourselves rather than EventHandler, but for now, this one will do nicely. If you needed something that was more specific then you would need to make one of your own.

Following this pattern is common practice if you want to create your own – a different ‘Args’ class would be derived from the class called EventArgs (you could call it something like ButtonEventArgs) and then the information that you want is added in – a reference to the pressed button, for example. After that, you would have a delegate that matches, taking a pair of parameters, the sender object, and the ‘Args” class that you created. You don’t have to do it this way but it is the most common way and you will see it a lot.

Raising an Event

Ok, we have our event so now we need to raise it when the perfect conditions arise. That requires some code. I don’t want you to add it in just yet; first, I want you to look at it. This is how an event is raised:

```
// This code will raise the event called PointChanged, but it is

// better if you do this within its own method, as you will see shortly.

if(PointChanged != null)

{

   PointChanged(this, EventArgs.Empty);

}
```

This really is quite a simple piece of code.

All we are doing is looking to see if our event is null. If it is, it indicates the event has no event handlers attached to it – we’ll see how to do those soon. When you raise an event that doesn’t have any event handlers, the result is a NullReferenceException, so this needs to be checked before the event is raised.

As soon as we know that there are event handlers attached to the event, we can raise it and, to do that, we call the event with the required parameters for the delegate. In our case, we use a reference to the sender and we use an EventArgs object (in this code, we have used the static object called EventArgs.Empty.

As I said before, you should get into the habit of placing this code inside a method of its own and one of the most common things is to give the methods the same name as the event but with a prefix of On, for example, OnPointChanged. However, you don't have to put the code inside a method and you do not have to name it as On… but that is what most programmers do.

So, add this method into your code:

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

What will happen now is whenever the event conditions are met, the event is raised by calling this method.

In this example, we want to have an event for whenever there is a change in the point so the method will be called when the X or Y values are set.

To do this, we just need to put a method call in the setters for the X and Y properties:

```
public double X
{
   get
```

```
    {
        return x;
    }
    set
    {
        x = value;
        OnPointChanged();  // The method call is added here.
    }
}

public double Y
{
    get
    {
        return y;
    }
    set
    {
        y = value;
        OnPointChanged();  // The method call is added here.
    }
}
```

The complete code with the event that has been declared, the method that raises the event, and the code that triggers the event whenever the correct conditions have been met should look like this:

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

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

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

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

        public event EventHandler PointChanged;

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

Attaching and Detaching Events

Our Point class has been set up now using an event for whenever a change is made. The next step is learning how to attach event handlers. In our cases, we want to attach a method to the event handler.

Remember, the C# events are based on the C# delegates so the method we use must match the requirements of our delegate – EventHandler. This delegate requires a return type of void and it requires two parameters (a sender object, which is what triggered the event, and the EventArgs object, which contains information about the event that we are raising – this won’t have a lot of information in it for this particular situation, but in other events, that could change).

Somewhere in your code (continue to use the Main method for now), create your method that will handle that event. Something like this:

```
public void HandlePointChanged(object sender, EventArgs eventArgs)
{
   // When the pointer changes, something intelligent should be done; maybe draw the GUI again,
   // or update a different data structure or anything you can think of, really.
}
```

Attaching event handlers to an event is really quite easy to do and, again, the code below can go wherever it needs to go:

```
Point point = new Point();

// This is the most important bit:
point.PointChanged += HandlePointChanged;

// Now, if the point is changed, the event called PointChanged is raised,
// and HandlePointChanged is called.
point.X = 3;
```

The important line in this is the line that attaches the event handle – point.PointChanged += HandlePointChanged;. We use the += operator for adding the method to the event; if you attempted to attach methods that do not meet the requirements of the delegate used by the event the result will be a compile-time error.

It is also worth noting that you can add as many event handlers as you want to any event.

We can also detach an event handler too, and this is an important thing to do when you don't need to be notified of the event anymore. Depending on how your code is organized, if you do not detach events, you could find yourself with one event handler attached to an event several times – best case, this will affect performance, and worst case, your program will break.

Detaching an event requires to use the -= operator in a very similar way to using the+= operator to attach an event:

point.PointChanged -= HandlePointChanged;

Once the code has been executed, HandlePointChanged is no longer going to be told when the PointChanged event happens.

Quick Primer

- Events are a very easy way for a piece of code to tell other pieces of code that a specific condition is met.
- Events require delegates to do the work so you need to have a basic understanding of the way delegates work.
- Creating events in classes requires code much like this: public event EventHandler PointChanged;. EventHandler is the delegate that is used by the event handlers and PointChanged is an event name.
- Events are raised by checking whether event handlers have been attached and then raising it like this: if(PointChanged != null) {PointChanged(this, EventArgs.Empty); }.
- When you attach a method it has to meet the needs of the delegate in use and it is done by using the += operator: PointChanged += PointChangedHandler;.

- When you detach a method, the -= operator is used: PointChanged -= PointChangedHandler;.

What's Next?

Events are another powerful feature and they are a very convenient way of one bit of code telling other bits that something has taken place. This is common in programming so events tend to be the go-to solution for most programmers.

In the next section, we are going to look at C# threading, which is a way of running several code sections simultaneously, thus speeding things up significantly.

Threading

When computers first came out they all had a CPU, a processor if you like, and for those of you who don't know, this is the computer's brain. These days, modern computers have two or more processors and each one may well be hyperthreaded, which makes it feel like you have double the amount of power. Some of the top computers can have eight, even 16 processors and some of them can be linked so that other computers are working for yours too, which gives you an almost unlimited amount of processing power.

What has all this got to do with C# programming? This – most computers have an incredible amount of power but when we run a C# program, we run on one processor and that wastes an awful lot of computing power.

We're going to be looking at a feature called threading. With the basic process, we can take pieces of code that are all independent of one another and get them running in different 'threads'. A thread is pretty much like its own program in that your computer will run several at the same time on all the different processers in the system. Just to clear things up, the one thing a thread isn't is its own program because it still has to share memory with whatever program or process that created it.

When multiple threads are running simultaneously, your computer will allow one or more, depending on how many your system has, thread run for a little while before switching it out for another one suddenly. The computer decides the time for the switch and it decides which thread is being switched, but it does this quite efficiently so we don't need to worry too much about that.

The most important thing to know at this stage is that several threads can run simultaneously and it is down to the operating system to determine when they get switched about and with which thread. They are all treated much the same but they all get switched about on occasion and this randomness can cause a few problems at times – we'll cover this in Thread-safety.

Threading involves quite a bit and we don't have the space to go into it all. Instead, we'll look at the basics and what you need to know to go hunting for more information on your own.

Threading Tasks

There are lots of things that threading can be used for, and no real limits on what you can and can't do with them. There are, however, two main categories for which threading is particularly good. The first one is doing stuff in the background which means that things are happening but you can carry on updating your GUI or doing other things at the same time. This happens a lot, usually with a progress bar on display indicating how much has been done or is left to do.

The second one is when you have a ton of things to do and you can break it down into separate, independent parts. Multiple threads will take a piece each and get to work on it without getting in the way of the other threads.

As we said, there really isn't really a limit to what you can do with a thread but we are going to use a simple example from the second category. Assume that we have a program and it contains an array of ints. We want to add a load of random numbers to the array, numbers between 0 and 99.

If we didn't use threading, our program could look like this:

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


namespace Threading
{
   class Program
   {
        static void Main(string[] args)
        {
            int[] numbers = new int[10000];

            Random random = new Random();

            for (int index = 0; index < numbers.Length; index++)
            {
                numbers[index] = random.Next(100);
            }
        }
    }
}
```

However, what we are going to do is change things around in this code so that we can use threads. Take a look at the code; notice that, although we want to fill a big list with numbers, every one of the numbers can be randomly determined by itself. None of them are dependent on the number that comes before or after it. That makes it very easy for multiple threads to be created and each one provided with a number block to get to work.

Pulling the Work Out for Threading

The first thing we need to do is pull the work out that is to be threaded and we pull it to a location where it can be assigned to an individual thread. Very shortly, you will see that a thread is started with a method that it will work in but that method will not take parameters and it will not return anything. So, if we want it to have those parameters, which is really the work we want it to do, or if we want it to return something, the best thing to do would be to make a class that the work and the results can be stored in and then have a class method with no return value and no parameters that we can then call.

Before we jump in, take a few minutes to sit and think about this class. What might it look like? What instance variables and properties do you think should be in it? What is the method that the thread works in going to look like?

This is what I came up with:

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


namespace Threading
{
    public class GenerateNumbersTask
    {
        // This is done so that we have just one Random Class instance
        // being used, regardless of the number of tasks that we create.
        private static Random random = new Random();

        public int[] Numbers { get; set; }
        public int StartIndex { get; set; }
        public int Count { get; set; }

        public void GenerateNumbers()
        {
            for (int index = StartIndex; index < StartIndex + Count;
index++)
            {
               Numbers[index] = random.Next(100);
            }
        }
    }
}
```

Here, I have three properties. The first one is called Numbers and that will eventually have a reference pointing the array that is going to be filled with our random numbers. We are also going to want each of the threads to know exactly which bit of the array it will be working on. To facilitate that, a StartIndex is created – this tells the thread which index it will start from. Lastly, we have Count and this tells the thread how many numbers it will work on.

Let's say that we create 10 threads and then ask each one to generate 1000 numbers. Each thread will have a reference pointing the array called Numbers, and because they are all generating 1000 numbers each, Count would be identical. The first thread would be given a StartIndex of 0, the second a StartIndex of 1000, and so on.

Lastly, we have a method called GenerateNumbers. This is where the work of generating the random numbers is done with each index working in the range that was assigned to it. Note that the GenerateNumbers method has a return type of void which means nothing is returned and it doesn't have any parameter so we can start using it to get the thread started.

Starting the Thread

The .NET framework and C# have the threading code built right in it and you can find it in the System.Threading namespace. Using it requires you to put an extra statement into your code right at the very top together with all the other 'using' statements:

```
using System.Threading;
```

Starting a new thread is quite simple to do so that is where I will start and then I will move on to the code that you need to use to complete the example we started above.

To start a thread, you will need these lines, in one way or another:

```
Thread thread = new Thread(MethodNameHere);
thread.Start();
thread.Join();
```

The first line is what creates our new thread. A method name is passed into the Thread constructor and this method will match what the required delegate wants – a void return type and no parameters.

In line two, the new thread is started and the method we passed on will start running.

In the last line, we have the Join method; with this, the main thread waits for the worker thread to finish before it will continue. If we didn't have this, the main thread would carry on doing what comes next in the code. Now, let me just clear this up – this isn't always a bad thing; it might even be what you want. But, you will have no way of knowing when the thread is done unless you add an event that will be raised by the worker thread when it is finished.

As I said earlier, you can write one piece of code in several ways.

Now, with all of that under your belt, we can go back to the example and get it finished. What I'm going to do now might seem a bit batty to you so let me help you make sense of it. Because we want a lot of threads, 10 as we said earlier, and we need to keep track of each one, we need two separate arrays, each one with a length of 10. One is going to store the GenerateNumbersTask list and one will store the Thread objects list.

Once these have been created, each will be looped through so we get a new GenerateNumbersTask with the settings we need and get a new thread going on it. Once that is done, each thread is joined up with individually ensuring that each has completed their work before moving to the next one.

Does that make sense?

It probably will when you see the code that goes with it:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;


namespace Threading

{

    class Program

    {

        static void Main(string[] args)

        {

            int[] numbers = new int[10000];


            // The list of threads and the list of tasks for the threads to get
working on is prepared.

            int threadCount = 10;

            GenerateNumbersTask[] tasks = new
GenerateNumbersTask[threadCount];

            Thread[] threads = new Thread[threadCount];


            for (int index = 0; index < threadCount; index++)

            {
                // The task is created and it is set up with the correct
properties – these will be
                // our parameters.
                tasks[index] = new GenerateNumbersTask()
                {
                    Numbers = numbers,
                    Count = 1000,
                    StartIndex = index * 1000
                };

                // Create the thread
                threads[index] = new
Thread(tasks[index].GenerateNumbers);

                // Start the thread running
                threads[index].Start();
            }

            // Wait and then join up with each of the threads in turn so
that
            // when we move on, we know that all of the work has been
done.
            for (int index = 0; index < threadCount; index++)
            {
                threads[index].Join();
            }
        }
```

```
    }
}
```

Thread-safety

That covers the basics of C# threading, but now we will move on to something to do with threading that is more advanced. If you're finished with threads then, by all means, go to the next section but I think that what we are going to talk about is very important.

In the last example, each of our threads had its own piece of work to do and we had absolutely no overlapping. What if they had to share something? Remember, all threads are swapped out at a time the operating system chooses, and at that point, a thread could very well be doing some important task.

Try to imagine that you have a game and it has hundreds of objects in it, such as objects and players controlled by the computer. All of these objects have to be updated. They are all in one array and an index indicated which object is to be updated next.

We could split all of this work so that separate threads takes care of a different bit of it – each thread would be responsible for updating a section of objects. Because some of the objects will update quickly and some will take longer, we need to look at an approach where the worker threads update the objects, one after the other, until all of them are updated.

When a worker thread has nothing to do, it will call a method to help it work out what to do next. Inside the method, the thread can get a reference to the object that needs working on and push the index, showing what object needs to be worked on, to the next object. This way, when the next thread calls the method, it will be given the next object.

However, this is not quite as easy as we think it should be. What would happen if one thread got some more work, picked up the reference to an object to work on, but, before it can move the index to the next item, the operating system makes the decision to swap out the thread; that thread will be paused and the next thread will go in and get an object to update and our index hasn't been updated so the new thread grabs the object that the previous thread had. That means that a specific object will be updated twice.

It doesn't end there. Now the second thread will change that index to the next object so the third thread will get the correct object when it requests work. That leaves the second thread to carry on with its own updating work. While all this is going on, the operating system makes yet another switch and now our first thread is back in the mix again. This thread already knows which of the objects needs to be updated -the one at index 0 but thread 2 is also updating this. The first thread moves on and it moves the index again, onto 2 so the next object (1) is skipped over completely. One is updated twice while another is skipped altogether.

There is one main underlying issue – we have two different threads each trying to get access to or change the same data and neither has checked that it is safe to do this. This can be fixed. There is a way to mark bits of code and ensure that one thread can access that section at any one time. We call this thread-safety. If a thread reaches the section of code while another one is in it, it will need to wait until the first thread has gone.

This can slow things down quite a bit but the upside is you have consistent data. If your code doesn't need to be made thread-safe, don't do it, but when you do need it, make sure you do it.

Let's discuss how to make your code thread-safe. We will need two pieces of code for this. When a thread is getting ready to go into a piece of code that has to be thread-safe, we use a different keyword – lock. With this, we also need to have an object to lock on. The easiest way is to lock on an instance variable marked as private so you should go ahead and add one where the method that has to be thread-safe is in your main thread controller class:

private object threadLock = new object();

If thread-safety is needed across all the class instances, this can also be created as a static variable rather than an instance variable.

private static object threadLock = new object();

I don’t need to tell you that you can name your variable whatever you want; I use threadLock for the sake of simplicity.

To use the threadLock object to make a code block safe, you would add code like this:

```
lock(threadLock)
{
   // The code here is now thread-safe.
   // We can only have one thread in here at a time.
   // The other others will need to wait
   // at the "lock" statement.
}
```

Quick Primer

- Threading lets you run several sections of code at the same time.
- C# makes it easy to start a new thread: Thread thread = new Thread(MethodNameHere); thread.Start().
- Whatever method you use, it must have the void return type and it cannot take parameters. If you want a thread to return information or you want it to pass parameters in, you need to make an object that contains the information you want as properties or instance variables and one method that have a void return type and no parameters for the thread to run in.
- You can use the Join method to wait for threads to finish: thread.Join().
- Use the lock keyword if you need to make a section of code thread-safe: lock(aPrivateObject) { /* the code here is thread-safe */}. Usually, if the code you want to make thread-safe is all inside one class, a private instance variable is needed. If you

need to make all class instances thread-safe, use a private static variable.

What's Next?

Threading is quite a complex feature and isn’t all that easy to learn. Not all programs will need it and sometimes, you will be using a framework, such as WinForms, XNA, or WPF, that takes care of thread-safety behind the scenes. But, when you need to use it, at least now you will have a basic understanding of how it works and how to do it.

Next, we are going to look at something that is quite important in C# - operator overloading.

Operator Overloading

So far, throughout this guide, we’ve seen and used many different operators. Each has its own functionality built-in that will do specific things. Most of the time, these operators and their functions are defined by standard math although there are one or two that do some not-so-standard stuff – the + operator, for example, used for concatenating or adding two strings together. There is no way to do that in standard meth but there is in C#.

When you create a class in C#, you can define the way the operators work with the class. For example, if you created an object called ‘Burger’, you could use the + operator to add a pair of Burger objects together and you also can determine exactly what it means. We call this operator overloading and it is one of the nicest C# features. Those of you who have a programming background will already know that, and you can also do operator overloading in C++, but you can’t do it in Java.

Not all operators can be overloaded. For example, you cannot overload the assignment (=) operator; it isn’t possible for you to define what that operator does and that’s no bad thing – you could cause all sorts of problems if you could!

Lots of them can be overloaded and you’ll obviously want to know which ones. These are the most common ones that you will use all of the time:

- +
- -
- *
- /
- %
- ++
- --
- ==
- !=
- >
- <

- >=
- <=

There are some others including +=, -=, *=, /=, ad %=. However, when the + operator is overloaded, it will automatically overload the += operator.

You should also know that when you overload a relational operator, you must do them in pairs. If the == operator is overloaded, so must the != operator be overloaded. And if the > operator is overloaded, you also need to overload the < operator, and so on.

We mustn't forget that some of the operators have two versions – unary and binary. Take the + operator, for example. The binary version will do operations on two things, for instance, adding 4 + 6 – the operator will apply to both numbers separately. The unary operator, on the other hand, will only do operations on one thing. Take -9 for example; the operator is only applied to the 9, no other number.

What can't be overloaded?

- &&
- ||
- =
- . (dot operator)
- new
- and a few others that we haven't looked at.

The [and] operator, which is used for indexing, may be overloaded by using indexers, but we will come onto that in the next section.

Overloading Operators

Operators can be overloaded for any class that you want, but to keep things relatively simple here, I will choose a class that actually makes sense from the overloading perspective. We'll use the Point class to do the overloading – this stores two coordinates of X and Y. If you understand vectors from physics and math then this will make some sense to you.

Let's begin with a basic class:

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

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

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

If you think back to your school days when you did math and physics, you will remember that two vectors can be added together. When you do this, you get another vector as the result, adding the X and Y components.

Overloading the + operator means adding the following code to the class called Vector:

```
public static Vector operator +(Vector v1, Vector v2)
```

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

No matter what one it is, every operator overload must be static and public and that should make sense to you; we want to be able to access the operator in the code and it doesn't just belong to a class instance; it belong to the entire class.

The next thing we do is specify what return type we want; in our case, this will be Vector. The operator keyword is used together with the specific operator we want to overload, which is the + operator. Next, we have two parameters, one for each side of the operator.

If we were using a unary operator, such as the negation (-) operator that makes the number a negative, we would have just the one parameter:

```
public static Vector operator -(Vector v)
{
   return new Vector(-v.X, -v.Y);
}
```

We can also have several versions of one operator:

```
public static Vector operator +(Vector v, double scalar)
{
   return new Vector(v.X + scalar, v.Y + scalar);
}
```

Now we can add a vector and a scalar together, which is nothing more than a plain number, something that simply doesn't make sense from a standard math perspective.

We can overload the relational operators the same way but the return must be a bool:

```
public static bool operator ==(Vector v1, Vector v2)
```

```
{
   return ((v1.X == v2.X) && (v1.Y == v2.Y));
}

public static bool operator !=(Vector v1, Vector v2)
{
   return !(v1 == v2);
}
```

Don’t forget – I mentioned earlier when you overload relational operators, they must be done in pairs. So, if you overload ==, you also have to overload != as you can see in the above example, although you can call the other operator instead if you wanted.

Just have a closer look at the overloads; you should be able to see that they are nothing more than methods with one difference – that operator keyword. C# just happens in any part of your code where the operator is used and converts it into a method call. Really and truthfully, this is nothing more than what is known as syntactic sugar – it just makes it look neat and tidy.

Now, our complete class should be like this:

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

namespace OperatorOverloading
{
   public class Vector
   {
```

```
public double X { get; set; }
public double Y { get; set; }

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

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

public static Vector operator +(Vector v, double scalar)
{
    return new Vector(v.X + scalar, v.Y + scalar);
}

public static Vector operator -(Vector v)
{
    return new Vector(-v.X, -v.Y);
}

public static bool operator ==(Vector v1, Vector v2)
{
```

```
            return ((v1.X == v2.X) && (v1.Y == v2.Y));
        }

        public static bool operator !=(Vector v1, Vector v2)
        {
            return !(v1 == v2);
        }
    }
}
```

Now the operator overloads have been defined, they can be used like any normal operator:

```
Vector a = new Vector(5, 2);
Vector b = new Vector(-3, 4);
Vector result = a + b; // The operator overload is used here.
// Here, the result is <2, 6>.
```

Quick Primer

- Operator overloading is a way of letting us define the way traditional operators work for the objects and the classes we create.
- Operator overloading will work on the =, -, /, *, %, ++, ==, --, !=, <, >, <=, and >= operators, along with a couple of others that we haven't talked about.
- It will not work on the assignment ==, dot (.), new, &&, or || operators.
- Overloading the + operator could look something like this: public static Vector operator +(Vector v1, Vector v2){ return new Vector(v1.X +v2X, v1.Y + v2.Y;}.
- Every operator must be marked as static and public.
- When the relational operators are overloaded, it must be in pairs so, for example, if == is overloaded, so != must be

overloaded.

- The [and] indexing operator can also be overloaded but this is done in a different way which we will see shortly.

What's Next?

Operator overloading is one of the simplest yet most powerful ways of making your code much easier to read. In the next section, we will look at indexers, a very similar way of overloading the [and] operators.

Indexers

We discussed operator overloading in the last section and, in this one, we'll extend that briefly by looking at indexers. An indexer is much the same as overloading the indexing operator. As with the operator overloading, just because you can do it, it doesn't mean that you should do it all the time.

Making an Indexer

When you define an indexer, it is much like a combination of a property and overloading an operator. It isn't hard to do, so the first thing we'll do is look at the piece of code that lets you do it. You could add this to your Vector class from earlier but, on the other hand, if you modified what the set and get accessors did you could put in any class:

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

What we've done here is specified what access level the indexer has – we used public – and then specified the return type. The 'this' keyword is used with the [] square brackets used to indicate indexing. In the square brackets, we put the indexing variable type and name that is going to be used in the indexer.

Then, in much the same way as we do with a property, we have our blocks of get and set code. Again like a property, you don't have to use both if you don't want or need to – just stick with one. Inside of these blocks, the variable called index can be used along with the value keyword inside the setter referencing the value that will be assigned.

In this example, I have made it so that the x and y components of the vector can be referenced by others using the 0 index for x and the 1 index for y. With all of that, we can now do this:

```
Vector v = new Vector(5, 2);

double xComponent = v[0]; // Now we can use the indexing operator.

double yComponent = v[1];
```

This is a good deal easier to read and understand than the alternative would be – that would have had a method called GetIndex, or something like it and would have read xComponent = v.GetIndex(0);.

But you don't just have to use an ints for your index. You could also use doubles, strings, or anything else that you want. In fact, think about when we talked about generics; the Dictionary class does just that.

This example shows you how to do indexing with strings:

```
public double this[string component]

{

   get

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

This is similar to what we used earlier, but we used a string for the indexing instead rather than an int. If x is asked for, the x-component is returned, and if y is asked for, the y-component is returned.

So now we can do this:

```
Vector v = new Vector(5, 2);

double xComponent = v["x"]; // Now we can use the indexing
operator.

double yComponent = v["y"];
```

This gets better because we can also index using several indices. Again, you may be able to do it but it doesn’t mean you should. Only do this when it is appropriate to do so.

To add several indices, all you do is list them inside the square brackets where the indexer is defined.

```
public double this[string component, int index]
{
    get
    {
        // I’m not quite sure what to do with the two indices we have
here.
        // I think I might just throw an exception. If making an indexer
with multiple
        // indices make sense, you would write some code right here
that would do
        // something intelligent with it.
        throw new Exception();
    }
    set
    {
         // I’m not quite sure what to do with the two indices we have
here.
        // I think I might just throw an exception. If making an indexer
with multiple
        // indices make sense, you would write some code right here
that would do
        // something intelligent with it.
        throw new Exception();
    }
}
```

In this case, it does make perfect sense and it shows you that it can be done; if you can use it, it’s easy enough to make it happen.

Quick Primer

- Indexers are a way of overloading the indexing [and] operator for classes.
- Defining an indexer is a lot like a combination of a property and

overloading an operator: public double this[int index] {get { /* get code goes here */ } set { /* set code goes here */ }}.

- You are not limited to using ints; any type can be used for indexing.
- To use multiple indices, you just list them where the indexer is defined.

Rather than using [int index], you could use [string someTextIndex, int numericIndex].

What's Next?

We have discussed overloading operators, a way in which we can make them look and work how we want with custom classes and then we moved on to indexers, which is pretty much the same thing using the indexing operator. Next, we move on to doing the exact same thing but using user-defined conversions. This means we can indicate how a class can be converted to another type.

User-Defined Conversions

With operator overloading, we can make the operators we use work in special ways with our classes and we can do much the same using the indexing operator. Now we want to look at how to custom define a way of ensuring that our classes can be cast or converted to different types. You will find that both cast and conversion are used interchangeably because they both mean the same thing – changing from a type into another type.

Implicit vs. Explicit Conversions

Before we discuss the user-defined conversions, we should explain how conversions work.

As an example, we'll look at two types – float and double. Let's say that we have a float; we can use this as a double if we want and it will be converted to a double automatically by the program. Here's how:

float a = 3.5f;

double b = a; // the 'a' value gets converted into a double automatically when it gets assigned to 'b'.

However, if we tried to do it the other way around, all that would happen would be a compile time error:

double b = 3.5;

float a = b; // this will not compile because you are attempting to convert a double into a float.

Why can't we do this? It comes down to size. You cannot put a double into something as small as a float that doesn't have the accuracy and precision of the double – all you will achieve is a loss of information.

That said, there is a way to convert it; you need to state explicitly that this is what you want to do:

double b = 3.5;

```
float a = (float)b;
```

What you have done here is inform the compiler that you do want to convert the double to a float even though you will likely lose information along the way.

That is the difference between implicit and explicit conversion. When converting a float to a double, the implication is that it can be changed very easily without having to do anything special. With the explicit conversion, you have to make it clear that you do know exactly what you are doing and that yes, you do want to convert this type to a different one. This is done using the cast operator (‘ and ‘) placing the type in between the set of parentheses, as the last example.

When we create some custom conversions, we will need to specify whether we are using implicit or explicit casting.

Creating a User-Defined Conversion

To show you how it works, we’ll go back to the MagicNumber class we created before:

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

namespace UserDefinedConversions
{
    public class MagicNumber
    {
        public int Number { get; set; }
        public bool IsMagic { get; set; }
    }
}
```

This class is simple; a number is stored as an int and we have a bool that tells us whether our number is 'magic' or not, whatever that means from the perspective of your program.

What you can most likely see is that it is possible to imagine how to take an int and turn it into a MagicNumber and vice versa – taking a MagicNumber and converting it into an int. Doing this requires just a small piece of code so that the conversions can happen.

To convert the int into a MagicNumber, we could do this implicitly, not needing to use (MagicNumber) to cast it with, just bringing the int in and making IsMagic evaluate false:

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

As with any operator, this one has to be marked as static and public. Next, we need to state whether the conversion is implicit or explicit – we've chosen implicit for this example.

Next, we specify the type that the conversion happens to – MagicNumber – and, inside a set of parentheses, the type that is being converted from. In the conversion body, we have the code that does the conversion.

Adding this to the MagicNumbers class, the conversion from an int to a MagicNumber is quite simple:

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

Creating an explicit cast is done in much the same way. We will now define a conversion from a MagicNumber to an int. It is, by the way, a good idea to require the cast to be explicit when information is going to be lost in the process of converting.

Making the cast explicit is as simple as adding something like this into our class:

```
static public explicit operator int(MagicNumber magicNumber)

{

   return magicNumber.Number;

}
```

As before, it has to be marked as both public and static. The explicit keyword is used for indicating that and, in order to use this, it must be explicitly stated that you want the conversion used along with the conversion operator. The operator keyword is used and, as we did before, the type we want to convert to. Inside our parentheses, we state what type we are converting from.

In the conversion block is the code we are using to do the conversion and, in this example, we take the number for MagicNumber and we leave IsMagic behind. What happens is that we lose the information that tells us whether the number was or want magic – whether IsMagic was set as true or false.

You can use this conversion elsewhere in the code but, for now, we need to make it clear that we are explicitly casting from one type to another:

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

int aNumber = (int)magicNumber;
```

Quick Primer

- User-defined conversions let you define the way that the compiler and
  runtime convert to and from user-created classes.
- Implicit casting is when the change can be made without the conversion
  operator being needed. Explicit casting does require the operator. For

example, implicitly casting from a float to a double – float a = 3; double b
= a;. The value assigned to a gets converted to a double automatically.
Then, because we cannot implicitly cast a double to a float so we need to do
it explicitly – double a = 3; float b = (float)a;. The value in a can be
converted into a float so long as it is explicitly stated that the conversion is
being made.

- Creating a user-defined conversion requires you to follow this – static
public implicit operator MagicNumber(int value) { /* your conversion is
done here to return a MagicNumber */};.
- To indicate the type of conversion you are doing, you can use the
keywords, implicit or explicit.

What's Next?

User-defined conversions are great features and used properly, they can turn your code into something easy to read and neat; and they are very powerful too.

In the next section, we look at extension methods, a C# feature that lets you add methods to classes that you never created and that you aren’t able to modify directly or add to. That will bring us to the end of this advanced section.

Extension Methods

Jumping straight into this, let’s assume that we have a class but somebody else created it. It could be a part of the .NET framework, such as the String class, which is the equivalent of a type we have made good use of – the string type.

Now let’s say that we have a method and we want to use it on the string type but you don’t have the class right there to modify. The String class or string type has two methods called ToUpper() and ToLower(), but what if we want to make a method that would convert it to another type, for example making it random as to whether each letter is lower or upper case, not one or the other? Actually making that method is quite simple. We want to say something along the lines of “Hello World!”.ToRandomCase() instead of saying ToUpper() or ToLower() and, without being able to access the class, we wouldn’t usually be able to add a new method of ToRandomCase(). That is not normal.

However, C# has a way for us to do that, using something called extension methods. What we do is create a static method inside a static class, and using the ‘this’ keyword, we create a method that can be used just like it belonged to the original class.

Creating an Extension Method

It really isn’t difficult to create extension methods – first, we create a static class and then we put a static method in it to do exactly what we require for our extension method.

The first thing to do is to add a new class file. You can call it what you want, but if I am going to be creating string class extension methods, I would call it StringExtensions. It is good practice to make one class for each of the classes that the extension methods are being made for. In short, every extension method that goes with the string class go into the class called StringExtensions; any extensions methods for the Point class would go in a class called PointExtensions, and so on – you get the idea. It lets the extension methods belong to the class without being in it.

}

On top of that, we want the static keyword to be on the class, thus ensuring the entire class is a static class. To begin with, the basic class, no extension methods yet, would look like this:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;


namespace ExtensionMethods

{

   public static class StringExtensions

   {

   }

}
```

We haven't talked very much about static classes yet so I will just explain briefly about what it means. When a class is static, every method inside the class must also be static. That also means that instances of static classes cannot be made – imagine saying something like StringExtensions extensions = new StringExtensions();. You just can't do it and while that might seem to be somewhat restrictive, it does mean that the compiler and .NET framework can handle things in a way that benefits performance.

Defining the extension method is simple – create a method as static and make the first parameter as the object type that we want the extension method for, i.e. string, and mark it using the 'this' keyword:

```
public static string ToRandomCase(this string text)

{

   // We will put the method implementation here shortly...

}
```

A return type can be indicated and, as well as the initial parameter that uses the ‘this’ keyword, we could add as many other parameters as we want.

As it was indexers and operator overloading, this is nothing more than syntactic sugar. The compiler will go ahead and change anywhere that the extension method is called into nothing more than calling a method (the extension in this case). When we have finished, we can simply say:

string title = "Hello World!"

string randomCaseTitle = title.ToRandomCase();

And the compiler will turn that into this:

string title = "HelloWorld";

string randomCaseTitle = StringExtensions.ToRandomCase(title);

What we end up with is a piece of code that looks better, is far easier to read, and looks like nothing more than a proper method for the string class.

But this isn’t all it seems to be. While the method will seem like it is part of the original string class, it isn’t. If your extension method isn’t in the same namespace as the original class, you could face a problem where your class is recognized but the extension method won’t be found, or you might move to another project only to find that what you saw as an original class method was actually an extension method that someone else wrote and now you can’t use it.

I’m not saying don’t use these extension methods. You should use them or I wouldn’t be telling you about them, but you just need to keep in mind that, while methods may look like part of a class, it might not be; it might be an extension method.

You might also need to have a using statement in the extension method, but that will be dependent on the namespace you used.

Let’s complete the code example by finishing the ToRandomCase method body:

```
string result = "";

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

return result;
```

One character at a time, this will go through our original string and pick a random number, which will be 0 or 1. For 1, the character is made upper case and for 1, the character is made lower case. That way, we end up with a collection of random upper and lower case letters which gives us the result we wanted.

The entire code should now look like this:

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

namespace ExtensionMethods
```

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

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

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

            return result;
        }
    }
}
```

As we said before, if your program knows there is an extension method, we can do this:

string message = "I'm so sorry, Simon. I really can’t do that.";

Console.WriteLine(message.ToRandomCase());

The extension method can be used as if it was part of our original class.

Quick Primer

- Extension methods help us to create methods that seem as though they are part of a class, like the string class, when you can’t access the class to modify it, and then help us to add the method.
- You must add an extension method as a static method in a static class. The first parameter must be the type for the class you want the method added to and you must use the ‘this’ keyword, i.e. public static class StringExtensions { public static string ToRandomCase(this text for the string){ /*. We put the method implementation here */}}.
- When you have created an extension method, it can be called just as if it were a proper part of the class: string text = “Hello World!”; randomCaseText = text.ToRandomCase();.
- An extension method is just a way of making your code look presentable and easier to read. Whatever you write, the compiler will rewrite it whenever the extension is method is used to call the static method directly.

Well, that brings us to the end of this advanced section, but we are not finished, not by a long shot. First, we’ll take a break. I’ve given you some questions to answer in the next section (answers at the back of the book) and then we’ll move on to more advanced topics.

Quick Quiz

1. Of the four choices below, which one is NOT allowed as an access modifier n C#?

a) public
b) friend
c) protected
d) internal

2. Look at the C# code and decide, from the choices, what [int i] is.

```
class MyClass
{
// ...

public string this[int i]
{
get{ return arr[i];}
set{ arr[i] = value; }
}
}
```

a) Property
b) Event
c) Indexer
d) Delegate

3. Look at the C# keywords below – which one is not associated with multithreading?

a) async
b) await
c) sealed
d) lock

4. In the C# Generics examples below, which is the invalid expression:
a) class A where T : class, new()
b) class A where T : struct, IComparable
c) class A where T : Stream where U : IDisposable
d) class A where T : class, struct

5. In C#, the new keyword is used to create new objects from a type. Which of these cannot make use of the new keyword:
a) Interface : var a = new IComparable();
b) Class: var a = new Class1();
c) Struct : var a = new Struct1();
d) C# object : var a = new object();

6. Look at the four examples below where button1 is used in WinForms as a Button Class object. Which one is an incorrect click event handler expression?
a) button1.Click += (s, e) => MessageBox.Show("Click");
b) button1.Click += delegate(EventArgs e) {MessageBox.Show("Click");};
c) button1.Click += new System.EventHandler(button1_Click);
d) button1.Click += delegate { MessageBox.Show("Click"); };

7. Look closely at the C# code below. Which of the using statements is incorrect?

```
using System; // 1.
namespace  Co
{
        using System.Text; // 2.
        namespace App
        {
```

```
        using System.IO; // 3.
        class Class1
        {
        using System.Text; // 4.
        static void A()
        {
        }
        }
        }
}|
```

a) 1
b) 2
c) 3
d) 4

8. Look at the following C# code – what will the output be?

```
int? i = 8 >> 5;
int? j = i > 0 ? i : null;
var a = j ?? int.MinValue;
Console.WriteLine(a);
```

a) 1
b) Null
c) 0
d) -2147483648

9. Which of the following statements regarding C# exceptions is correct:

a) C# exceptions occur at linking time
b) C# exceptions occur at compile time

c) C# exceptions occur at run time
d) C# exceptions occur at JIT compile time

10. Which of the following Main() method prototypes, the C# entry point, is invalid?
a) public static long Main(string[] args)
b) public static int Main()
c) public static void Main()
d) public static void Main(string[] s)

11. In the following code that uses Generics, which is NOT the correct explanation?

```
T t = default(T);
```

a) If T is int type, variable t will have 0
b) If T is string type, variable t will have an empty string
c) If T is bool type, variable t will have false
d) If T is a reference type, variable t will have null

12. A number of built-in data structures are supported by C# and .NET. Which of these data structures is NOT built-in?
a) Array
b) D-Array
c) Binary Tree
d) Stack
e) Linked List

13. Which of these interfaces should you implement if you want to use LINQ to Objects?
a) IEnumerable
b) Ienumerable
c) ICollection
d) Icollection

14. Look at the following code – what will the result be?

```
List<string> names = new List<string>();
names.AddRange(new []{“Park”, “Tom”, “James”});
var v = names.OrderBy(n => n.length)
        .SingleOrDefault();
Console.WriteLine(v);
```

a) The console will display null
b) The console will display 4
c) A compile error will happen
d) A runtime exception will happen

15. Which of the following is the wrong way to C# var?
a) var a = null;
b) var a = 3.141592;
c) var a = db.Stores.Single(p => p.Id == 1);
d) var a = db.Stores;

16. In the example below, variable a is a string. Which of these is wrong or returns a result which is different?
a) a = a == null ? "" : a;
b) a = (a is null) ? "" : a;
c) a = a ?? "";
d) if (a == null) a = "";

17. Which of the following assemblies should you use to provide resources to other languages or cultures?
a) Public Assembly
b) Shared Assembly
c) Private Assembly

d) Satellite Assembly

18. Which of these will NOT let you use the static keyword in C#?

a) (Method) static void Run() {}
b) (Field) static int _field;
c) (Destructor) static ~MyClass() {}
d) (Property) static int Prop {get; set;}
e) (Class) static class MyClass {}
f) (Event) static event EventHandler evt;
g) (Constructor) static MyClass() {}

19. In C#, what is like the C++ function pointer?

a) Event
b) Method
c) Interface
d) Delegate

20. Which of these is NOT a valid C# method?

a) private var GetData() {}
b) public dynamic Get() {}
c) protected override int[] A() {}
d) public void Set(dynamic o) {}

21. Which of these statements is NOT a valid way of creating a new C# object?

a) var a = new [] {0};
b) var a = new String();
c) var a = new IComparable();
d) var a = new Int32();

22. Which of these operators CANNOT be used for overloading?

a) operator true
b) operator ++
c) operator &
d) operator ||

23. In the example below, Class A has an attribute of [Serializable()}. When does that attribute get checked?

a) At Linking
b) At compile time
c) At CLR runtime
d) At JIT compile time

24. Here are some integer array examples. Which of them is NOT valid in the C# language?

a) int[][][] cc = new int[10][2][];
b) int[][] c = new int[10][];
c) int[] a = new int[10];
d) int[, , ,] d = new int[10, 2, 2, 2];
e) int[,] b = new int[10, 2];

25. Look at these statements about the anonymous type in C#. Which one is true?

a) It can add an event
b) Once created, it can add a new property
c) The anonymous type allows you to use delegates for methods
d) It is an immutable type

26. Look at the code snippet and determine which of the following choices is the result of variable a and b.

```
var a = 5L == 5.0F;
var b = 24L / 5 == 24 / 5d;
```

a) a=true, b=false
b) a=true, b=true
c) a=false, b=false
d) a=false, b=true

27. Which of these is NOT valid when you use C# Generics to define a class?
a) class MyClass where T : struct
b) class MyClass where T : IComparable
c) class MyClass where T : MyBase
d) class MyClass where T : class
e) All of the above are correct

28. Which of these statements about delegates is NOT correct?
a) You can use delegates when you pass references to methods
b) C# delegates have support for multicast
c) The C# delegate is considered to be the technical base for an event

Part 4: More Advanced Section

C# Reflection

We left our advanced section after covering extension methods and now it’s time to continue. We start with Reflection, a feature that we use to get the metadata on the types we use at runtime. What that means is that you can inspect, look at the metadata on those types inside your program – dynamically – and get the information that is on the loaded assemblies, along with the types that were defined inside them. If you have a C++ background, then you will see that C# Reflection is very similar to RTTI, or Runtime Type Information.

With .Net, using reflection requires that you include a namespace called System.Reflection in your program. When you use reflection, get objects of the type called Type, are used for representing modules, types, and assemblies. Reflection can also be used for dynamically creating instances of types and invoking methods of the type.

In the System.Reflection namespace, there are some types that have already been defined:

- Assembly
- ConstructorInfo
- Enum
- EventInfo
- FieldInfo
- MemberInfo
- MethodInfo
- Module
- ParameterInfo
- PropertyInfo
- Type

Let’s have a look at some code that uses reflection. Here, we start with a class named Customer:

public class Customer

```
{
    public int Id
    {
        get; set;
    }
    public string FirstName
    {
        get; set;
    }
    public string LastName
    {
        get; set;
    }
    public string Address
    {
        get; set;
    }
}
```

In the next snippet, you can see how to use reflection to get the name of the class and Customer class namespace name:

```
Type type = typeof(Customer);

Console.WriteLine("Class: " + type.Name);

Console.WriteLine("Namespace: " + type.Namespace);
```

Here you can see how to retrieve a list of the Customer class properties and have their names displayed in the console window:

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

```
        {
            Type type = typeof(Customer);
            PropertyInfo[] propertyInfo = type.GetProperties();
            Console.WriteLine("The Customer class properties list is:--");
            foreach (PropertyInfo pInfo in propertyInfo)
            {
                Console.WriteLine(pInfo.Name);
            }
        }
```

The method called GetProperties for the Type class will return an array of the PropertyInfo type, which lists the public properties for the type. This array can then be iterated and we can retrieve a name for each public property that has been defined in the type. Because three properties are defined in the Customer class, all three names would display in the console window on the execution of the window.

Below, the example shows how to use reflection to display the metadata for the public methods and the constructors of a specified type. Going back to the Customer class we defined earlier, we are going to add two new methods – a method named Validate used to validate the parameter, which is a customer object, and a default constructor. The new version of our Customer class should now look like this:

```
public class Customer
    {
        public Customer()
        {
            //Default constructor
        }
        public int Id
        {
            get; set;
        }
        public string FirstName
        {
            get; set;
        }
        public string LastName
        {
            get; set;
        }
        public string Address
        {
            get; set;
        }
        public bool Validate(Customer customerObj)
        {
            //This code will be used for validating the customer object
            return true;
        }
    }
```

The next snippet shows you how to display all the constructor names belonging to Customer. We only have the one constructor in our class so we would only see the one displayed:

```
Type type = typeof(Customer);
ConstructorInfo[] constructorInfo = type.GetConstructors();
```

```
Console.WriteLine("These are the constructors in our Customer class:--");
foreach (ConstructorInfo c in constructorInfo)
 {
   Console.WriteLine(c);
 }
```

The method called GetConstructors() in the Type class will return an array of the ConstructorInfo type – basically, a list of the public constructors that are defined inside the reflected type.

Ok, so now we want to display all the Customer class public methods and, again, we only have the one so all you would see on the console when the program is executed is the one name. To do this, the code would look like this:

```
static void Main(string[] args)
 {
   Type type = typeof(Customer);
   MethodInfo[] methodInfo = type.GetMethods();
    Console.WriteLine("The Customer class methods are:--");
          foreach (MethodInfo temp in methodInfo)
          {
            Console.WriteLine(temp.Name);
          }
          Console.Read();
      }
```

Note, you may see the names of some extra methods, such as Equals, ToString, GetType, and GetHashCode. That is because those methods were inherited from the class called Object and, as you already know, all classes in .NET will derive from that class because Object is the base class of everything.

The code below shows you how to iterate through a method's attributes. If you have defined custom attributes of the methods, the method called GetCustomAttributes can be used in the instance of the class called MethodInfo to get the method attributes:

```
foreach (MethodInfo temp in methodInfo)
 {
   foreach (Attribute attribute in temp.GetCustomAttributes(true))
    {
       //You usual code will go here
    }
 }
```

If you use attributes to decorate business objects in your application, C# reflection can be used to reflect on the type, get the attributes for the type methods, and then do whatever action needs to be performed.

C# Type Class

The C# Type class is used to represent the type declarations for the interface types, class types, array type, enumeration types, value types, and so on. It is located in the System namespace and inherits the class called System.Reflection.MemberInfo.

Here is a list of the more important properties of the Type class:

Property	**Description**
Assembly	Retrieves the assembly for the type
nblyQualifiedName	Retrieves the name of the type that is Assembly qualified
ıtes	Retrieve the attributes that are associated with the specified type

BaseType	Retrieves the parent or base type
ame	Retrieves the fully qualified name of the specified type
IsAbstract	Used for checking if the type is an Abstract type
IsArray	Used for checking if the type is an Array type
IsClass	Used for checking if the type is a Class type
IsEnum	Used for checking if the type is an Enum type
IsInterface	Used for checking if the type is an Interface type
IsNested	Used for checking if the type is a Nested type
IsPrimitive	Used for checking if the type is a Primitive type
IsPointer	Used for checking if the type is a Pointer type
IsNotPublic	Used for checking if the type is not a Public type
IsPublic	Used for checking if the type is a Public type
IsSealed	Used for checking if the type is a Sealed type
IsSerializable	Used for checking if the type is a Serializable type
erType	Used for checking if the type is a Member type of the Nested Type
Module	Retrieves the type's module
Name	Retrieves the type's name
Namespace	Retrieves the type's namespace

C# Type Methods

Below you can find a list of the more important Type class methods:

Method	**Description**
GetConstructors()	This will return all of the Type's public constructors

onstructors(BindingFlags)	This will return all the Type’s constructors that have the specified BindingFlags
GetFields()	This will return all the Type’s public fields
elds(BindingFlags)	This will return all the Type’s public constructors with the specified BindingFlags
GetMembers()	This will return all the public members for the Type.
embers(BindingFlags)	This will return all the Type’s members with the specified BindingFlags
GetMethods()	This will return the Type’s public methods
ethods(BindingFlags)	This will return all the Type’s methods with the specified BindingFlags
GetProperties()	This will return all the Type’s public properties
operties(BindingFlags)	This will return all the Type’s properties with the specified BindingFlags
GetType()	This will get the current Type
GetType(String)	This will get the Type for the specified name

We’ll finish with a series of examples so you can see how some of these work:

Get Type

```
using System;
public class ReflectionExample
{
  public static void Main()
   {
      int a = 10;
```

```
        Type type = a.GetType();
        Console.WriteLine(type);
    }
}
```

The output would be:

```
System.Int32
```

Get Assembly

```
using System;
using System.Reflection;
public class ReflectionExample
{
   public static void Main()
    {
        Type t = typeof(System.String);
        Console.WriteLine(t.Assembly);
   }
}
```

The output would be:

```
mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

Print Type Information

```
using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);
        Console.WriteLine(t.FullName);
        Console.WriteLine(t.BaseType);
        Console.WriteLine(t.IsClass);
        Console.WriteLine(t.IsEnum);
        Console.WriteLine(t.IsInterface);
    }
}
```

The output would be:

```
System.String
System.Object
true
false
false
```

Print Constructors

```
using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);

        Console.WriteLine("Constructors of {0} type...", t);
        ConstructorInfo[] ci = t.GetConstructors(BindingFlags.Public | Bi
ndingFlags.Instance);
        foreach (ConstructorInfo c in ci)
        {
            Console.WriteLine(c);
        }
    }
}
```

The output would be:

```
Constructors of System.String type...
Void .ctor(Char*)
Void .ctor(Char*, Int32, Int32)
Void .ctor(SByte*)
Void .ctor(SByte*, Int32, Int32)
Void .ctor(SByte*, Int32, Int32, System.Text.Encoding)
Void .ctor(Char[], Int32, Int32)
Void .ctor(Char[])
Void .ctor(Char, Int32)
```

Print Methods

```
using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);
        Console.WriteLine("Methods of {0} type...", t);
        MethodInfo[] ci = t.GetMethods(BindingFlags.Public | BindingFl
ags.Instance);
        foreach (MethodInfo m in ci)
        {
            Console.WriteLine(m);
        }
    }
}
```

The output would be:

```
Methods of System.String type...
Boolean Equals(System.Object)
Boolean Equals(System.String)
Boolean Equals(System.String, System.StringComparison)
Char get_Chars(Int32)
Void copyTo(Int32, char[], Int32, Int32)
Char[] ToCharArray()
....
```

Print Fields

```
using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);

        Console.WriteLine("Fields of {0} type...", t);
        FieldInfo[] ci = t.GetFields(BindingFlags.Public | BindingFlags.St
atic | BindingFlags.NonPublic);
        foreach (FieldInfo f in ci)
        {
            Console.WriteLine(f);
        }
    }
}
```

The output would be:

```
Fields of System.String type...
System.String Empty
Int32 TrimHead
Int32 TrimTail
Int32 TrimBoth
Int32 charPtrAlignConst
Int32 alignConst
```

Anonymous Functions

An anonymous function is a function that does not have a name and, in C#, there are two types of them:

- Lambda Expressions
- Anonymous Methods

We use the lambda expression to create an anonymous function.

A lambda expression is an anonymous function that has a sequence of operators or expressions. Every lambda expression must use the lambda operator (=>) which means “becomes” or “goes to”.

The input parameters go on the left side of the operator and an expression or a block of code goes on the right and working with the entry parameters. Normally, a lambda expression is used instead of a delegate, which, if you remember, is a type used for referencing a method or used as a predicate.

Expression Lambdas

- Parameter => expression
- Parameter-list => expression
- Count => count + 2;
- Sum => sum + 2;
- n => n % 2 == 0

The lambda operator is used to divide lambda expressions into two parts – the input parameter and the body of the lambda. Here’s a simple example:

```
using System;

using System.Collections.Generic;

using System.Linq;

public static class demo

{

   public static void Main()
```

```
    {
        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
        List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

        foreach (var num in evenNumbers)
        {
            Console.Write("{0} ", num);
        }
        Console.WriteLine();
        Console.Read();

    }
}
```

The output would be:

```
2 4 6
```

In this example, we loop through all the numbers in the collection and every element with the name of x is looked at to work out whether the number is a multiple of 2 or not – this is done using a Boolean expression – (x % 2) == 0
The preceding example loops through the entire collection of numbers and each element (named x) is checked to determine if the number is a multiple of 2 (using the Boolean expression (x % 2) == 0).

Here's another example:

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

```
class Cat
{
    public string Name { get; set; }
    public int Age { get; set; }
}
class demo{
    static void Main()
    {
        List<Cat> Cats = new List<Cat>() {
            new Cat { Name = "Molly", Age = 4 },
            new Cat { Name = "Simon", Age = 0 },
            new Cat { Name = "Serendipity", Age = 3 }
        };
        var names = Cats.Select(x => x.Name);
        foreach (var name in names)
        {
            Console.WriteLine(name);


        }
        Console.Read();
    }
}
```

The output is:

```
Molly
Simon
Serendipity
```

What we have done here is created a collection that contains data from a specified class. We used the class called Cat which contains properties called Name and Age. From this, we want a list of the names of the Cats. Using the var keyword, we instruct the compiler that we want the variable type defined depending on what the result is that was assigned to the right of the operator’s equal sign.

Using Lambda Expressions with Anonymous Types

This is an example of how lambda expressions are used with anonymous types:

```
using System;
using System.Collections.Generic;
using System.Linq;
class Cat
{
  public string Name { get; set; }
  public int Age { get; set; }
}
class demo{
  static void Main()
  {
    List<Cat> Cats = new List<Cat>() {
      new Cat { Name = "Molly", Age = 4 },
      new Cat { Name = "Simon", Age = 0 },
      new Cat { Name = "Serendipity", Age = 3 }
    };
    var newCatsList = Cats.Select(x => new { Age = x.Age, FirstLette
r = x.Name[0] });
```

```
      foreach (var item in newCatsList)
      {
         Console.WriteLine(item);
      }
      Console.Read();
   }
}
```

The output is:

```
{ Age = 4, FirstLetter = M }
{ Age = 0, FirstLetter = S }
{ Age = 3, FirstLetter = S }
```

In the newCatsList collection that was created, there are elements of the anonymous type that take parameters of the Age property and the FirstLetter property.

Sorting

This is how to use a lambda expression for sorting:

```
var sortedCats = Cats.OrderByDescending(x => x.Age);
foreach (var Cat in sortedCats)
{
   Console.WriteLine(string.Format("Cat {0} is {1} years old.", Cat.Na
me, Cat.Age));
}
```

The output is:

```
Cat Molly is 4 years old.
Cat Serendipity is 3 years old.
Cat Simon is 0 years old.
```

Anonymous Methods

Anonymous methods were first introduced with v2.0 of C# and they are code blocks that are used with delegates parameters. You can use the anonymous method wherever you want and, if you remember back a few sections, the delegate is defined in the line – there is no method name but there is a method body and optional parameters. The parameter scope in an anonymous method is the anonymous code block for the method. Lastly, you can use generic parameter types with the anonymous method, just like you would with any other method.

Let’s see if we can make this clearer.

For a piece of client code to handle an event, a method should be defined and that method should match the associated delegate’s signature. We call this an event handler, for example:

```
Predicate<Employee> empPredicate = new Predicate<Employee>
(FindEmp);

Employee empl = listEmp,Find (emp => FindEmp(emp));


2 references

Private static bool FindEmp(Employee emp)

{

        Return emp.ID = = 05;

}
```

These are two separate pieces of one piece of code. What we have done is placed an overhead coding in the instantiation of delegates because we need a separate method. It is worth noting that we only call the event handlers from the part of the program that invokes the delegate, and nowhere else.

You can associate delegates straight to statement block at the time the event is registered. To delegate the code block, we use the delegate keyword and it is called an inline delegate or an anonymous method.

Because an anonymous method has no name, it is an easy way to create instances of delegates without the need to write a separate function or method as you can see from the brief example above. Using an anonymous method, we could rewrite the first part of that example like this:

```
delegate (Employee emp)

{

        Return emp.ID = = 02;

}
```

As you can see, we have replaced the FindEmp() function from the first example with a delegate.

Lambda Expressions

As with the anonymous methods, it is better to write the blocks of code inline and this requires the use of delegates. Here's an example:

```
int res = list.Find(delegate(int n)

{

   if (n == 5) return true;

   else return false;

});
```

You can see from this example that writing the syntax for the anonymous method is a little bit harder and isn't all that easy to manage.

In v3.0 of C#, we saw the lambda expression. This gives us a much easier and more concise syntax, more functional and better to use to write those anonymous methods.

Lambda expressions are written as lists of parameters, with the lambda operator (=>) following the list. After this is an expression or a block of statements. Here's an example of a lambda expression:

ArgumentsToProcess => StatementsToProcess

C# has two types of lambda expression:

- Expression Lambda – has a single expression, no return statement and no curly braces
- Statement Lambda – a collection of multiple statements

We can rewrite the example that used the anonymous methods using the expression lambda. In the code below, the Find() method from the List<T> generic class will take a lambda expression to search and the search will be based on a specified criterion, for example, if 4 found in the list.

```
Static void Main(string[] args)
{
        List<int> list = new List<int>();
        list.Add(1);
        list.Add(2);
          list.Add(3);
          list.Add(4);
          list.Add(5);
          int res = list.Find(n => n == 3);
          if (res==0)
          {
                  Console.WriteLine(“item not found in this list”);
          }
          else
          {
                  Console.WriteLine(“item is in the list”);
          }
          res = list.Find(n => n == 6);
        if (res == 0)
        {
                Console.WriteLine("item not found in this list");
        }
        else
        {
                Console.WriteLine("item is found in the list");
        }
        Console.ReadKey();
}
```

And the output would be:

```
item is in the list
item is not found in this list
```

If we needed several code lines processed by the expression, we can use the statement lambda. In the following example, the FindAll() method of the List<T> generic class is used and it will return all elements that are a match to the conditions that the specified condition defines:

```
static void Main(string[] args)
{
        List<int> list = new List<int>();
        list.Add(1);
        list.Add(12);
        list.Add(3);
        list.Add(4);
        list.Add(15);
        List<int> lst=list.FindAll(n=>
        {
        if (n <= 5)
        {
        return true;
        }
        else
        return false;
          });
        Console.WriteLine("Value of n<=5 is :");
        foreach (int item in lst)
        {
        Console.WriteLine(item);
        }
        Console.ReadKey();
}
```

And the output of this would be:

```
Value of n<=5 is:
1
3
4
```

Asynchronous Programming

With C# v5.0, there is yet another new feature called asynchronous programming that lets you write asynchronous code. Let's just say that you are writing an application based on a Windows form. You click on a button that downloads a web image synchronously. Now, it can take upwards of 30 seconds to download that image which might sound a long time, but while the download is happening, your application has frozen. From the perspective of a user, that is not good news – the vast majority of users will move away from an unresponsive application. It makes sense to have the image downloading in an asynchronous manner.

By the end of this section, you will have a much better understanding of what this all means in C#, and more importantly, how you can use it when you write your own programs.

In this chapter, we will explain what "asynchronously" means in C#, and how we can use this feature in our applications. The issue we mentioned here, your application freezing when an image is being downloaded, can be avoided very easily and it's done with the use of two more of those incredibly useful keywords – async and await.

Async

When you declare a function and use the async keyword in front of it, that function immediately becomes an asynchronous function. When you make use of this keyword, you have access to the resources that the .NET framework provides for creating asynchronous frameworks and that function is called asynchronously. The syntax for async looks like this:

```
public async void MyProcess()
            {}
```

That function can now be called asynchronously.

Await

The async keyword informs the compiler that we have an asynchronous function and that function must also have another keyword in it – await. The syntax for this is

```
public async void MyProcess()
{
// do the work asynchronously
 await Task.delay(5);
}
```

The above method does the work requested of it but it will do it after a specified delay, in this case, 5 seconds.

Back to the issue we talked about at the start of the section. By using the code below, your web image will download synchronously:

```
 private void button_Click(object sender, EventArgs e)
{
   WebClient image = new WebClient();
   byte[] imageData = image.DownloadData(“http://urlOfTheImage”);
   this.imageView.Image = Image.FromStream(new
MemoryStream(imageData));
}
```

While the image is downloading, your web application freezes and becomes unresponsive, at least while the code is being executed. Now, by using the next piece of code, which includes those two new keywords, we can ensure that the entire operation happens asynchronously:

```
private async void button_Click(object sender, EventArgs e)
{
   WebClient image = new WebClient();
```

byte[] imageData = await image.DownloadDataTaskAsync("http://urlOfTheImage");

this.imageView.Image = Image.FromStream(new MemoryStream(imageData));

}

At first glance, this looks much the same as our first example, but there are three differences in it:

- We used the async keyword in our method.
- The call that requests the image download has been preceded by the await Keyword.
- The method called DownloadData has been swapped for DownloadDataTaskAsync, which is the asynchronous equivalent.

The method called DownloadData belongs to the WebClient class and it does two things – synchronously downloads data and then puts control straight back to the caller and this is what makes the program turn unresponsive. The method called DownloadDataTaskAsync, on the other hand, immediately returns control to the caller and downloads the image asynchronously.

The really interesting thing here is the await keyword – unless the download is finished, that keyword will release the UI thread. Whenever the program comes up against the await keyword, the function will return and will not resume until the specific operation has been completed. When it does resume, it starts again from where it was stopped.

All asynchronous methods can return these three value types:

- **Void –** it will return nothing and it performs one single operation:

Task<T>: this returns a task that has a parameter of T type. In general, a Task is void and will not return a value. However, a

Task<int> will return an int type element – this is known as a generic type.

And, if you do not put the await keyword into an async method, it will still run synchronously.

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

class Program
{
  static void Main()
  {

          Task task = new Task(ProcessDataAsync);
          task.Start();
          task.Wait();
          Console.ReadLine();
  }

  static async void ProcessDataAsync()
  {

          Task<int> task = HandleFileAsync("C:\\enable1.txt");


          Console.WriteLine("Please be patient " +
              "while I do something that is important.");


          int x = await task;
          Console.WriteLine("Count: " + x);
  }

  static async Task<int> HandleFileAsync(string file)
  {
          Console.WriteLine("HandleFile enter");
          int count = 0;


          using (StreamReader reader = new StreamReader(file))
          {
              string v = await reader.ReadToEndAsync();


              count += v.Length;


              for (int i = 0; i < 10000; i++)
              {
                  int x = v.GetHashCode();
                  if (x == 0)
                  {
                      count--;
                  }
              }
          }
          Console.WriteLine(“HandleFile exit”);
          return count;
  }
}
```

Output:

```
HandleFile enter
Please be patient while I do something that is important.
HandleFile exit
Count: <number_of_characters>
```

In this example, we have our Main method and after that, we create a Task instance. We pass in an argument (parameter) of ProcessDataAsync and then, in the following line, we use task.Start() to start the task. We wait for it to end using the task.Wait() method.

ProcessDataAsync is, as you should have guessed, an asynchronous method and that means it is mandatory to use the await keyword. In the method Task<int> task = HandleFileAsync(‘C:\\enable1.txt’), the first code line is calling a method called HandleFileAsync. Because that method is an asynchronous method, the control will return to this method before HandleFileAsync returns.

While HandleFileAsync is doing its job, a message is displayed on the screen that reads “HandleFile enter” and “Please be patient while I do something that is important”. The first line of that message is from the HandleFileAsync method and this happens because the method is printed to the screen when it is called and control is passed straight back to the ProcessDataAsync method. Then the second line of the message is printed on the screen.

With the await task line of the ProcessDataAsync method, we are telling it to wait for HandleFile to finish before the result is assigned to the variable called x and printed to screen.

The HandleFileAsync method is also an asynchronous method and also has the mandatory await keyword in it. Once the first line has printed to screen, the variable called count, which is a dummy integer variable, is initialized and set as equal to 0 – we do this because a file location was passed in the argument when the method was called – C:\\enable1.txt. To read the file data, the reader needs to be initialized. It's called StreamReader and it is passed to the file location. The asynchronous reader.ReadToEndAsync() method, which is a built-in method, is used to read the file and the await keyword is used to tell it that it has to wait until it has read the file. The result is assigned to a variable called v, which is of the string type.

Lastly, the total string length is added to the dummy count variable and some dummy code is added to count that value. That code is purely to help you understand what is going on; the last thing is, the method returns and a simple line that reads HandleFile exit is printed to screen along with the dummy value.

And that, folks, is how to use asynchronous methods and functions in your C# code.

Next, we move on to an in-depth look at LINQ.

LINQ

LINQ stands for Language Integrated Query and those who already have some familiarity with databases already know just what a query is. For those who don't, a query is nothing more than a way for a programmer to interact with a database and manipulate the data. A programmer accesses a table in a database using a query so that they can insert data, delete it, edit what's there, or retrieve it. However, unlike that usual way that we query data, C# provides developers with an entirely new way of accessing multiple data types and working with them and that includes databases, XML files, dynamic data, and Lists. That way is using LINQ.

There are two basic units in a LINQ function – elements and sequences. LINQ sequences are sets of items that implement the interface called IEnumerable<T>. Each of the items in a set are known as elements and an example of a sequence that implements this particular interface is the array. Look at this:

```
  string[] cities = {"Berlin", "New York ", "Sofia", "Istanbul", "London" };
```

This array is of the string type and it has the names of several different cities. We call this collection type a 'local' sequence because all of the items are stored in the system's local memory.

Query Operators

A LINQ query operator is used to take LINQ sequences as input, transform the entire sequence, and then return the new sequence as the output. In the System.Linq namespace is the class call Enumerable and that contains roughly 40 of these query operators and we're going to be looking at the ones you will use the most.

Where Operator

We use this operator to filter sequences based on a specified condition. For example, if you wanted to retrieve the names of every city in the array called cities, where the city length is greater than or equal to 6, you would do it like this:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{
    class Program
    {

        public static void Main()
        {

            string[] cities = {"Berlin", "New York ", "Sofia", "Istanbul",
"London" };

            List<string> citiesendingwiths = (from c in cities
                                              where c.Length >=6
                                              select c).ToList();

            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }

            Console.Read();
        }
    }
}
```

A LINQ query sequence is much like an SQL query. In the above example, all we did was retrieve every city whose name length was equal to or greater than 6. The result would be displayed on the screen and, if you ran this, you would see all the cities except for Sofia.

The syntax we used for LINQ in this example is known as query syntax because it is similar to the syntax for the SQL query. This isn’t the only way for LINQ queries to be executed – we can also do it with lambda expressions in a way that is called fluent syntax. It works to provide the exact same functionality that the first example gave is in this next example:

```
using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace MyCSharpApplication

{

  class Program

  {


    public static void Main()

    {


      string[] cities = {"Berlin", "New York ", "Sofia", "Istanbul",
"London" };
```

```
            List<string> citiesendingwiths = cities.Where(c => c.Length
>= 6).ToList();


            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }


            Console.Read();
        }
    }


}
```

As with SQL, we can also use the logical operators together with the comparison operators in LINQ. So, if you wanted all city names with a letter ‘o’ in them and with a length that is greater than 5, you would use the AND logical operator, like this:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{
    class Program
    {


        public static void Main()
        {

            string[] cities = {"Berlin", "New York ", "Sofia", "Istanbul",
"London" };

            List<string> citiesendingwiths = cities.Where(c => c.Length
>= 6 && c.Contains("o")).ToList();

            foreach(string c in citiesendingwiths)
            {
                Console.WriteLine(c);
            }

            Console.Read();
        }
    }

}
```

This time, you will only see the cities of Sofia and New York on the screen as these are the only ones with the letter ‘o’ and greater than or equal to 6.

Select Query Operator

Select queries are used mostly with objects that contain several members. It can be used for the retrieval of a specified member in all collection objects and it can be used by any entire object. The basic usage of this query is in the next example:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
namespace MyCSharpApplication
{

    public class Person
    {

        public int age;
        public string name;

        public Person(int age, string name)
        {
            this.age = age;
            this.name = name;
        }

    }

    class Program
    {

        public static void Main()
        {

            Person p1 = new Person(9, "Jane");
}

            Person p2 = new Person(8, "Jack");
            Person p3 = new Person(13, "Mike");
            Person p4 = new Person(15, "Evan");
            Person p5 = new Person(6, "Rupert");

            List<Person> plist = new List<Person>();
            plist.Add(p1);
            plist.Add(p2);
            plist.Add(p3);
            plist.Add(p4);
            plist.Add(p5);

            List<string> personnames = plist.Where(p => p.age <=
10).Select(per => per.name).ToList();

            foreach(string pname in personnames)
            {
               Console.WriteLine(pname);
            }

            Console.Read();
        }
    }

}
```

What we have is a newly created class called Person. It contains two member variables called age and name. We can use a constructor to initialize the variables. In our Main method, five objects were created of this class and stored in a list collection called plist. Then a LINQ query was executed using both the where and the select operators. We use the where operator to filter the objects with a lower age than 10 – if we omit the select operator, we would see the entire Person object returned. Because we only want the names of those who are younger than 10, the select operator is used. We pass the operator lambda expression which will select the name of the person.

Those are two of the LINQ query operators that you will use the most in your programs. I really could continue, but we don’t have the space or time to discuss every operator – the entire subject of LINQ would take a book of its own – so we’ll move on and look at how to connect LINQ to both SQL and to XML.

What really gives LINQ the edge over other similar technologies is the flexibility it has to work with several data types. What that means is that LINQ uses the same query syntax to handle incoming data independently of the source type for the data, where the other technologies require you to write a different query for each separate source. For example, you would need an SQL query if you wanted to interact with the SQL server and to interact with the XML data type, you would need an XQuery. What we are going to look at here is the relationship between LINQ and SQL, and LINQ and Lambda expressions.

LINQ and SQL

Using LINQ is the smart choice as an alternative to using the old-fashioned way of SQL queries to access SQL data. Where you see the term LINQ to SQL, it is usually describing a relationship by which we can use LINQ to access the SQL databases. The first thing we need to do is map the target or existing SQL database to LINQ.

Mapping LINQ to SQL

This is referring to .NET being able to take the database that already exists and recognize it as an Object or Class. This is easy enough; just open Visual Studio and go to Solution Explorer. Click Target Project and then click on Add - > New Item. You will see an option for Categories; click on Data - >LINQ to SQL Classes (this will be on the left under Templates.

The result of this will be .dbml file with a GUI (Graphical User Interface) and the GUI will have two separate parts. The first lets you do drag and drop operations on tables as a way of auto-creating classes from the tables. The second lets you drop your stored procedures. Overall, you can select all the essential tables, drag them and drop them as you need to.

Selecting Data

Once the .dbml file has been created, a DataContext file that corresponds to it is created, all by itself, by the .NET framework responsible for database communication. Then, LINQ queries use the class objects for working with the databases. We can see how this works from the next example:

```
public bool checkValidUser(string Name, string passcode)

{

DBToysDataContext sampleDB = new DBToysDataContext();

var getter = from u in sampleDB.Users

                    where u.Username == Name

                    && u.Password == passcode

                    select u;

return Enumerable.Count(getter) > 0;

}
```

Before we look at what this does, there is one important thing to know – when the DBToys.dbml file was mapped, a class file called DBToysDataContext was automatically created. In the example, we passed two strings to the function called checkValidUser, and they were called Name and Passcode. The function is used to validate the name and password that a user enters against a table called Users that we find in the database called Toys. The first line in the function is instantiating an object called sampleDB so that it uses the class file called DBToysDataContext to access the Toys database.

The 'u' in the line that reads 'from u in sampleDBUsers' is treated by Visual Studio as a User's class object which refers to the Users table in the Toys database. Next, we pass the incoming values for column and field as a var object type, which refers to dynamic data. No matter what data types are supplied by the LINQ query, the var type can be stored in a variable called getter. All we are doing here is retrieving the username and the password from the Users table and saving it. Last, the Enumerable.Count function is returning to the number of data rows that the LINQ query returns.

All that said, there is another way of accessing the exact same data in LINQ but without having to use the SQL-like syntax:

```
public bool checkValidUser(string Name, string passcode)

{

      DBToysDataContext sampleDB = new DBToysDataContext();

      List<Users> getter = sampleDB.Users.Where(u => u.Username
== Name && u.Password==passcode);

      if(users.Count>0)

      {

            return true;

      }

      return false;

}
```

That is much nicer. Rather than using the SQL syntax, we use the method called 'where' that we discussed earlier. This will access the

Toys database table called Users directly and the rest of the process remains as it was – the only exception is that the data type of List is used for returning the values that come back from our LINQ query.

Here’s another example:

```
   public User bring User(string name)
{
    DBToysDataContext sampleDB = new DBToysDataContext();
    User use = sampleDB.Users.Single(u, u.UserName=>name);
    return use;
}
```

It is also easy to retrieve one row (object) and send it using the LINQ query. The function called bringUser accepts a single argument of the name we want to be matched to an object in the Users table. In the line that reads ‘sampleDB.Users.Single(), the method called single will look for a match to the specified name and returns the one row or object as soon as a successful match is made.

LINQ and Lambda Expressions

We talked about lambda expressions before, and where LINQ is concerned, queries generally imply these expressions when we deal with data lists or collections to filter them as per a specified condition.

We can see how that is done in this example:

```
IEnumerable <SelectListItem> toys = database.Toys
        .Where(toy => toy.Tag == curTag.ID)
        .Select(toy => new SelectListItem { Value = toy.Name, Text =
toy.ID });


 ViewBag.toySelector = toys;
```

In this code, a variable called toys has been declared. We cast it to the SelectListItem type so that the resulting row is saved from our LINQ query. We have two methods, called Where and Select, that we use to find the target that matched with our specific query. The line that reads ‘toy => toy.Tag == curTag.ID’ will choose a toy as per a tag that gets passed but the line that reads ‘toy => new SelectListItem {Value = toy.Name, Text = toy.ID’ will choose the toy based on an ID and name that are passed. The result, the toy that matches the specified criterion, is saved to the variable called toys.

Now we turn our attention to the way that LINQ works with XML data. XML stands for Extensible Markup Language and it is one of the most used languages on the internet. But it is far more than a set of static labels that are text-based. XML is also sometimes known as self-defining data or self-describing data. Its tag usage is customizable to the highest extent as well as being standardized; it is used globally for sharing data, for accessing it and manipulating it. What we will look at now is how to use LINQ to retrieve XML data, to delete it, and how to insert and update it.

Retrieve and delete XML data

Every XML data file has a parent or root tag, also known as an element, that will both encapsulate and define the type of its child records and define the attributes for them. The child records or elements that result from this have real data in them that can be manipulated as needed. Look at this example code; this is XML code that we will use throughout the rest of this section for LINQ queries:

```
<?xml version=‘1.0’ encoding=‘utf-8’?>

<Toys>

  <Toy ID=‘1’>

    <Name>Millie</Name>

    <Price>$0.45</Price>

  </Toy>

  <Toy ID=‘2’>
```

```
    <Name>Dove</Name>

    <Price>$0.40</Price>

  </Toy>

  <Toy ID='3'>

    <Name>Rook</Name>

    <Price>$0.55</Price>

  </Toy>

</Toys>
```

In this example, the root element is ‘Toys’ and it has several child elements, each a ‘Toy’ and each of these has its own attribute called ID with two inner elements called Price and Name.

What we want to do is retrieve the child toys and we do that by using a LINQ query, like this:

```
private string file = 'SampleData.xml';


        private void GetData()

        {

            try

            {
                XDocument doc = XDocument.Load(file);
                var comingToys = from toy in doc.Descendants("Toy")
                select new
               {
              ID= Convert.ToInt32(toy.Attribute("ID").Value),
              Name = toy.Element("Name").Value ,
         Price = toy.Element("Price").Value
               };

              foreach (var x in comingToys)
```

```
     {
           Console.Write("Toy ID " + x.ID + " ");
  Console.Write("Toy Name " + x.Name + " ");
  Console.WriteLine("Toy Price " + x.Price+ " ");
    }

    }
    catch (Exception err)
    {
       Console.WriteLine(err.Message);
    }

  }
```

The first line in the code is pointing to SampleData.xml, which is the XML sample data file. In the function, we use the function called Load() to load the xml file, and once this is done, child ‘toy’ elements are retrieved by using the function called Doc.descendents(). After that, each of the ‘toy’ child elements is selected and we retrieve the ID along with the two inner elements. These are then passed to comingToys, which is a dynamic variable. At the end, a foreach loop is used to display the data we retrieved.

The output will be:

```
Toy ID 1 Toy Name Millie Toy Price $0.45

Toy ID 1 Toy Name Dove Toy Price $0.40

Toy ID 1 Toy Name Rook Toy Price $0.55
```

In much the same way, if we wanted a child element removed from the XML file, we would do it like this:

```
private string file = ‘SampleData.xml’;


private void DeleteData (int id)

{

        try
        {
        XDocument sampleXML = XDocument.Load(file);
        IEnumerable<XElement> cToy =
sampleXML.Descendants("Toy").Where(c => (int)c.Attribute("ID") ==
id);
        cToy.First().Remove();
        sampleXML.Save(file);
        }
        catch (Exception e)
        {
        Console.WriteLine(e.Message);
        }
}
```

Once the target file is loaded, the code above will traverse through the child ‘toys’ to find a match to the toy ID we passed in. Once found, the match is removed using the function called Remove() and the file is saved.

Insert and Update XML data

It isn’t a much different route to take for adding data to an XML file. In essence, all we really need is an object that is of XElement type that has a matching signature to the elements that already exist in the target file. Then that object is inserted in the data file by using an object of XDocument type:

```
private string file = ‘SampleData.xml’;


private static void InsertData(string name, string price)

{
        try
        {
        XDocument doc = XDocument.Load(file);
        XElement newToy = new XElement("Toy",  new
XElement("Name", name), new XElement("Price", price));
        var lastToy = doc.Descendants("Toy").Last();
        int newID = Convert.ToInt32 (lastToy.Attribute ("ID").Value);
        newToy.SetAttributeValue("ID",++newID);
        doc.Element ("Toys").Add (newToy);
        doc.Save (file);
        }
        catch (Exception err)
        {
        Console.WriteLine(err.Message);
        }
}
```

Once the object of XDocument type is created we also created an object of XElement type whose signature matches that to the Toy elements that already exist. This is done using the line that reads XElement newToy = new XElement(‘Toy”, new XElement(‘Name’, name), new XElement(‘Price’, price)’". Then we use a function called doc.Descendents(‘Toy’).Last() to get the final toy element from the variable called lastToy.

When we have that element, we retrieve the value for the ID and increment it at the same time as setting the attribute to the new element, using the line that reads newToy.SetAttributeValue(‘ID’,++newID)”. Last, the toy object is inserted using the function called Add() and the changes to the file are saved.

In our sample file, we now have another child. If, for example, we had passed in a name of ‘John ‘ with a price of $0.1 to the function above, it would look like this:

```
<Toy ID=‘4’>
   <Name>Hare</Name>
   <Price>$0.65</Price>
 </Toy>
```

To change XML data or elements that already exist, the first thing we need to do is find the target element against certain criteria with data values that we want to change, which is the line that reads XElement cToy = doc.Descendants(‘Toy’). In this line, (c=>c.Attribute(‘ID’).Value.Equals(id)” does that by looking for a match to the ID we provided and letting cToy, which is the XElement object, point to the element. As soon as we have the reference to the element, the values of its inner elements can be updated using cToy.Element(‘Price’).Value = price. And, finally, the XML document is saved with all the changes.

```
private string file = ‘SampleData.xml’;

private static void UpdateData(string name, string price, int id)

{
        try
        {
        XDocument doc = XDocument.Load(file);
        IEnumerable<XElement> cToy =
doc.Descendants("Toy").Where (c=>(int)c.Attribute("ID") == id);
        cToy.First().Element("Name").Value = name;
        cToy.First().Element("Price").Value = price;
        doc.Save(file);
        }
        catch (Exception err)
        {
        Console.WriteLine(err.Message);
        }
}
```

Now let’s assume that we passed that function above three more values as arguments – id=1, name=Alligator, and price=$0.75. Once the above function has been executed the XML data file will no longer show toy name=Millie at id=1. Instead, name=Alligator will be there instead:

```
 <Toy ID=‘1’>

   <Name>Alligator</Name>

   <Price>$0.75</Price>
```

```
  </Toy>
```

Next, we move on to PLINQ and Parallel Classes.

Parallel Class and PLINQ

PLINQ or Parallel LINQ is nothing more than a parallel implementation of a LINQ pattern. PLINQ queries are very much like standard LINQ queries to Objects. And like a standard LINQ query, the PLINQ query operates on any in-memory IEnumerable or any IEnumerable<T> data source. Their execution is deferred, meaning that execution does not start until the specified query has been enumerated.

The main difference is PLINQ tries to make use of every processor on a system and this is done by partitioning each data source into segments; the data query is then executed on each of the segments on a separate worker thread on multiple processors and in parallel. In some cases, using PLINQ will ensure that the query is much faster.

Using parallel execution, PLINQ can provide some serious improvements to performance against the legacy code for some query types and that can often be achieved simply by adding a query called AsParallel to the data source. However, with some query types, using parallelism will only slow them down so it is important that you understand PLINQ.

We will be using Lambda expressions to define delegates and we are going to go over PLINQ classes, and h0w PLINQ queries are created. What you won’t get is an in-depth look at everything, but more of an overview with examples.

The ParallelEnumerable Class

Most of the PLINQ functionality is exposed in the class called System.Linq.ParallelEnumerable. Both this and the other namespace types for System.Linq are compiled into the assembly called System.Core.dll and the default projects in Visual Studio (C# and Visual Basic) will all referencing the assembly and they will import the relevant namespace.

With ParallelEnumerable, you get the implementations of every standard query operator that is supported by LINQ to Object but what it won’t do is try to parallelize each of them. As well as those query operators, you will also find some methods in the ParallelEnumerable that enable certain behaviors that are specific only to parallel execution. You can see what these methods are below:

ParallelEnumerable Operator	Description
el	This is the PLINQ entry point and it is used to specify parallelization of the remainder of the query if it can be done.
ntial	This is used to specify that the remainder of the query should be run as a non-parallel LINQ query and sequentially.
ed	This is used to specify that the source sequence ordering should be preserved by PLINQ for the remainder of the query or until there is a change to the ordering, for example, if an orderBy clause is used.
ered	This is used to specify that there is no requirement for the source sequence ordering to be preserved by PLINQ for the remainder of the query.
cellation	This is used to specify that the state of the cancellation token provided should be monitored periodically by PLINQ and, if requested, execution should be canceled.

reeOfParallelism	This is used to specify the maximum number of processors that should be used by PLINQ for parallelization of the query.
geOptions	This is used to give a hint to the way the parallel results should be merged by PLINQ, if it is possible, into a single sequence on the consuming thread
cutionMode	This is used to specify if the query should be parallelized by PLINQ even if the default behavior is that it should be sequentially run.
	This is a multi-threaded method of enumeration that enables the parallel processing of the results without merging back to the consumer thread first, not like iteration over the query results.
te Overload	This is an overload method unique to PLINQ that enables the intermediate aggregation over partitions that are thread-local. It also includes a function for final aggregation, combining all the partition results.

The Opt-in Model

You can opt into PLINQ when you write queries simply by invoking the extension method called ParallelEnumerable.AsParallel on the data source, as you can see in this example:

```
var source = Enumerable.Range(1, 10000);

// Use AsParallel to opt into PLINQ.
var evenNums = from num in source.AsParallel()
            where num % 2 == 0
            select num;
Console.WriteLine("{0} even numbers out of {1} total",
              evenNums.Count(), source.Count());
// This example will give us an output of:
//      5000 even numbers out of 10000 total
```

This extension method will bind the query operators that follow, the where and select operators in this case, to the implementations of System.Linq.ParallelEnumerable.

Execution Modes

PLINQ is conservative by default. The PLINQ infrastructure will analyze the query structure at run time and, if the query is likely to be sped up by the use of parallelization, then PLINQ will split the source sequence into separate tasks that can be run synchronously or simultaneously. If parallelization is not safe to do, the query is run by PLINQ sequentially.

If there is a choice between using a parallel algorithm that is potentially expensive or a cheaper sequential algorithm, PLINQ will, by default, choose the sequential algorithm. If you want PLINQ to opt for the parallel algorithm instead, you can use both the method called WithExecutionMode and the System.Linq.ParallelExecutionMode enumeration. This makes it much easier when you already know that a query will execute much faster in parallel because you have tested and measured it.

Degree of Parallelism

Also by default, PLINQ will make use of every processor on the host system. Again, you can tell PLINQ that you only want a specific number of processors used and you do this by using the method called WithDegreeOfParallelism. This is another useful method to use when you need to ensure that all of the other processes on the system will still get a specified amount of CPU time. The following code shows you how to use the method to limit the query to using no more than two processors:

```
var query = from item in
source.AsParallel().WithDegreeOfParallelism(2)

            where Compute(item) > 42

            select item;
```

Where a query may be doing a lot of work that is classed as non-compute-bound, like File I/O, it may be more advantageous to specify that the degree of parallelism should be more than the number of processor cores on the system.

Ordered vs. Unordered Parallel Queries

In certain queries, in order for results to be produced that keep the source sequence ordering preserved, a specific query operator must be used. With PLINQ, you get the AsOrdered operator and this is not the same as the AsSequential operator. The AsOrdered sequence still is processed in parallel but the results will be buffered and then sorted. Preserving errors is time-consuming because of the extra work involved, and because of this, AsOrdered sequences may be slower in processing than an AsUnordered sequence, which is the default option. There are several factors that determine whether a parallel operation that is ordered is going to be faster than the sequential version.

In the example below, you can see how you can opt into order preservation:

```
var evenNums = from num in numbers.AsParallel().AsOrdered()

             where num % 2 == 0
```

```
            select num;
```

Parallel vs. Sequential Queries

There are operations that require the source date is delivered in a sequential way. When required, the ParallelEnumerable query operators will revert automatically to sequential mode. For those user delegates and user-defined query operators that need sequential execution, the AsSequential method is built-in to PLINQ. When this method is used, all of the other operators inside the query will be sequentially executed until such time as AsParallel is called again.

The ForAll Operator

In a sequential LINQ query, the execution is put off until the query has been enumerated using a foreach method or through invocation of a method like ToDictionary, ToArray, or ToList. With PLINQ, foreach can also be used for query execution and for iteration through the results. However, foreach will not run in parallel, and as such, the output from the parallel tasks are required to be merged back to the thread the loop ran on. With PLINQ, foreach can also be used for preservation of the final ordering of the results and when the results are processed in a serial way, i.e. when Console.WriteLine is used on each element. When the order does not need to be preserved and when result processing can be parallelized, the ForAll method can be used for faster execution of the PLINQ query. The final merge step is not performed by ForAll. In the following example, you can see how the ForAll method is used. We use System.Collections.Concurrent.ConcurrentBag<T> because it has full optimization for multiple threads to be concurrently added without trying to remove items.

```
var nums = Enumerable.Range(10, 10000);

var query = from num in nums.AsParallel()

          where num % 10 == 0

          select num;
```

```
// The  results are processed as each of the threads completes
// and they are added to a
System.Collections.Concurrent.ConcurrentBag(Of Int)
// because this can take concurrent add operations in safety
query.ForAll(e => concurrentBag.Add(Compute(e)));
```

Cancellation

PLINQ has full integration with .NET Framework 4 cancellation types. Because of that, we can cancel a PLINQ query, unlike the sequential LINQ to Object query. If you want to create a PLINQ that can be canceled, the WithCancellation operator should be used on the query and a CancellationToken instance used as the argument. When the IsCancellationRequested token property is set as true, it will be noticed by PLINQ which will then cease processing on the threads and an OperationCancelledException will be thrown.

Exceptions

When a PLINQ query is executed, exceptions might be thrown from different threads all at the same time. Plus, the code needed to handle the exception may be on a separate thread than the code that caused the exception to be thrown. The AggregateException type is used by PLINQ for encapsulating every exception a query throws and sends the exceptions right back to the calling thread. On this thread, we only need a single try-catch block, but it is possible to iterate through every encapsulated exception in AggregateException; any that can be recovered from safely will be caught. There are rare cases where exceptions can be thrown that aren’t wrapped up in the AggregateException; ThreadAbortExceptions will also not be wrapped.

When an exception is allowed back up to the joining thread, the query may continue processing a few items after the exception is raised.

Custom Partitioners

There will be cases where the query performance can be improved and that is done by writing a custom partitioner that will take full advantage of a source data characteristic. Inside the query, the enumerable object that is queried is the custom partitioner:

```
int[] arr = new int[9999];
Partitioner<int> partitioner = new MyArrayPartitioner<int>(arr);
var query = partitioner.AsParallel().Select(x => SomeFunction(x));
```

PLINQ has support for a set number of partitions, although, during run time, there is the chance that the data can be dynamically assigned to the partitions for the sake of load balancing. The for and foreach statements only support dynamic partitioning and this means that, at run time, the number of partitions will change.

Measuring PLINQ Performance

Sometimes we can parallelize a query but there are costs to this – setting up parallel queries has overheads that far outweigh what benefit is gained in terms of performance. If a query doesn't need to do a lot of computation or if there is a small data source, PLINQ queries can still be slower than sequential LINQ to Object queries. The Parallel Performance Analyzer can be used in Visual Studio Team Server to find any bottlenecks in processing, to compare how various queries perform, and to work out whether sequential or parallel processing is being used.

Understanding PLINQ Speedup

The primary purpose of PLINQ is to speed up LINQ to Object query execution. It does this through parallel execution of query delegates on multicore systems. PLINQ works the best when elements in source collections are processed independently of one another and there is no shared state among the individual delegates.

Operations like this are common in both PLINQ and LINQ to Object queries and these operations are sometimes called “delightfully parallel” and this is because they easily lend themselves to being scheduled on several threads. However, not every query has these ‘delightfully parallel’ operations; most of the time, queries involve certain operators that will slow parallel execution down or simply can’t be parallelized at all. Even if a query is ‘delightfully parallel’, the data source still has to be partitioned by PLINQ and work still has to be scheduled on the threads. This usually ends in the results being merged on completion of the query.

All of these kinds of operations cause the cost of parallelization to rise in terms of computation and these costs are known as overheads. For the best performance to be achieved in PLINQ queries, the ‘delightfully parallel’ parts need to be maximized and the parts that need the overhead should be minimized. To finish this section on PLINQ, we are going to look at writing PLINQ queries that are as efficient as they can be while still providing the right results.

What Can Impact PLINQ Query Performance?

What we are going to look at now are some of the very important factors that can have an impact on the performance of parallel queries. These are general and, on their own, are simply not enough for a prediction on the performance of a query in all cases. As with everything, you should measure the real performance of a query on computers that offer a series of representative loads and configurations.

- **Computational cost – overall work**

To be speeded up, PLINQ queries need to have sufficient ‘delightfully parallel’ work in order for the overhead to be offset. That work can be expressed in terms of the computational cost of each individual delegate multiplied by how many elements there are in the source collections. Assuming that parallelization is possible on an operation then the more expensive it is in computational terms and that means there is more opportunity for it to be sped up.

For example, let’s say that a specific function will execute in one millisecond. If you have a sequential query that goes over a thousand elements, the operation will take one second to perform. On the other hand, a parallel query done on a computer that has four cores could be done inside of 250 milliseconds. The speedup gained here is 750 milliseconds. If the elements in the function needed one second each to execute, the speedup would equate to 750 seconds. If the delegate works out to be significantly expensive, PLINQ could offer sufficient speedup when the source collection only has a few items. On the other hand, collections with little source data and trivial delegates generally don’t make very good PLINQ candidates.

In the example below, you can see that queryA is a good PLINQ candidate assuming that there is a lot of work involved in the Select function. However, queryB probably isn’t a great candidate because the Select statement has insufficient work and the parallelization overhead cost will offset most, if not all, of any speedup gained:

```
var queryA = from num in numberList.AsParallel()
            select ExpensiveFunction(num); //a great candidate for
PLINQ


var queryB = from num in numberList.AsParallel()
            where num % 2 > 0
            select num; //not such a good candidate for PLINQ
```

- **Degree Of Parallelism**

In simple terms, determine how many logical cores the system has. This is an obvious follow-on because a ‘delightfully parallel’ query will clearly run faster on a system that has more cores simply because the work can be broken down into chunks and run concurrently on separate threads. How much speedup is gained will depend on the percentage of the query work is parallelizable. However, what you should not do is assume that every query is going to run at twice the speed on a computer with eight cores than on a computer with four cores. When queries are tuned for the best performance, you must measure the real results on computer systems with varying numbers of cores. In relation to the previous factor, if you have large computing resources, you need large datasets to take full advantage of it.

- **Type and Number of Operations**

The AsOrdered operator is provided by PLINQ when the element order inside the source sequence needs to be maintained. Ordering comes with a cost but this isn’t usually too much. The operations of Join and GroupBy also incur a cost. PLINQ works the best when you let it process the source collection elements in any order it wants to, and passing them to another operator when they are ready.

- **Query Execution Form**

If the results of a query are being stored, usually by calling ToList or ToArray, the results from every parallel thread have to merge into one data structure. To do this incurs an overhead that simply cannot be avoided. In the same way, if the results are iterated over with a foreach loop, the worker thread results must be serialized on to the enumerator thread. However, if you only want an action performed based on each thread result, the ForAll method can be used to do this on multiple threads.

- **Merger Option Types**

There are two ways to configure PLINQ – to buffer its own output and produce it in small chunks or in one go once the entire set of results has been produced, or to stream the results as they come in individually. With the first way, execution time is shorter and, with the second, latency in between the elements that result is reduced. While the merger options don't always impact significantly on the overall performance of the query, they can have an impact on the perceived performance because they are in control of the length a time a user waits before they see the results.

- **Partitioning**

Sometimes, PLINQ queries over source collections that are indexable can result in a workload that isn't very balanced. When this happens, you can create custom partitioners to try to increase the performance of the query.

When Sequential Mode is Chosen

PLINQ always tries to execute queries at a speed that is as fast or faster than sequential execution. PLINQ doesn't look to see the computational cost of the user delegates and it doesn't look to see the size of the input source, but it will look for specific shapes in the queries. More specifically, it will look to see if operators or combinations of operators are in use that tends to slow down query execution when in parallel mode. When it sees them, by default, PLINQ will go back to sequential mode.

However, if you measure the performance of a specific query and determine that parallel mode would be far better and quicker, you can make use of the flag called ParallelExecutionMode.ForceParallelism by using the method called WithExecutionMode, thus telling PLINQ that you want the query run in parallel mode.

The next list describes each of the query shapes that will be executed sequentially by PLINQ:

- Queries with an indexed Where, Select, ElementAt or indexed SelectMany
  clause once the filtering or ordering operator has either rearranged or
  removed altogether the original indices

- Queries with a Skip, SkipWhile, Take or TakeWhile operator and where
  the source sequence indices are no longer in their original order
- Queries with SequenceEquals or Zip, with the exception of any data source
  whose index is in the original order and the other data source is an
  IList(T), array, or another one that can be indexed
- Queries with Concat unless it has been applied to a data source that can be
  indexed
- Queries with Reverse, unless it has been applied to a data source that can
  be indexed

PLINQ Order Preservation

You already know that PLINQ has one main goal – maximum performance while keeping everything correct. Queries should be able to run as fast as they possibly can while producing the right results. Sometimes, the requirement for correct results will require preservation of the source sequence order, but ordering can come at a high computational cost. Therefore, PLINQ, by default, will not preserve that order. In this sense, it is much like a LINQ to SQL query but not like the LINQ to Objects query, where order preservation is required.

You can, however, override the default and enable order preservation and you do that by using the operator called AsOrdered on the source sequence. Then, order preservation can be disabled later with the operator called AsUnordered. On both of these, the query processing will be based on whatever heuristic determines whether parallel or sequential execution should be done.

In the next example, you can see a parallel query, unordered, and filtering for any element that matches a specific condition without attempting to order the results.

```
var cityQuery = (from city in cities.AsParallel()
            where city.Population > 10000
            select city)
             .Take(1000);
```

The result of this query will not necessarily be the first 1000 cities in the sequence that match the condition. Instead, it may be a set of any 1000 cities meeting that condition. The query operators in PLINQ will partition the sequence into several subsequences, all of which are processed concurrently. If there is no specification for order preservation, each partition's results are handed off to the next part of the query in an order that is not necessarily consistent. In addition, one of the partitions may give a result subset before it continued processing the rest of the elements. The order could be different every single time and your application has no way of controlling this because it will all come down to how the threads are scheduled by the operating system.

In the next example, the AsOrdered operator is used on the source sequence to override the default behavior, thus ensuring that the method called Take returns the first 1000 cities that meet the specified condition:

```
var orderedCities = (from city in cities.AsParallel().AsOrdered()
               where city.Population > 10000
               select city)
              .Take(1000);
```

However, this is unlikely to run at the speed of the unordered version. The ordered version has to track the original ordering through each partition, and when it comes to merging, must ensure consistent ordering. As such, it is recommended that AsOrdered is used only when absolutely needed and only on bits of the query that need it. When you don't need order preservation any longer, make sure the AsUnordered operator is used to disable it.

The next example composes a pair of queries to achieve this:

```
var orderedCities2 = (from city in cities.AsParallel().AsOrdered()
                      where city.Population > 10000
                      select city)
                        .Take(1000);



var finalResult = from city in orderedCities2.AsUnordered()
                  join p in people.AsParallel() on city.Name equals
p.CityName into details
                  from c in details
                  select new { Name = city.Name, Pop = city.Population,
Mayor = c.Mayor };

foreach (var city in finalResult) { /*...*/ }
```

Note that, in PLINQ, the order of a sequence is preserved by using operators that impose order for the remainder of the query, In other words, ThenBy, OrderBy, and other similar operators are treated as if an AsOrdered call follows them.

Query Operators and Ordering

The query operators listed below are used to bring order preservation into all subsequent query operations until a call to AsUnordered is made:

- ***OrderBy***
- ***OrderByDescending***
- ***ThenBy***
- ***ThenByDescending***

And these query operators may sometimes require the source sequences to be ordered for the correct results to be produced:

- Reverse
- SequenceEqual
- TakeWhile
- SkipWhile
- Zip

However, other operators behave in a different way depending on whether the source sequence is unordered or ordered. The operators are shown below together with the result on ordered and unordered source sequences:

Operator	**Ordered**	**Unordered**
egate	Nondeterministic where the operation is noncommutative or nonassociative for both ordered and unordered sequences	
	Not Applicable for either ordered or unordered sequences	
	Not applicable for either ordered or unordered sequences	
numerable	Not applicable for either ordered or unordered sequences	
age	Nondeterministic where the operation is noncommutative or nonassociative for both	

	ordered and unordered sequences	
	Ordered Result	Unordered Result
cat	Ordered Result	Unordered Result
nt	Not applicable for either ordered or unordered sequences	
ultIfEmpty	Not applicable for either ordered or unordered sequences	
nct	Ordered Result	Unordered Result
ientAt	Returns a specified element	Returns arbitrary element
ientAtOrDefault	Returns a specified element	Returns arbitrary element
ept	Unordered results for both sequences	
	Returns a specified element	Returns arbitrary element
OrDefault	Returns a specified element	Returns arbitrary element
ll	Nondeterministic parallel execution for both sequences	
ipBy	Ordered Results	Unordered Results
ipJoin	Ordered Results	Unordered Results
sect	Ordered Results	Unordered Results
	Ordered Results	Unordered Results
Last	Returns a specified element	Returns arbitrary element
LastOrDefault	Returns a specified element	Returns arbitrary element
LongCount	Not applicable for either ordered or unordered sequences	

Min	Not applicable for either ordered or unordered sequences	
OrderBy	Sequence is reordered	New ordered section
OrderByDescending	Sequence is reordered	New ordered section
Range	Not applicable for either ordered or unordered sequences*	
Repeat	Not applicable for either ordered or unordered sequences*	
Reverse	Reverses	Doesn’t do anything
Select	Ordered Results	Unordered Results
Select (Indexed)	Ordered Results	Unordered Results
SelectMany	Ordered Results	Unordered Results
SelectMany(Index)	Ordered Results	Unordered Results
SequenceEqual	Ordered Comparison	Unordered Comparison
Single	Not applicable for either ordered or unordered sequences	
SingleOrDefault	Not applicable for either ordered or unordered sequences	
Skip	Skips the first n elements	Skips any n elements
SkipWhile	Ordered Results	Nondeterministic Performs on a current arbitrary order

Nondeterministic where the operation is noncommutative or nonassociative for both ordered and unordered sequences

	Takes the first n elements	Takes any n elements
While	Ordered Results	Nondeterministic. Performs on current arbitrary order
ThenBy	Supplemental to OrderBy on both sequences	
ThenByDescending	Supplemental to OrderBy on both sequences	
ToArray	Ordered Results	Unordered Results
ToDictionary	Not applicable for either ordered or unordered sequences	
ToList	Ordered Results	Unordered Results
ToLookup	Ordered Results	Unordered Results
Union	Ordered Results	Unordered Results
Where	Ordered Results	Unordered Results
Where (indexed)	Ordered Results	Unordered Results
Zip	Ordered Results	Unordered Results

Same defaults as AsParallel

The unordered results are shuffled actively; there just isn't any specific ordering logic that is applied to them. Sometimes, unordered queries can keep the source sequence order.

For those queries that make use of the indexed Select operator, there is a guarantee in PLINQ that the output elements are returned in the order of the increasing indices but there are no guarantees as to which index is assigned to which element.

That brings our look at PLINQ to an end. Next, we will look using unsafe code.

Using Unsafe Code

The one thing that puts C# apart from languages like C and C++ is that it doesn't have pointers as data types. Instead, there are references and we have the ability to create objects that a garbage collector can manage. Together with some other cool features, this design is what makes C# one of the safest of all the programming languages. At its core, the language does not allow us to leave variables uninitialized. It doesn't allow us to leave pointers hanging, and it doesn't allow expressions of index arrays that go far beyond what its boundaries allow. The one thing this does is eliminates a host of bugs that you often find in the C and C++ languages.

In both C and C++, almost every pointer of the construct type has a C# reference type equivalent. However, that doesn't stop situations from arising where it is necessary to have access to a pointer type. For example, accessing devices that are memory-mapped, trying to interface with the operating system, and implementation of algorithms that are time-critical – these are neither practical nor generally possible if you can't access a pointer and to help facilitate this, C# allows us to write unsafe code.

With unsafe code, you can declare a pointer and you can operate on it, you can perform a conversion between an integral type and a pointer, you can take a variable, and so on. In all honesty, unsafe code is not really any different to include C or C++ code in your C# program.

Unsafe code is, from the perspective of users, developers, and programmers actually a safe feature. However, there is a requirement for it to be marked using the unsafe modifier, thus explicitly declaring it, and stopping a programmer or developer from accidentally using features that are declared as unsafe. The way code execution works, unsafe code cannot and will not be executed in an environment that is not trusted.

Unsafe Contexts

It goes without saying that the C# unsafe features can only be used in an unsafe context and we create that by adding an unsafe modifier to a member or type declaration or we can use an unsafe_statement.

If you include an unsafe modifier in struct, class, delegate or interface declaration, the entire declaration for the type, which includes the interface, struct, or class, will be considered to be an unsafe contest.

You may also add an unsafe modifier to a field, property, method, operator, indexer, event destructor, instance constructor or static constructor declaration and the whole member declaration becomes an unsafe context.

If you use an unsafe_statement, it creates an unsafe context for use and the whole extent from a contextual point of view becomes an unsafe context.

You can see the syntax needed for each unsafe_statement type below:

```
class_modifier_unsafe
    : 'unsafe'
    ;


struct_modifier_unsafe
    : 'unsafe'
    ;


interface_modifier_unsafe
    : 'unsafe'
    ;
```

```
delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
```

```
    : 'unsafe' block
    ;
```

Here’s an example:

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

In this, we specify the unsafe modifier within the declaration for the struct and this makes the whole textual extent of the declaration an unsafe context. As such, you can declare the Left Field and the Right Field to be of type pointer. You could also write that example like this:

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

In this example, the unsafe modifiers are in the field declarations and that makes those declarations into unsafe contexts.

Aside from making an unsafe context, which allows us to use pointer types, the unsafe modifier cannot have any effect on any member or type.

Here’s another example:

```
public class A
```

```
{
   public unsafe virtual void F() {
      char* p;
      ...
   }
}

public class B: A
{
   public override void F() {
      base.F();
      ...
   }
}
```

In this example, we have an unsafe modifier for the method called F in the class called A and all it will do is make the method textual extent into an unsafe context. In B, where F has been overridden, we do not need to specify that modifier again. The only time you would need to do this is the F method in Class B requires access to any of the unsafe features.

It is a bit different when the method signature contains a pointer type:

```
public unsafe class A
{
   public virtual void F(char* p) {...}
}
```

```
public class B: A
{
   public unsafe override void F(char* p) {...}
}
```

Because the signature for F has a pointer type in it, we can only write it in the unsafe context. We can do this in two ways – as we did with A, we can make the whole class into an unsafe context, or, as we did with B, we can add the unsafe modifier into the method declaration.

Pointer Types

In the unsafe context, a type can be a value_type, a reference_type, or a pointer_type. However, you can also use a pointer_type in expressions external to unsafe contexts because this type of use is not considered to be unsafe:

```
type_unsafe
   : pointer_type
   ;
```

We write pointer_types as unmanaged_types or we use the void keyword and a * token:

```
pointer_type
   : unmanaged_type '*'
   | 'void' '*'
   ;


unmanaged_type
   : type
   ;
```

In a pointer_type, the type that we specify before the * token is known as a referent type of a pointer type. It is used to represent the variable type that the pointer type value is pointing at.

Unlike a value of a reference type, the garbage collector will not track a pointer. In fact, the garbage collector doesn't even know the pointers exist, much less the data they are pointing to. This is why pointers are not allowed to point at references or at structs that have references; also, the pointer's referent type has to be an unmanaged type.

Unmanaged types are any that are not constructed types, reference_types, and do not contain, at any nested level, a field of constructed type or a field of reference_type. Basically, unmanaged types are any of the following:

- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- char
- float
- double
- decimal
- bool
- any of the enum_types
- any of the pointer_types
- any struct_type defined by a user – mustn't be a constructed type and it must have fields that only have unmanaged_types in them

The rule that should be followed to mix references and pointers is reference or object referents can have pointers in them but pointer referents can't have references in them.

You can see some of the pointer types here:

Pointer Type	**Description**
byte*	Pointer to byte
char*	Pointer to char
int**	Pointer to pointer to int
int*[]	A one-dimensional array of pointers to int
void*	Pointer to unknown type

For any implementation, every pointer type must be of the same representation and size.

In C and C++ we declare multiple pointers in one declaration but, in C#, we write the * only with the underlying type; it cannot be used as a prefix punctuator on the individual pointer names. An example:

```
int* pi, pj;    // it isn’t written as int *pi, *pj;
```

For a pointer with type T*, the value is representing an address of a variable that is also of type T. The indirection operator for a pointer can be used to access the variable. For example, let’s say that we have a variable called P and it is an int* type. The expression of *P tells us that the int variable is at the specified address that is in P.

As with object references, we can have pointers that are null. When you put the indirection operator against a null pointer, the result will be behavior that is defined by the implementation, and all-bits-zero is used to represent pointers with the value of null.

To represent pointers to unknown types, we use the void* type. As we have an unknown referent, we can’t apply the indirection operator to a void* type pointer. We also can’t do any arithmetic on these pointers. However, the void* pointers may be cast to another type and other types can be cast to void*.

Pointer types are an altogether separate types category. Unlike the value and the reference types, a pointer type cannot inherit from objects and there are no conversions between object and pointer types. What you can do is conversions between two different pointer types and between an integral type and a pointer type.

We can't use a pointer_type as a constructed type (type argument) and type inference will not be successful on any call to a generic method where a type argument would have been inferred as a pointer type.

Although we can pass pointers as out or ref parameters, it can result in undefined behavior. This is because the pointer may be pointing at a local variable that doesn't exist at the time the method that was called returns, or it may be pointing at what was a fixed object but is now not fixed. For example:

```
using System;


class Test
{
    static int value = 20;


    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;


        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }


    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);    // undefined behavior
        }
    }
}
```

Methods can return values of a type and that can be a pointer type. For example, when you have a pointer to an int sequence, the element count of that sequence, and another int value, the method below will return the value address from the sequence, provided there is a match. If there isn't a match, then a null will be returned:

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}
```

In unsafe contexts, there are a few constructs that you can use for operations on pointers:

- The * operator – used for pointer indirection
- The - > operator – used for accessing struct members via a pointer
- The [] operator – used for indexing pointers
- The & operator – used for getting variable addresses
- The ++ and – operators – used for incrementing and decrementing pointers
- The + and – operators – used for arithmetic on pointers
- The !=, ==, <=, >=, < and > operators – used for pointer comparison
- The stackalloc operator – used for allocating memory from the stack
- The fixed statement – used for a temporary fix on a variable so we can obtain the address

Fixed And Moveable Variables

Fixed statements and address-of operators split the variables into two separate categories called fixed variables and moveable variables respectively.

A fixed variable will live in those storage locations that the garbage collector doesn't touch. Fixed variables include value parameters, local variables, and those created by dereferencing pointers. The moveable variable will live in the storage locations that the garbage collector will either dispose of or relocate. Moveable variables include array elements and object fields.

The address-of operator (&) will allow us to get a fixed variable address without any restrictions. However, as a moveable variable can be disposed of or relocated, we need to use a fixed statement to get the address and the address will only be valid for the lifetime of the fixed statement.

A fixed variable is:

- A variable that comes from a simple_name which is referencing value parameters or local variables. The exception is if an anonymous function captures the variable.

- A variable that comes from a form V.I member_access, where V is a struct_type fixed variable.
- A variable that comes from a form P* pointer_indirection_expression, a for P - > I pointer_member_access or a form P[E} pointer_element_access.

Any variable other than these is a moveable variable.

Note

Static fields and ref or out parameters are moveable variables, even if the parameter argument is a moveable variable. Variables that result from pointer dereferencing will always be fixed variables.

Pointer Conversions

In unsafe contexts, the set of implicit conversions available are extended so that these two are included:

- From pointer_type (any) to void* type
- From null literal to pointer_type (any)

In terms of the explicit conversions available in an unsafe context, these pointer conversions are now included:

- From pointer_type (any) to pointer_type (any other)
- From byte, sbyte, short, ushort, int, uint, long, ulong to pointer_type (any)
- From pointer_type (any) to byte, sbyte, short, ushort, int, uint, long, ulong

Lastly, the standard implicit conversion set has been expanded to include the following conversion for unsafe contexts:

- From pointer_type (any) to void* type

When you do a conversion between two types of pointer, the pointer value is never changed. In other words, it will not affect the underlying address that the pointer supplies.

When you convert one pointer type to another, if the pointer that results from the conversion has not been aligned properly for the pointed-to type, if the result gets dereferenced we get undefined behavior. Generally, the concept of being properly aligned is pretty much transitive. This means that if a pointer to type A is properly aligned for a pointer to type B and the pointer to Type B is properly aligned for a pointer to type C, the pointer to type A is also properly aligned for a pointer to type C.

In this example; we use a pointer to one type to access a variable of another type:

```
char c = 'A';

char* pc = &c;

void* pv = pc;

int* pi = (int*)pv;

int i = *pi;        // undefined

*pi = 123456;       // undefined
```

When we convert a pointer type to a pointer to byte, the variable's lowest addressed byte will be the result. When the result is incremented successively, up to the variable size, we get pointers to the variable's remaining bytes.

As an example, the method below will display all eight bytes from a double type as a hexadecimal value:

```
using System;


class Test

{

  unsafe static void Main() {

    double d = 123.456e23;

      unsafe {

        byte* pb = (byte*)&d;
```

```
            for (int i = 0; i < sizeof(double); ++i)
              Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

The output depends entirely on the ‘endianness’ or, to us mere mortals, the sequential order the bytes were arranged in numerical values when they were stored in memory.

Pointer Arrays

We can construct pointer arrays in an unsafe context but we can only apply some of the normal array type conversions on the pointer arrays:

- An implicit reference conversion that goes from array_type (any) to System.Array, along with any interface that it may implement. However, if you were to try to access the elements in the array using System.Array or through an implemented interface, an exception is thrown at run time – pointer types cannot be converted to object types.
- Implicit and explicit reference conversions from one-dimensional arrays of type S[] to System.Collections.Generic.IList<T> and any of the generic base interfaces. These can never be applied to a pointer array because we cannot use a pointer type as a type argument and we can’t convert from a pointer type to a non-pointer type.
- An explicit reference conversion that goes from System.Array and any of its implemented interfaces to any of the array_types can be applied to the pointer array.
- Explicit reference conversions between System.Collections.Generic.IList<S> and base interfaces to one-dimensional array types of T[]. This is because we can’t

use pointer types as arguments and we can’t convert from pointer to non-pointer.

All of these restrictions ensure that expanding a foreach statement over an array cannot be done with the pointer arrays. Instead, we must use a foreach statement of the following format:

```
foreach (V v in x) embedded_statement
```

In this, the x type is an array type of T[,…,] form, N indicates how many dimensions there are minus 1 and V or T are the pointer types. We can expand this with a nested for loop like this:

```
{
   T[,,...,] a = x;
   for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0);
i0++)
   for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1);
i1++)
   ...
   for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N);
iN++) {
       V v = (V)a.GetValue(i0,i1,...,iN);
       embedded_statement
  }
}
```

The variables called a, io, i1, … iN cannot be seen nor are they accessible to x, to the embedded statement, or any other of the program source code. If an explicit conversion has not been down from the element type T to V, an error is thrown and you can go no further. If x had a null value, at runtime a System.NullReferenceException is thrown.

Pointers In Expressions

In unsafe contexts, expressions can result in point types but, if that happened outside of the unsafe context, you would get an error at compile time. So, if any of the following results in a pointer type outside an unsafe context, you would get the error:

- simple_name
- member_access
- invocation_expression
- element_access

The primary_no_array_creation_expression and unary_expression will allow these constructs in an unsafe context:

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;


unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

We'll discuss these constructs now; associativity and precedence of unsafe operators are implied in the grammar.

Pointer Indirection

This expression is an asterisk (*) and a unary_expression:

```
pointer_indirection_expression
    : '*' unary_expression
    ;
```

The unary * operator is indicating pointer indirection and it is used to obtain variables that the pointer is pointing to. Where P is a pointer type T* expression, the result of the evaluation of P will be a type T variable. If you tried to apply the * unary operator to a void* type expression, or to a non-pointer expression, you would get an error at compile time.

When you apply the * unary operator to null pointers, the effect is implementation-defined and there is no guarantee that you won't get a System.NullReferenceException thrown.

If you assign an invalid value to the pointer, the * unary operator will have undefined behavior. Two examples of invalid values using the * operator to dereference a pointer are a variable address that is no longer in use and an address that has not been properly aligned for the type being pointed to.

If you evaluate an expression of *P and the result is a variable, it is considered to be initially assigned.

Pointer Member Access

This expression is a primary_expression, a "- >" token, an identifier and a type_argument_list which is optional.

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

In a P - >I, pointer member access, P has to be an expression of any pointer type other than void* and it has to denote a member that is accessible of the type that P is pointing to.

These pointer members evaluate a (*P).I.

An example:

using System;

struct Point

```
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
       }
    }
}
```

In this, we use the -> operator for accessing fields and to invoke struct methods though pointers. Because P->I is the exact equivalent of (*P).I, we could have written the Main method like this:

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

Pointer Element Access

This expression is a primary_no_array_creation_expression, and another expression inside a set of square brackets [].

```
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;
```

In the P[E] form of a pointer element, P has to be an expression of any pointer type that isn't a void* and E has to be an expression that can be converted implicitly to an int, uint, long or a ulong.

A pointer element of this format is evaluated exactly as *(P + E).

An example:

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

The character buffer inside a for loop is initialized using a pointer element access. As P[E] is exactly the same as *(P + E), the example could have been written like this:

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

The pointer element access operator will not make any checks for an out of bounds errors, and when an out of bounds element is accessed, the behavior is undefined.

The address-of Operator

This expression is an ampersand symbol (&) with a unary_expression

```
addressof_expression
    : '&' unary_expression
    ;
```

Let’s assume that you have an expression called E of a type T which is a fixed variable. The construct of &E will return the address of the variable that E gives. The result type will be T* and is a value. If E were not a variable, and was a read-only variable or denoted a moveable variable, an error would occur at compile time. In the case of it denoting a moveable variable, a fixed statement could be used to fix the variable temporarily before you get the address. Outside of a static constructor or an instance constructor for a class or a struct that has a read-only field defined, the field is not considered to be a variable; it is a value and that means you cannot get the address. In the same way, you cannot get the address of a constant.

There is no need for the argument of an & operator to be definitely assigned. However, after an & operation, the variable the operator is applied on is definitely assigned within the execution path that the operation happens in. It is down to the programmer to make sure that the variable is correctly initialized.

An example:

```
using System;


class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

In this i is definitely assigned after the &i operation that initialized p. The assignment inside *p will initialize i but it is down to the programmer to ensure that this initialization is included and that, if the assignment was taken out, that there wouldn't be a compile time error.

The rules for definitive assignment of the & operator ensure that you can avoid local variables being redundantly initialized. For example, some internal APIs will take a pointer that points to a structure that the API fills in. Calls to APIs like this generally pass a local struct variable address and, if that rule didn't exist, there would be a requirement for the struct variable to be redundantly initialized.

Pointer Increment and Decrement

In unsafe contexts, you can apply the ++ operator and the -- operators to pointer variables of any type except for void*. As such, for every T* type pointer, these are the implicitly defined operators:

- T* operator ++(T* x);
- T* operator --(T* x);

The first operator produces the result of x + 1 and the second produces a result of x – 1. So, for a type T* pointer variable, the ++ operator will add sizeof(T) to the address the variable contains and the -- operator will do the opposite – subtract sizeof(T) from the address the variable contains.

If an increment or a decrement of a point overflows the pointer type domain, the result will be implementation-defined but there will not be any exceptions thrown.

Pointer Arithmetic

In unsafe contexts, you can apply the – operator and the + operator to all pointer type values with the exception of those from a void* type. So, for every T* type pointer, these are the implicitly defined operators:

T* operator +(T* x, int y);

T* operator +(T* x, uint y);

T* operator +(T* x, long y);

T* operator +(T* x, ulong y);

```
T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Assuming that you have an expression called P and it is a T* pointer type. You also have an expression of N, which is of type uint, int, long, or ulong type. The expression of P + N and the expression of N + P will both compute the type T* pointer value that comes from adding N* sizeof(T) to the address provided by P. In the same way, the expression of P – N will compute the type T* pointer value that comes from subtracting N* sizeof(T) from the address provide by P.

Assuming that you have an expression of P and an expression of Q, both of T* pointer type, the expression of P – Q will give us the difference between the addresses that P and Q both give and then the difference is divided by sizeof(T). The result type will always be a long. Effectively P -Q will be computed as ((long)(P) – (long)(Q)) / sizeof(T).

An example:

```
using System;

class Test
```

```
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

The output of this will be:

```
p - q = -14
q - p = 14
```

If an operation of pointer arithmetic overflows the pointer type domain, the result will always be truncated in a similar fashion to implementation-defined but there will be no exceptions thrown.

Pointer Comparison

In unsafe contexts, you can apply the !=, ==, <=, =>, < and > operators to all pointe type values. These are the comparison operators for the pointers:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because there exists an implicit conversion from pointer types to the void* type, we can use the operators to compare operands of any pointer type. When you use the comparison operators, you can compare addresses that two operands give just as if they were both unsigned integers.

The sizeof Operator

The result from this operator will be how many bytes a variable of a specified type occupies. When you specify a type to sizeof as an operand, it must be of an unmanaged_type:

```
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;
```

The result that a sizeof operator produces is an int type value. For some of the predefined types, that result will be a constant value as seen below:

Expression	**Result**
sizeof(sbyte)	1
sizeof(byte)	1
sizeof(short)	2
sizeof(ushort)	2
sizeof(int)	4
sizeof(uint)	4
sizeof(long)	8
sizeof(ulong)	8
sizeof(char)	2
sizeof(float)	4
sizeof(double)	8
sizeof(bool)	1

For all the other types, the result will be implementation-defined and it is a value and not a constant.

The order that members get packed into a struct is not specified.

For the purpose of alignment, the start, the middle, and the end of a struct may have unnamed padding. The bits that are used as padding are of indeterminate content.

When you apply this to an operand of struct type, the result will be the total bytes on a variable of that specified type and that includes the padding.

The Fixed Statement

In the unsafe context, embedded_statement production allows for the fixed statement, which is another construct used to fix moveable variables temporarily. This will allow the address of the variable to stay constant throughout the life of the statement.

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')'
embedded_statement
    ;


fixed_pointer_declarators
   : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;


fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;


fixed_pointer_initializer
```

```
    : '&' variable_reference
    | expression
    ;
```

Every fixed_pointer_declarator will do two things – declare a local variable of the specified pointer_type, and initialize it using the address provided by the fixed_pointer_initializer that corresponds to it. When you declare a local variable in a fixed statement, it can be accessed in any of the fixed_pointer_initializers that occur on the right side of the declaration and it can be accessed in the fixed statement's embedded_statement. When you declare a local variable using a fixed statement, it is a read-only variable. An error will occur at compile time if the embedded statement tries to modify the variable using either the ++ and – operators, or tries to pass the variable as a ref or out parameter.

A fixed_pointer_initializer can be any one of the following:

- The & token with a variable_reference to a moveable variable of type T (unmanaged) so long as the T* type can be implicitly converted to the pointer type specified in the fixed statement. The initializer will compute the given variable address and the variable will stay at a fixed address for as long as the fixed statement is 'alive'.
- An array_type expression that contains unmanaged type T elements, as long as the T* type can be implicitly converted to the pointer type supplied in the fixed statement. The initializer will compute the address from the array's first element and the whole array will stay at a fixed address for the life of the fixed statement. If there are zero elements in the array or the expression is null, the address computed by the initializer will be equal to zero.
- A string type expression, so long as the char* can be implicitly converted to the pointer type provided in the fixed statement. The initializer will compute the address of the string's first character and the whole string will stay at a fixed address for

the duration of the fixed statement. If the string expression is null, the fixed statement behavior is implementation-defined.

- A member_access or a simple_name that provides a reference to moveable variable’s fixed-size buffer member so long as the fixed-size buffer member type can be implicitly converted to the pointer type the fixed statement provides. The initializer will compute a pointer of the fixed-size buffer’s first element. The fixed-size buffer will stay at a fixed address for as long as the fixed statement is ‘alive’.

For every address that a fixed_pointer_initializer computes, the fixed statement will make sure that the variable the address references isn’t subject to disposal or relocated by the garbage collector throughout the life of the fixed statement. For example, if the computed address is referencing an object field or an array instance element, the fixed statement will ensure that the object instance that contains it will not be disposed of or relocated while the statement is running.

The programmer has the responsibility to make sure that the pointers the fixed statements create cannot live longer than the statement execution. For example, a fixed statement creates pointers that are passed on to external APIs, the programmer must make sure that the API cannot retain the memory of the pointers.

Heap fragmentation may be caused by fixed objects because these objects cannot be moved. Because of this, you only fix objects when it is absolutely necessary and for as short a time as needed.

An example

```
class Test

{

   static int x;

   int y;
```

```
    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

This example shows a number of ways the fixed statement can be used. In the first statement, we are fixing a static field and obtaining the address of it. In the second, we fix an instance filed and obtain the address, and in the third, we fix an array element and obtain the address. In each one, if we had used the regular & operator, it would have been an error because each of the variables is a moveable variable. The fourth statement will give a result much like the third one.

In this example, a string is used by the fixed statement:

```
class Test
{
    static string name = "xx";
    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

In unsafe contexts, one-dimensional array elements are stored in an order of increasing indexes. This starts at index 0 and ends in index Length -1. In the case of a multi-dimensional array, the elements are stored in a way that the indices of the dimension to the right are increased followed by the leftmost, then the next right, and left, and so on. In a fixed statement that gets a pointer to an array instance called a, the pointer values that range from p to p + a.Length – 1 are representing addresses of the array elements. In the same way, the variables that range from p[0] to p[a.Length – 1] are representing the actual elements of the array. Given how the arrays are stored, a dimension array (any dimension) can be treated as if it were linear:

An example:

```
using System;

class Test
```

```
{
    static void Main() {
        int[,,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}
```

And the output of this would be:

Copy

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

Another example:

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

We used a fixed statement for fixing the array so we can pass the address to a method that will take a pointer:

An example:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

Here, we used a fixed statement to fix a fixed-size buffer of struct type so that we can use the address as a pointer.

When you fix a string instance and the result is a char* value, it will always point to a string that is null-terminated. In a fixed statement that gets a pointer p to a string instance s, the values of the pointers, ranging from p to p + s.Length are representing string character addresses and the pointer value p + s.Length will always point to character with a value of \0, or a null character.

When you modify an unmanaged type object using a fixed pointer, the result is generally undefined behavior. For example, strings are immutable and, because of that, it is down to the programmer to make sure the characters the pointer references to a fixed string cannot be modified.

One thing that is convenient is the automatic null-termination of strings. This works well for external APIs that are expecting strings of C-style. However, note that string instances cannot have null characters; if they are present when the string is treated as a char* that is null-terminated; the null characters will look truncated.

Fixed-Size Buffers

We use fixed-size buffers to declare in-line arrays of C-style as struct members. They are very useful when you want to interface with an unmanaged API.

Fixed-Size Buffer Declarations

A fixed-size buffer is a member used to represent the storage of fixed-length buffers of specified type variables. A declaration of a fixed-size buffer brings in at least one fixed-size buffer of a specified element type. We can only use fixed-size buffers in a struct declaration and they can only be declared in an unsafe context.

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;


fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed'
buffer_element_type fixed_size_buffer_declarator+ ';'
    ;


fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

The declaration could have a set of attributes, a combination of each of the four access modifiers (valid), a new modifier, and an unsafe modifier. The modifiers and the attributes are applied to every member that the declaration declares. If the same modifier appears more than once in a declaration, it would be an error.

Fixed-size buffer declarations cannot include static modifiers.

The type of buffer element in the declaration will specify what the element type is of the buffers that the declaration brought in. The element type must be of a predefined type:

- sbyte
- byte
- short
- ushort
- int
- uint

- long
- ulong
- char
- float
- double
- bool

Following the buffer element type is a list of the buffer declarators (fixed-size) and each of these will bring in a new member. The declarator is an identifier that provides the member name, a constant expression inside a set of [] square brackets, and tokens. In the constant expression, the number of member elements that the declarator brought in is dented. The constant expression type must be able to be implicitly converted to the int type and the value has to be a positive integer that is non-zero.

The fixed-size buffer elements will be laid out in memory sequentially.

If a fixed-size buffer declaration declares several fixed-size buffers, it is considered to be the equivalent of several declarations of one declaration that has the same element types and attributes:

```
unsafe struct A
{
  public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
  public fixed int x[5];
  public fixed int y[10];
  public fixed int z[100];
```

}

Fixed-Size Buffers In Expressions

A member lookup on fixed-size buffer members is exactly the same as member lookup on fields.

Using simple_name or member_access, you can reference a fixed-size buffer. When you use simple_name to reference it, you get the same effect as member access of this I, in which I is the fixed-size buffer member.

With member access of form E.I, where E is a struct type, and member lookup of I in that type results in a fixed-size member being identified, then E.I will be evaluated and classified in this way:

- A compile-time error happens if expression E.I is not in an unsafe context
- A compile-time error will happen if E gets classified as a value
- Otherwise, where E is a moveable variable and E.I isn't a fixed_pointer_initializer, then an error will occur at compile time
- Otherwise, where E is referencing a fixed variable and the expression result is a pointer to the initial element of the fixed-size buffer member of I inside E, the result will be S* type – S is I's element type and is a value
- We can use pointer operations from the initial element to access all subsequent buffer elements. Unlike array access, element access in a fixed-size buffer is considered an unsafe operation. and as such, isn't range checked.

In the next example, we are declaring a struct and using it with a fixed-size buffer member:

```
unsafe struct Font

{

   public int size;

   public fixed char name[32];

}
```

```
class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Definite Assignment Checking

A fixed-size buffer isn't subject to checking for definite assignment and the members of a fixed-size buffer are ignored when it comes to definite assignment checking on variables of the struct type.

When the outermost buffer member struct variable is static, is an array element, or is an instance variable of an instance of a class, the buffer elements are initialized automatically to their default value. In any other case, the initial content of the buffer will be undefined.

Stack Allocation

In unsafe contexts, declaration of local variables might have a stack allocation initializer used to allocate memory from the stack:

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;


stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

The unmanaged_type indicates the item type to be stored in the location that was just allocated and the expression is used to indicate how many items there are. Together, these two are used to specify the allocation size that is required. Because the stack allocation size is not allowed to be negative, specifying the item number as a constant_expression that will evaluate as a negative value will result in a compile-time error.

A form stackalloc T[E] is a stack allocation initializer that requires T to be an unmanaged type and that E is an int expression. E* sizeof(T) bytes is allocated by the construct from the call stack and a type T* pointer is returned to the allocated block. Should E be a negative value, the result is undefined behavior. Should E be zero, there won’t be any allocation and the returned pointer will be implementation-defined. If there is insufficient memory for allocation of a specified size of block, the result will be a System.StackOverflowException.

The newly allocated memory content is undefined.

Stackalloc cannot be used to free memory explicitly. All the stack allocated blocks of memory that are created when a function member is executed will be discarded automatically when the function member returns.

For example:

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

In this, we used a stackalloc initializer in the method called IntToString so that a buffer of 16 characters could be allocated on the stack. When the method has returned, the buffer will be discarded automatically.

Dynamic Memory Allocation

With the exception of the stackalloc operator, there are no predefined constructs in C# for the management of memory that is non-garbage collected. This kind of thing is usually provided by the class libraries or is directly imported from the operating system. For example, the Memory class in the example below shows how the heap functions from the operating system may be accessible from C#:

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
   // Process heap handle used in every call to the
   // HeapXXX APIs you can see in the following methods.
   static int ph = GetProcessHeap();

   // Private instance constructor that prevents instantiation.
   private Memory() {}

   // This allocates a memory block of the specified size and the
allocated memory will be
   // initialized to zero automatically.
   public static void* Alloc(int size) {
      void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }
```

```
    // Copies count bytes from src to dst. The source and destination
    // blocks are allowed to overlap.
    public static void Copy(void* src, void* dst, int count) {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd) {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd) {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--
ps;
        }
    }

    // Frees up a memory block.
    public static void Free(void* block) {
        if (!HeapFree(ph, 0, block)) throw new
InvalidOperationException();
    }

    // This will reallocate a memory block and if the request is for a
    // bigger size, the extra memory region will be initialized
    // to zero automatically.
    public static void* ReAlloc(void* block, int size) {
        void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block,
size);
```

```
    if (result == null) throw new OutOfMemoryException();
    return result;
}

// This returns the memory block size.
public static int SizeOf(void* block) {
    int result = HeapSize(ph, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}

// Heap API flags
const int HEAP_ZERO_MEMORY = 0x00000008;

// Heap API functions
[DllImport("kernel32")]
static extern int GetProcessHeap();

[DllImport("kernel32")]
static extern void* HeapAlloc(int hHeap, int flags, int size);

[DllImport("kernel32")]
static extern bool HeapFree(int hHeap, int flags, void* block);

[DllImport("kernel32")]
```

```
    static extern void* HeapReAlloc(int hHeap, int flags, void* block,
int size);


   [DllImport("kernel32")]
    static extern int HeapSize(int hHeap, int flags, void* block);
}
```

The example below makes use of the Memory class:

```
class Test
{
    static void Main() {
        unsafe {
            byte* buffer = (byte*)Memory.Alloc(256);
            try {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}
```

In this example, we have used Memory.Alloc to allocate 256 bytes of memory and to initialize the memory block with values that start from 0 and go to 255. Next, it will allocate a byte array of 256 elements and copies the memory block contents to the byte array using Memory.Copy. Lastly, Memory.Free is used to free up the memory block and the byte array contents were output to the console.

There is more to this subject but this will give you a good head start and an idea of what it is all about. Next, we move onto WinForms or Windows Forms.

An Introduction to Windows Forms

Windows Forms, or WinForms, are classes contained in the .NET framework and used to create Windows-based GUI desktop applications. The most important class in the namespace is called Systems.Windows.Forms is the Form class. This is the base class that is used for just about every window in WinForms applications. It is the base class for all of the top-level windows, and that includes the main window in an application, the view windows, and the dialog boxes that you may make for the application.

In this section, you will learn about the Form class and three other important classes from the same namespace:

- **Application –** this is used for managing application-level program characteristics
- **ApplicationContext –** This class is used to manage application behavior, i.e. how it is launched, how it is closed, etc.
- **MessageBox –** This is used to display dialog boxes that the user will know as message boxes.

We will also discuss some of the more common methods and properties that are associated with three of the classes – Form, MessageBox, and Application. Next, we will discuss the Microsoft Visual C# Forms designer and look at the source code that the designer generates for injection into your own source code. The .NET framework has several controls and forms that expose many properties that affect the behavior of the controls and forms, in every aspect and the Forms Designer is a visual interface for each of those properties.

While we focus on the required classes that you need to program a Windows Forms application, we’ll move on to what controls you have at your disposal, how user input is collected, and some other advanced techniques.

Understanding Windows Forms

Windows Forms is a rich framework that helps build client applications that are easier to use, offer supportive tools, and lower costs of deployment. You will often see desktop apps in Windows termed as rich-client apps as a way of differentiating them from the web apps that are browser-based and downloaded from one central place. We find the classes for creating these rich-client apps in the System.Windows.Forms namespace and, collectively these are called the Windows Form Classes. Here, you will find the Form class, control classes, clipboard interaction classes, menu classes, printing classes, and many more.

Using Forms as Dialog Boxes

One of the most common uses of forms is interactive dialog boxes. These can be very basic forms with nothing more than a text box control or the most elaborate of forms that contain multiple controls. Most of what we will be discussing here are those controls.

Some dialog box types are used so much that they now tend to be integrated into the operating system. Collectively known as common dialog boxes, these include the dialog boxes that are prebuilt to handle tasks like file selection, color specification, choosing fonts, printer settings configuration, and more. In the .NET framework, you will find several classes that give you access to those common boxes, and in the next few sections, we'll cover those classes.

More often than not, we use dialog boxes to show users a notification message. Rather than every programmer having to develop their own dialog boxes for generic messages, Microsoft built message box control into the Windows operating system. This is a special dialog box that makes it much easier to show a user a simple text message and it includes several button layouts and icons that have been predefined for you. The .Net framework wraps this message box in a class called MessageBox, which we will cover later.

Using Forms as Views

Typical applications built from Windows Forms tend to be made up of more than dialog boxes and top-level windows. They may also have child windows that allow easier interaction between users and the application. If you have knowledge of the MFC Library, or Microsoft Foundation Classes library that comes with Microsoft Visual C++, you will already know that this window type is known as a view and it is derived from a class called CView. In .NET framework, the modal dialog boxes, top-level windows, and the non-modal app windows all have the same base class – they share the Form class.

The Forms that we use as views tend to be different from those that we use for dialog boxes; the views don't have any button controls for dismissing the form. If you want a form used as a view, simply call the form's Show method, like this:

UserView userView = new UserView();

userView.Show();

And when you don't need the view any more you dismiss it using the Close method:

userView.Close();

Later, you will see how we use the Show command to display modeless dialog boxes. Most of the time, dialog boxes are modal and this means that you cannot work with the application while the box is displayed. When you use a nonmodal form for user interaction, multiple forms can be open at the same time and this gives the user a far better experience for some application types.

A Simple Windows Forms Project

The best way to learn about Windows Forms is to use them so let's create a simple project. Open a New Project in Visual Studio and choose Windows Applications. Click on OK and Visual C# .NET will generate a basic project for you automatically. Your Windows Form project will already have a simple Main form that is for the top-level window and some other files that are associated with the project.

A Look at the Visual C# >NET Files

In the Project directory, there are several files used to create the compiled program for the application. For basic projects, like SimpleForm, the files that are generated are:

- **App.ico -** This is the default icon for your application.
- **AssemblyInfo.cs –** This a C# source file that has the attribute declarations for targeting the project-generated assembly.
- **Form1.cs –** This the main project source file.
- **Form1.resx** - T\this is an XML file that is used for storing resource information that goes with Form1.cs.
- **SimpleForm.csproj** – This is the main C# project file.
- **SimpleForm.csproj.user** – This is a C# project file where you can find settings that are user-specific.
- **SimpleForm.sln** - This is the main C# project solution file.
- **SimpleForm.suo** - This is a C# solution file that has the information specific to users.

Out of all of these files, only two of them contain any source code – Form1.cs and AssemblyInfo.cs. The AssemblyInfo file will only contain assembly attributes that are used to define the characteristics for the assembly that is compiled and Form1.cs is where the bulk of the source code for the new project is found. When you open a source file that will implement a form, you will see the form displayed, at first, in design mode.

Press on F7 or right-click on the form and select View Code to see the source code for the form. Both ways will show the source code in a format so you can edit the code while, on another tab, C# .NET will keep the form in design mode open. Below is the standard contents of Form1.cs:

```
using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;
using System.Data;

namespace SimpleForm
{
    public class Form111 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container components = null;

        public Form111()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Clean up the resources that are in use.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// This method is required for Designer support and you must
NOT modify
        /// the method contents using the code editor.
        /// </summary>
        private void InitializeComponent()
        {
           this.components = new System.ComponentModel.Container(
);
           this.Size = new System.Drawing.Size(300,300);
           this.Text = "Form111";
        }
        #endregion

        /// <summary>
        /// This is the main application entry point .
        /// </summary>
        [STAThread]
        static void Main()
        {
           Application.Run(new Form111());
        }
    }
```

```
}
```

If you are familiar with console applications, you will find that this source code is similar to that used for those applications with a couple of differences. One is the using statements. At the top of the file, those using statements are referencing several namespaces that are commonly used in Windows Form applications, such as these:

- **System.Collections** – This is the Collections namespace and it contains classes that will provide containment for other types in the style of collection containment.
- **System.ComponentModel** – This namespace has classes used to implement the behavior of the controls and containers. A form will use a System.ComponentModel.Container instance to track the components that the form uses.
- **System.Data** – This namespace has classes in it needed for access to Microsoft ADO.NET databases. You don't use this namespace in default. applications so, if you are not doing any ADO.NET programming, you can take this using statement out of the code.
- **System.Drawing** – This namespace includes the classes that give you access to the GDI+ drawing functions.
- **System.Windows.Forms** – This namespace has all of the Windows Forms classes useful to call the namespace types using the much shorter and easier names.

There is another difference between Windows Forms and console applications – the Form1 class has been derived from the base class of System.Windows.Forms.Form. As we said earlier, this is the base class for any form window that you will create in your .NET application.

Note that there is a part of the source code that is protected using the preprocessor directives called #region and #endregion. Both of these define code regions for the Code Editor that are collapsible. In the region, we have in the file, we have some source code that the Forms Designer uses to store configuration information and properties for the form, and also child controls that the form owns.

When you include code related to design within a region, by default, it can be collapsed and that protects it from accidental editing. If you want to see that code in the Code Editor, look for the plus sign beside the region that is labeled as Windows Form Designer Generated Code and click on it.

In a Windows Form application, the Main method also has a call to Application.Run, as you can see below:

```
static void Main()

{

   Application.Run(new Form1());

}
```

Later, when we discuss starting applications, the message loop that all Windows applications require will be started by the Run method and the main application form is displayed. Once that main form has been closed, the Run method returns.

Lastly, every form class will interact with handles in the operating system, more specifically with window handles, and that means every form class will also implement a method called Dispose. If you include resources that need disposing of, you should always use the Dispose method to arrange for the resources to be disposed of.

Managed sources can be references only if disposing evaluates to true. If it evaluates to false, you cannot touch managed references safely because the garbage collector may already have reclaimed the object.

Executing Windows Forms Projects

We compile Windows Form projects in the same way that we compile any project written in C#. Just open the Build menu and click on Build Solution. If you want your Windows Form project opened in debugger mode, press on the F5 key or click on the Debug Menu and click Start. If you were to execute the SimpleForm code from above, you would see a simple form on the screen with no buttons and no controls.

SimpleForm Application

Windows Form projects are far easier to debug than console applications. All Windows Form projects are classed as event-driven because they will respond to specified events, and that makes it much easier to look at their behavior in the debugger. To do this, you just put a breakpoint inside one of the event handlers.

Adding a New Form to a Project

Most of the time, your Windows Forms projects will have more than one form. Most have several that are used to handle all the different parts of the applications and generally, a new form is nothing more than a class that descends from the base class of System.Windows.Forms.Form.

You can add new forms in two ways – click on the Project Window and then click on Add New Form, or go to Solution Explorer and right-click on the project name; click on Add Windows Form from the menu. Both ways will open a dialog box to Add New Item.

To add the form, just type in a name for the new class and click on the Open button. The new form class gets created by .Net and added to your project. Any forms that you add to your project will automatically have the Dispose method in them along with a method called InitializeComponent that the Forms Designer uses.

Modal Forms vs. Modeless Forms

Forms can be displayed to users in two different ways:

- **Modal Forms –** this type of form will not allow any other part of the application to be accessed while it is being displayed. This form is useful when your dialog box contains information that a user must accept or acknowledge before they can move on.
- **Modeless Forms –** this type of form will allow access to other parts of the application while being displayed. This form is useful when you want a form that can be used over a period of time.

Modal form examples include message boxes, dialog boxes with error information, and dialog boxes used to open or save files. Modeless forms are used when application interaction is required, for example, when child forms are displayed in MS Word.

Most of the time, modeless forms are the preferred format as the user has more control over application flow. However, when there is a need for a user to acknowledge information or the user is required to input some information or some other kind of input is required before application execution can continue, modal forms are ideal.

The form will be displayed as per the method used for calling it. For modeless forms we use the Show method, like this:

Form addressForm = new AddressForm();

addressForm.Show();

And don’t forget that the Close method is required to destroy forms that use the Show method to display them:

addressForm.Close();

Or you could just make the form invisible by using the Hide method:

addressForm.Hide();

For modal forms, the ShowDialog method is used:

addressForm.ShowDialog();

The call to this method will not return until the user has dismissed or closed the dialog box.

The DialogResult Value

A modal dialog box will return a value that is used to determine the method of the dialog box dismissal. You should know what this value is because the code that invokes the box can determine whether a valid date is contained in the dialog box, like this:

Form addressForm = new AddressForm();

DialogResult result = addressForm.ShowDialog();

```
if(result == DialogResult.OK)
{
   // Dialog box contents are valid.
}
```

The return value for DialogResult usually tells us which button the user clicked to close the box. The value comes from the DialogResult enumeration, and in that, you will find these values:

- Abort
- Cancel
- Ignore
- No
- None
- OK
- Retry
- Yes

Usually, the return value will be the value of the button that the user clicked. For example, if the user pressed the ESC button, the value would be DialogResult.Cancel.

Passing Values to Forms

When you use forms for dialog boxes, more often than not, you will want information passed to and from the box. Generally, the box would need to be populated with the information that the users initially see and then, later, once the user has dismissed the dialog box, you would collect their input.

The easiest way of passing information to and from forms is simply to define the properties that give us access to the user-provided input. For example, in the code below, we define properties to pass information about name and address to a dialog box which will then show the user this information:

```
public void DisplayAddress(Address userAddress)
```

```
{
    AddressForm addressForm = new AddressForm();
    addressForm.UserName = userAddress.user;
    addressForm.StreetAddress = userAddress.StreetAddress;
    addressForm.ShowDialog();
}
```

We can associate each of the properties with a specified member variable that is used to initialize the controls of the dialog box. When we use the AddressForm class, we can use the properties to put the right values in the dialog box, as you saw in the last example. When you use it with the DialogResult value that ShowDialog returned, we can only retrieve the property values when the user clicks on the OK button, as seen in the next example:

```
public Address GetAddressForUser(string user)
{
    Address userAddress = null;
    AddressForm addressForm = new AddressForm();
    addressForm.UserName = user;

    DialogResult result = addressForm.ShowDialog();
    if(result == DialogResult.OK)
    {
        userAddress = new Address();
        userAddress.StreetAddress = addressForm.StreetAddress;
        userAddress.City = addressForm.City;


    ⋮
    }
    return userAddress;
}
```

Properties are useful because they obtain values from forms like dialog boxes very easily. When the user tries closing the form, the input can be validated and control values stored in the relevant member variables.

Using Message Boxes to Display Information

The class called MessageBox is used to provide user interaction; a message box, which is a special type of dialog box with a message in it, is displayed to the user. The box may also contain buttons and an optional icon. We tend to use these to present information that we can format into a text message.

The one thing that you can't do with the MessageBox class is create an instance of it. Instead, the box is displayed using the static Show method:

```
MessageBox.Show("A simple message box");
```

This will show a very basic message box and this has the string that was passed in as a parameter and a button that says OK. There are no less than 12 overloaded Show method versions that provide you with a lot more flexibility in how your message box behaves, and that includes the icons and buttons that are shown on the box.

The next section explains what options the overloaded method versions provide.

If you want a title or a caption on the message box, a second string needs to be passed to MessageBox.Show, like this:

```
MessageBox.Show("A simple message box", "MessageBox demo");
```

Specifying Message Box Buttons

You don't have to have the OK button on your message box; you can have other ones and this means you can ask questions in the message box that require a response from the user. You can have up to three buttons in your box but you can't define these buttons yourself. Instead, you make your choice from a set of predefined groups in the MessageBoxButtons enumeration.

In the next example, you can see how to pass a value as a parameter from MessageBoxButtons to specify a different layout for the buttons on your message box:

```
DialogResult result = MessageBox.Show("A message box that
has 3 buttons",
                                "MessageBox demo",
                                MessageBoxButtons.YesNoCancel);
switch(result)
{

  ⋮

}
```

The Show method return value is one of the return values from DialogResult that we looked at earlier.

Adding Icons to Message Boxes

We used the MessageBoxIcon enumeration to specify which icon the message box displays and there is a choice of four. However, although there are only four icons, the MessageBoxIcon enumeration has nine members:

Member	**Description**
Asterisk	Shows a circle with a lowercase i
Error	Shows a red circle with a white X in it

Exclamation	Shows a yellow triangle with an exclamation point in it
Hand	Shows a red circle with a white X in it
Information	Shows a circle with a lowercase i
None	There won’t be any icon shown
Question	Shows a circle with a question mark in it
Stop	Shows a red circle with a white X in it
Warning	Shows a yellow triangle with an exclamation point in it

If you want to specify which of the icons is shown in your message box, a value from the MessageBoxIcon enumeration must be passed to the Show method and you have to specify what button layout you want, along with other parameters, like below:

```
DialogResult result = MessageBox.Show("A message box that
has an icon",
                               "MessageBox demo",
                               MessageBoxButtons.YesNoCancel,
                               MessageBoxIcon.Asterisk);
switch(result)
{

   ⋮

}
```

Defining Default Buttons for Message Boxes

You can specify which of the message box buttons will be the default by using the MessageBoxDefaultButton enumeration. When you do this, if the user presses the enter key, the specified button is returned to the user as a return value from DialogResult. There are three buttons in the enumeration and they are called Button1, Button2, and Button3. The next example shows how to specify a default button using the enumeration:

```
DialogResult res = MessageBox.Show("The default button will
be Ignore",
                         "MessageBox demo",
                         MessageBoxButtons.AbortRetryIgnore,
                         MessageBoxIcon.Asterisk,
                         MessageBoxDefaultButton.Button3);
switch(res)
{
  ⋮
}
```

Using this code will result in a message box with three buttons – Abort, Retry, Ignore and the default is the Ignore button.

Controlling Special Cases

We can use the MessageBoxOptions enumeration to control the way a user sees the message box and define special-case behavior. Those members are:

Member	**Description**
DefaultDesktopOnly	The message box will be shown on the active desktop.
RightAlign	The text in the message box is aligned to the right.

ading	This specifies that the text in the message box is shown in right to left reading order.
eNotification	The message box will be shown on the active desktop.

We can combine these values with the bitwise OR operator. The example below shows how the values are combined:

```
DialogResult res = MessageBox.Show("The service has not\nstarted
\n.",
                              "Service demo",
                              MessageBoxButtons.AbortRetryIgnore,
                              MessageBoxIcon.Exclamation,
                              MessageBoxDefaultButton.Button3,
                              MessageBoxOptions.RightAlign ¦
                              MessageBoxOptions.DefaultDesktopOnly);
switch(res)
{

   ⋮

}
```

This will show a message box that a Windows Service can use, and the text will be aligned to the right.

Specifying a Parent Window for a Message Box

To make sure that your message box is shown in front of the right form, a reference to that form can be specified when you call the MessageBox.Show method. The overloaded Show method versions that we mentioned earlier all have a similar overload allowing you to specify the parent to the message box. Most of the time, all you need to do is make sure the first parameter is a pointer to the form:

MessageBox.Show(this, "Just a message", "MessageBox demo");

That first parameter can reference any object that can implement the interface called IWin32Window. All of the forms and controls in the .NET framework implement this interface and it is a standard one for all types that encapsulate the Win32 window.

Controlling a Windows Forms Application

We use the class called Application to provide Windows Forms applications with application-level behavior. You do not directly create instances of this class. Instead, static methods and properties that the class exposes are invoked. The Application class provides several static methods; like Run, or Exit, and this lets you control the way your application starts and stops. The static properties that the class exposes will let you determine details such as the path the application data is stored on. While you can't create an instance of the class, we will look at how context objects and event handlers can be used to control how the application behaves.

Starting an Application

We call the static method of Application.Run to start a Windows Form application, like this:

Application.Run(new Form1());

The Run method has three overloaded versions and the example above is using the most common version – it will accept the application's top-level form as a parameter. The Run method creates a message loop and the form that was passed in as a parameter is displayed. This Run version is automatically called .NET Windows Forms projects.

You could also call the method without any parameters:

Application.Run();

When you call Run without parameters, a Windows message loop is created by the Application class on the current thread but an initial form is not displayed. This is useful if you want the form displayed later or not at all.

The last version will accept an object of ApplicationContext type that the Application class uses. The default object that was used with the other two versions automatically exits the application as soon as the main form has been dismissed or closed. When you derive a new ApplicationContext class, specialized behavior can be defined. In the next example, we have such a class that will override what the default behavior is to shut the application down:

```
public class QueryApplicationContext: ApplicationContext
{
   private string _saveMessage = "Exit and discard changes?";
   private string _caption = "Application Exit";

   public QueryApplicationContext(MainForm form)
   {
      form.Show();
      form.Closing += new CancelEventHandler(FormClosing);
   }

   private void FormClosing(object sender, CancelEventArgs e)
   {
      DialogResult result = MessageBox.Show(saveMessage,
                               _caption,
                               MessageBoxButtons.YesNo,
                               MessageBoxIcon.Question,
                              MessageBoxDefaultButton.Button1);
        if(result == DialogResult.Yes)
        {
            ExitThread();
        }
        else
        {
            e.Cancel = true;
        }
    }
}
```

In this, we used the FormClosing method to handle the main form's closing event. That event is raised when the form is closed allowing the handler to stop closure by passing in a parameter of the CancelEventArgs object. In the method called FormClosing, the user sees a confirmation method; if they click on Yes, the application calls ExitThread and terminates. If they click on No, the event will be canceled.

If you want a specialized version of the class of ApplicationContext, simply pass a class instance to Application.Run, like this:

```
static void Main()
{
    MainForm form = new MainForm();
    QueryApplicationContext appContext;
    appContext = new QueryApplicationContext(form);
    Application.Run(appContext);
}
```

Application Path Information

When you want to locate the files that your Windows Form application uses, more often than not, you will need to know the path information. In the Application class, there are properties that make it easy to find the paths that are used most commonly, like the application path, or the folder where application data is stored.

You can obtain the path to the executable responsible for landing the current application by using the property called ExecutablePath, like this:

string executablePath = Application.ExecutablePath;

If you want the start-up directory path, use the property called StartupPath, like this:

string startupPath = Application.StartupPath;

According to the design guidelines for Windows applications, all application data has to be stored in certain ways to enable users to easily interact with the application. You should always store the application data in a subdirectory of the folder called Documents and Settings. The information that is specific to users should be stored in a path in My Documents – it should map to Documents and Settings\<user>name>. When you store data like this, users can have one location for the data and the application easily used in roaming profiles. To help you get this right, the Application class calculates all the paths for you.

Where you have common data that all users of the Windows Form application share, it should be stored in a Documents and Settings subdirectory that is specific to the application. You use the property called CommonAppDataPath to see the location of that subdirectory, like this:

string commonDataPath = Application.CommonAppDataPath;

The directory location will be in a subdirectory of Documents and Settings called <company_name>\<product_name>\<version>. The names for company and product are determined through inspection of the assembly attributes from the file called AssemblyInfo.cs. If these are empty attributes, as they would be when you first create the project, both names are set to the name given to the C# project.

In order for the directory to be created, you must access the property to request its location. Then, the Application class makes the assumption that you want to store information inside the common shared application data directory and, if it isn't already there, it will create the directory.

You can obtain the path to get to the user-specific information using the property called UserAppDataPath, like this:

```
string userDataPath = Application.UserAppDataPath;
```

The path that leads to the per-user information comes under the same set of rules that the common application data does with one exception – the current user identity is used to compose the path. For example, if the user was called Kelly, the per-user path would be Documents and Settings\Kelly\Application Data.

Performing Idle Work

In the Application class is an event that is raised when the message loop isn't doing anything. This is called the Idle Event and is used to trigger tasks that are lower in priority and are done when nothing else is being done. For example, the Idle event could be used to trigger updates to the interface elements, like the status bar, or some other displays that hold information.

To handle the Idle event, the first step is to define a method that will conform to the signature of the EventHandler delegate, like this:

```
private void OnIdle(object sender, EventArgs e)
{
   // Perform idle processing here.
}
```

Before the event handler is called, Idle events must be subscribed to and, to do this, we create an EventHandler delegate. This is then appended to any Application.Idle delegates that already exist, like this:

```
MyForm()
{
   ⋮
   Application.Idle += new EventHandler(OnIdle);
}
```

Closing an Application

By default, when the main form is closed, an application will exit. However, this can be overridden by using the ApplicationContext class, as we discussed earlier. However, there is another way – you can interact directly with the Application class.

In the class is a method called Exit and you can use this to initialize application shutdown. Any thread can call the Exit method, like this:

```
Application.Exit();
```

This method will not make the application immediately close down. Instead, every one of the message pumps will be closed and the Run method will return, and it is this that generally makes the application close down. This is because the Main method in the application tends to only have code that cleans the application resources once Run has returned, like this:

```
static void Main()
{
   ⋮
```

```
    Application.Run(new Form1());
    appMutex.Close();
    stream.Close();

    ⋮

}
```

Normally, Exit is not directly called because it would shut down any open forms immediately. Instead, we call the Close method from the main form.

If you want a specific thread from a Windows Form application closed, the ExitThread method is called:

```
if(Application.MessageLoop == true)
    Application.ExitThread();
```

In this, the property called MessageLoop determined if the current thread has a message loop on it before ExitThread is called. If there is a message loop on the thread, ExitThread will make that loop shut down. If the current thread's Run method is called, ExitThread will make the Run method return to what called it.

If you want a notification displayed when an application exits, a handler needs to be added to the ApplicationExit event. As you can see below, this event handler will help to clean up the application resources:

```
// Form constructor
AddressForm()
{

    ⋮
```

```
   Application.ApplicationExit += new EventHandler(OnAppExit);
}

private void OnAppExit(object sender, EventArgs e)
{
   appMutex.Close();
   stream.Close();
}
```

When you handle ApplicationExit events, resources that have not been disposed of yet can also be cleaned up. What you can't do is control the application exit. By the time you invoke the ApplicationExit event handler, the application has already started closing down.

Using Form Properties To Change Form Behavior

Every control and form object that is created using the Form Designer will have behavior control properties. There are two ways to access these properties – through your C# application or by using the Properties window for either control or form.

The Properties Window

When you use the Properties window to update property values, the Form Designer will automatically generate the code needed and will inject the code straight into the source file. There aren't any property configuration files hidden away to control how your forms and controls behave, it's all down to the source code.

Setting Border Styles

You can define border styles for forms and this is one of the most important of the form properties because the border is one of the first things that a user sees and it gives them an idea of how they can interact with the form. For example, if it looks to the user like a window can be resized, they will expect to be able to do it and your code must have the right properties for correct rendering. We use the FormBorderStyle property to define border styles and it must be set to one of the values in FormBorderStyle enumeration:

Value	**Description**
Fixed3D	A fixed border that is three-dimensional
FixedDialog	A fixed border in thick dialog style
FixedSingle	A fixed border with a single line
ToolWindow	A tool window that cannot be resized. It will not show in the taskbar nor will it show when ALT+TAB is pressed. While these forms don't show in the taskbar, you must make sure that the property for ShowInTaskbar is set as false – it has a default of true.
	There is no border
e	The border is resizable
eToolWindow	A tool window with a resizable border

In the next example, you can see how the FormBorderStyle property is set through the code:

```
FormBorderStyle = FormBorderStyle.FixedToolWindow;
```

The form border properties don't just affect the edge rendering and resizing behavior of the form. If the property is set to FormBorderStyle.None, there won't be a caption shown because there is no title bar. If you set the border style as one of the styles for tool windows, the title bar height is reduced and you won't see the system menu either.

By default, a form that is created with Form Designer has the FormBorderStyle property set as FormBorderStyle.Sizable but this is not suitable for use with dialog box forms because these are usually fixed in size.

Defining Other Form Properties

As well as the border style, there are several other properties in the Form class that are used to control many parts of the behavior and appearance of the Form. Some of the more common ones that the Form class exposes are:

- **BackColor –** This is used to specify the form's background color. The value is set by default, to Control and that will specify that the background color should be the one used by the Windows color scheme currently in use.
- **Size –** This specifies the form dimensions.
- **StartPosition –** This specifies the location where the form is to be created.
- **Text –** This specifies the caption for the form. By default, the value is the name given to the form, but usually, this should be changed to something that has more meaning.
- **WindowState –** This is used to specify the current form state and that comes from the WindowState enumeration.

The Color structure represents the .NET framework 32-bit color values. The color will be specified as a value of 32-bits made up of ARGB – alpha, red, green, blue – components. The Color type has many different predefined colors, not just the common ones like Red, Green, Blue or White, but some more exotic ones like BurlyBrown, AliceBlue, SeaGreen, and more. The code snippet below shows the color type required to change the color of the form background:

BackColor = Color.SeaGreen;

The StartPosition property is used to specify the position of a new form, and by default, it is set as WindowsDefaultLocation.

The FormWindowState property is updated when the application is executing and is set as one of these values, depending on what the current state is:

- Minimized Form is currently minimized.
- Maximized Form is currently maximized.
- Normal Form is currently at its normal size.

In the next example, we are setting several properties including Text, BackColor, and StartPosition:

```
public Form1()
{
   InitializeComponent();
   FormBorderStyle = FormBorderStyle.FixedToolWindow;
   BackColor = Color.AliceBlue;
   Text = "Basic Form Demo";
   StartPosition = FormStartPosition.CenterScreen;
}
```

That concludes a basic look at Windows Forms applications. The System.Windows.Forms namespace classes can help us create some rich-client C# applications. The Form class is the base for most of the windows in one of these applications, and when you add new forms to the project, the class is created with the required methods predefined for you. The Form class also exposes several properties that can be changed through the code or through the Properties window for Form Designer.

The MessageBox class is used to show a user a message box and there are several options we can modify that control how the message box looks. With this class, you can control the icon display placement of buttons, and several other properties.

The Application class controls Windows Forms applications and is responsible to get the message loop started and show the top-level window. The Application class also has properties that help us to store application or per-user data. We don’t create instances of the Application class; instead, we call the static methods that the class exposes, like Application.Run and Application.Exit. And form properties can also be used to control form behavior, like border style and background properties.

This is a subject that could go on and on but, in all truthfulness, it is a book on its own. This guide was just designed to give you an overview of what is possible.

Conclusion

This really is just the beginning of your C# journey. There is so much more to learn and so much further you can go; this is only the tip of the iceberg. You've learned the syntax and the language conventions; you've learned what goes into a program and how to make things more complex. The most important part of any computer programming language is the logical aspect. Yes, you may have learned how to loop through an array like you would look through a list of grocery items. But, if someone asked you to take an image and process it, would you know how to do it? Would you know how to scale it? Rotate it? Flip it? By using an API and your brain, you could figure it out.

Real programming requires you to use the logical part of your brain. It's much like learning a foreign language. Learning the language conventions and the grammar rules is one thing, but the real skill comes from writing the language in a clean and structured way and in a well-thought-out and logical manner. As a programmer, the skill comes from being able to put your knowledge and logic into practice by writing clean code that makes sense, and most importantly, that works.

Learning a program and truly understanding it is great – it really is the nearest we have to having magic power. You can make something from nothing, build it from the ground up, and have it do exactly what you want it to do.

In this series of guides, you learned C# basics, and while I couldn't possibly teach you everything you need to know, I have at least set you off on a journey of discovery. I hope that you now feel you can take the next step, and that you are prepared to dive deeper into C# and learn it until you are proficient at using it.

Thank you for taking the time to read my guide and I want to take this opportunity to wish you luck on your C# programming journey.

Answers

1. The answer is b) friend.
2. The answer is c) Indexer
3. The answer is c) sealed
4. The answer is d) class A where T : class, struct
5. The answer is a) Interface : var a = new IComparable();
6. The answer is b) button1.Click += delegate(EventArgs e) {MessageBox.Show("Click");};
7. The answer is d) statement 4
8. The answer is d) -2147483648
9. The answer is c) C# exceptions occur at run time
10. The answer is a) public static long Main(string[] args)
11. The answer is b) If T is string type, variable t will have an empty string
12. The answer is c) Binary tree
13. The answer is b) IEnumerable
14. The answer is d) A runtime exception will occur
15. The answer is a) var a = null;
16. The answer is b) a = (a is null) ? "" : a;
17. The answer is d) Satellite Assembly
18. The answer is c) (Destructor) static ~MyClass() {}
19. The answer is d) Delegate
20. The answer is a) private var GetData() {}
21. The answer is c) var a = new IComparable();
22. The answer is d) operator ||
23. The answer is c) At CLR runtime
24. The answer is a) int[][][] cc = new int[10][2][];
25. The answer is c) The anonymous type allows you to use delegates for methods
26. The answer is a) a=true, b=false
27. The answer is e) All of the above are correct
28. The answer is d) The += and -= operators cannot be used by a C# delegate

References

https://www.kunal-chowdhury.com

https://www.codeproject.com

https://www.tutorialsteacher.com

https://www.ict.social

https://programmingwithmosh.com

https://csharp.net-tutorials.com

https://www.tutorialspoint.com

https://www.c-sharpcorner.com

https://www.guru99.com

https://www.programiz.com

https://www.w3resource.com/

http://rbwhitaker.wikidot.com

https://www.javatpoint.com

https://www.infoworld.com

https://www.geeksforgeeks.org

https://www.c-sharpcorner.com

https://code-maze.com

https://introprogramming.info

https://docs.microsoft.com

https://codescracker.com