

C#

THE ULTIMATE BEGINNER'S GUIDE TO
LEARN C# PROGRAMMING STEP BY STEP



RYAN TURNER

C#

*The Beginner's Ultimate Guide to Learn
C# Programming Step by Step*

Ryan Turner

© Copyright 2018 – Ryan Turner – All rights reserved.

The content contained within this book may not be reproduced, duplicated or transmitted without direct written permission from the author or the publisher.

Under no circumstances will any blame or legal responsibility be held against the publisher, or author, for any damages, reparation, or monetary loss due to the information contained within this book. Either directly or indirectly.

Legal Notice:

This book is copyright protected. This book is only for personal use. You cannot amend, distribute, sell, use, quote or paraphrase any part, or the content within this book, without the consent of the author or publisher.

Disclaimer Notice:

Please note the information contained within this document is for educational and entertainment purposes only. All effort has been executed to present accurate, up to date, and reliable, complete information. No warranties of any kind are declared or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. The content within this book has been derived from various sources. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, — errors, omissions, or inaccuracies.

Table of Contents

Introduction

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2
HEADING 3

CHAPTER

HEADING 2

HEADING 3

CHAPTER

HEADING 2

HEADING 3

CHAPTER

HEADING 2

HEADING 3

Conclusion

HEADING 2

HEADING 3

Description

C# has a growing popularity and is steadily being used more and more. Many famous sites and enterprise level applications are powered by C#. You can find C# in the scientific computing being run on supercomputers. System administration tasks such as package management and configuration use C# as well. No matter what your programming interests, the possibilities for learning, exploring, and growing are endless.

The first thing that you might want to look into when it comes to a coding language is to understand why C# is so special and why you would even want to learn how to use this particular programming language.

There are many different coding languages out there to choose from, and they all work differently. Some people go with a certain coding language because it is easier to use than some of the others and some will choose their programming language based on the kind of programming they can do with it.

C# is extensively utilized to create software ranging from desktop applications to dynamic websites. C# incorporates various advantages of different programming languages. Since the strong programming structure of C# and .NET Framework is combined with visual programming in the Visual Studio environment, you can create visually elegant and functional applications easily.

Historically, a team led by Anders Hejlsberg of Microsoft Corporation designed and developed C# by taking the advantageous sides of C, C++ and Java 18 years ago. Since then, C# is continuously developed and supported. By the way, you may wonder what C# means literally. In this naming, # is taken from musical notation where it means a semitone higher in pitch. Therefore, C# is a “semitone higher language” compared to the standard C programming language.

Working with the C# language is a great experience, and you'll be writing a lot of great codes in no time. Use your time to look through this guidebook to help you get started with writing in this language today!

Introduction

Although there is nothing easy about learning a new language of any sort, the aim of this book is to allow you to pick up and understand as quickly and easily as possible. The goal of this is to provide you with complete and comprehensive skills that are able to help you retain important knowledge and guide your programming and coding experience in C#. In order for you to learn in the most efficient manner, I hope that you will also do the practice that will reinforce the lessons we explore in this book.

The book will cover a basic description of what C# is, and finally give you some helpful hints and resources to reinforce any subjects or skills where you may need a little more assistance.

This language is one that will mirror some components of other computer programming languages—it is almost invaluable experience if you are familiar with other languages themselves. Although this book is not quite just a beginner's manual that introduces you to the art of programming, it will reiterate and acknowledge a lot of these skills to help you achieve your maximum programming results in the quickest amount of time you can. Keep in mind that this language provides the programmer with many abilities, and this book may refer to things in terms of gameplay programming I hope you are also familiar with.

Although I provide you with the information and some basic skills to practice, it is up to you to put these skills into practice. I highly recommend you find a strict routine to abide by that involves research, study of the material and practice in your own virtual environment that familiarizes you with these skills. As commonly known, you can guide a horse to the water but you can't force it to drink!

I can give you the knowledge and resources but it will be up to you to take on the challenge of synthesizing and using them to create your own programming and codes.

With an object-oriented and structured language such as C#, I think you will find it easy to learn and user-friendly. This high-level language builds on and, in some ways, complements the languages of C, C++ and even Java. However, even if you are a master of these languages, this new task will have just a steep of a learning curve as any of the other programming languages do! Throughout this process, I encourage you to use your own knowledge and instincts to apply the concepts and strategies in this book. Remember, everyone is a beginner before they are a master!

Chapter 1: What is C#

You have come across and purchased this book in order to learn how you can become a C# programmer.

Throughout this book, you will learn basic rules of how to apply C# within your own programming projects. C# (pronounced as “see sharp”) is an object-oriented computer programming language that was both designed and developed by Microsoft.

This is a fairly new language and is considered to be very functional and object-oriented in nature. Although C# is most certainly not the same language as C or C++, there are definitely some similarities between the languages and it could even be considered helpful to have an insight or knowledge and experience in either or both of those languages.

Just as with all programming languages, having knowledge of how they work on a basic and functional level is helpful. All languages have one thing in common, they are working towards telling the computer what and how to set up systems, databases or how and when to run and execute programs.

This particular language has roots stemming from a programmer, Anders Hejlsberg, who was a member of the development team of .Net Framework for Microsoft^[1]. Although some people tend to criticize C# as being somewhat of a “rip off” of Java, you will see throughout the course of this learning manual that (if you have a knowledge of Java that is) there are distinct differences in the content and structure of the language that make it different.

With a design that was formulated for an environment that has the ability to use high-level languages and consists of executable code (also referred to as Common Language Infrastructure or CLI) it appeals to many who are interested in this easy-to-learn language.

It boasts efficiency as one of its objects, so there are quite a few advantages to using it. Throughout this book, I will introduce you to how this object and component-oriented language can be easily acquired as a new language for you to use.

Along with being a structured language, you can count on efficient results from your programming and coding within the C# language. I will assist you with optimizing your performance, in order to obtain your desired output from the work you put into programming and coding in this language.

Although C# was developed by using the Microsoft platform, the language itself is not just limited to it. Another great concept of C# is that this versatility allows you to take the object-oriented language to different platforms to create multidimensional and available programs and software.

As I am sure you are familiar with other languages, certain aspects of languages benefit certain desired results. There are many benefits, that will be covered in this guide to learning C#, that assist a user with portability, typing, meta programming, property, memory access and methods and functions—among an array of other positive attributes we will discuss and show examples of in the coming chapters.

This language has only been around, in official versions, since 2002, and therefore could be considered a newer programming language. However, do not let that lead to you believe it cannot and does not compete with other languages, such as Java, that have been around a little longer.

Developers of this language really aimed to improve upon and add different features, initially designed specifically for the .Net framework but now have been expanded, and were successful in this with many features—one many tend to like is the Syntax used.

There is a possibility for this language to be used to focus on gameplay. It is important to gamers that, not only is the game interesting and holds one's attention, they also want the experience itself to be enjoyable.

As you are probably somewhat of a programmer yourself (or at least appreciate the capabilities of computers), after reading this book I expect you will have a great understanding of just how hard gaming programmers work in order to create the experience you have as an end result.

Although I do not think reading this book alone will allow you to create a masterful game (you need a lot of practice as well!)—I do believe that you will understand concepts to a greater degree that are essential when you are dealing with the many components of graphic programming and design. Keep in mind, throughout the book, that you are creating and fine-tuning a potentially endless world through your coding and programming skills in this language. The object-oriented component really helps to reinforce this idea.

In the coming chapters, you will learn both how and why C# will be very useful to you and how you can easily pick up and become fluent in all the basics within less than a month.

With a standard library that rivals any of the other languages, by the end of this book you will know how to find the source code you may need and execute it to produce

fairly consistent and exquisite results. Another positive feature of this language is the Boolean conditions, which are favored by many, along with the Windows integration that allows you to access one of the most popular operating systems with ease.

As a final note on the very simplistic explanation of what C# is in this initial chapter, I just want to note a few general concepts of programming and coding that are good to keep in mind throughout this process.

First of all, remember that computers do not have intuition. This means that they will execute tasks exactly as you tell them. A computer will not infer a message from your code, as a human might. Always keep this in mind when you are having any struggle.

Also remember that a simple mistake in spacing or a wrong letter can cause a whole lot of headache. If you come to, what may seem like, an impossible impasse—perhaps you need to take a quick break and come back to look at your coding again.

Sometimes we cannot see our own mistakes when we have been staring at a screen for a long period of time.

Finally, remember that you are not the first person to have a struggle with this language and will certainly not be the last. The last chapter of this book will be a guide to further helpful resources and hints as to address any struggles I foresee you having in the future as you begin to learn this language.

This book sets up reasonable expectations for you to learn C#. I encourage you to try to learn in that exact manner. It is important for you to not overwhelm yourself with too much content in a short period of time.

Conversely, don't allow too much time to lapse before coming back to your work. You may forget subtle nuances that may drive you crazy. Allow yourself to have reasonable expectations, which this book is designed to do.

Lastly, enjoy yourself! Allow yourself to take pride in the new skill you are learning and get ready for a crash course in basic C# mastery!

Chapter 2: Detailed Overview

Since you are a beginner, we want to focus on building a strong foundation that gives you a full picture of the basics of C#.

Essentially, by the end of the first week you will have a firm grasp on the environment in C#, the program structure, basic syntax and data types, type conversion, variables, constants and operators. If you are unfamiliar with the terminology or methodology used, I will try to give a quick, basic review of the terms. If this does not satisfy jogging your memory, you may need to find a quick refresher from Google or another resource. I would also like to recommend a free interactive shell program you can try out code on - https://www.tutorialspoint.com/compile_csharp_online.php, this should help you complete any training exercises included in the book.

Environment

As seen, C# was designed to be integrated with Microsoft and .Net Framework by the developers. As one would guess, this is a large influence on the environment you find within C#. An important part of understanding C# comes from the understanding you have of how the two interact with each other. First, here is a brief synopsis of .Net framework to help you better understand how the two interact.

As the latter part of the name framework suggests, the .Net framework is a platform in which one can run applications. The .Net framework is a platform for software development developed by Microsoft. The framework was created for development of applications that can run on the Windows platform. The first version was announced in 2002 and given the name .Net framework 1.0. Since then, a lot of updates have been made on the framework up-to-date.

We can use the .Net Framework for creation of both web-based and form-based applications. We can also use it to develop web services. Other than C#, the .Net framework also supports the visual basic programming language. This means that the programmer has the option of choosing the language in which to develop their application in.

There are three different types of applications that the framework can assist you with writing—web services, web applications and windows applications. When one is using the framework applications, it can actually be used by a variety of programming languages, as it is a multi-platform application. The multitude of languages, from Virtual Basic, Jscript, C++ and COBOL, also have the ability to communicate with each other within the framework. Another great attribute of this framework is the extensive code library that can be used by a language, for our purposes we will be talking about C#.

One of the great things about any language or framework is the potential for a library of codes you can utilize. For the .Net framework and C#, there is a huge library that you can access and utilize for your coding projects. One advisory statement I always give people is that although these libraries are excellent resources, be sure you understand the hows and whys of the coding itself to ensure that you are able to fix an issue, should it arise.

Microsoft itself also offers a great tool for its users in order to assist their development in their C# programming projects.

There are two excellent ones that the company even offers for free, great resources for writing any kind of application, from simple to complex. The names of the two I would recommend (because of their free price!) are Visual C# 2010 Express and Visual Web Developer.

If these options are not something you feel comfortable with or enjoy using, you can always use a text editor in order to write the source code you may need. I recommend the other options because I find them to be a step up in helpfulness from a simple code editor. However, lots of people prefer that method—it's really up to you and the comfort level you feel with a program. Test them out and see which methodology works better for you.

You can download the two Microsoft programs I spoke up from a website called Microsoft Visual Studio.

Finally, you can also run C# on Linux or Mac OS, which is a great ability to code across different platforms. However, in order to do so, you obviously need a non-Microsoft framework to do so on your different operating system. I recommend “mono”—which is an open source option for use on both Linux and Mac.

With the basics of the environment, go download or open the programs you would like to get familiar with to start your coding process. Play around with the software—or you can sleep on your new knowledge and you can pick up your project for the next phase!

IDE (Integrated Development Environment)

An IDE is a program that allows you to write your program. Most .Net developers prefer using Visual Studio Community as the IDE. It allows you to create, debug and run your applications. This IDE can be used for development of both web-based and form-based applications. To use this IDE, you must download and install it on your computer. You can find it in the URL given below:

<https://www.visualstudio.com/downloads/>

You are allowed to choose between the Visual Studio Community Forum, which is free, and Visual Studio Professional Edition, which comes with a 30 day trial period, and after the expiry of that period, you must pay for it.

Once the download is complete, double click the setup file to begin the installation of visual studio. Follow the on-screen instructions until the installation of visual studio is complete.

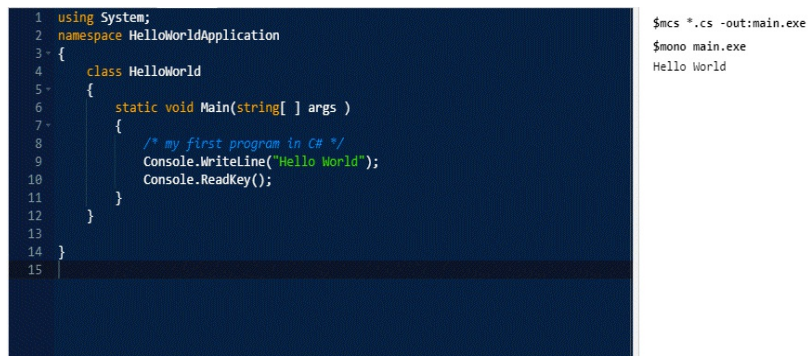
After that, you will be ready to start creating your own C# applications.

Program Structure and Basic Syntax

Now that we have established a little about how the environment was created for C#, let's look at the actual structure and how you will interact with programs through the syntax of C#.

When we are looking at how C# is structured, we are looking at 7 basic components that make up the program. These are that there are class methods and attributes, namespace declaration, a class, a main method, comments and statements and expressions. Now, what do these look like for you in terms of your programming and coding? Why don't we take this "Hello World" example and break it down into parts to see this in action.

```
using System[2];
namespace HelloWorldApplication
{
    class HelloWorld
    {
        static void Main(string[ ] args )
        {
            /* my first program working with the wonderful C# */
            Console.WriteLine("Hello World");
            Console.ReadKey();
        }
    }
}
```



```
1 using System;
2 namespace HelloWorldApplication
3 {
4     class HelloWorld
5     {
6         static void Main(string[] args)
7         {
8             /* my first program in C# */
9             Console.WriteLine("Hello World");
10            Console.ReadKey();
11        }
12    }
13 }
14
15
```

```
$mcs *.cs -out:main.exe
$mono main.exe
Hello World
```

[3]

So hopefully this looks somewhat familiar to you and, although you may not quite understand the meaning, you know all the symbols and spaces are important.

When we are using C#, the first line is always going to consist of using System, although most of the time you will see more than one statement with the first key word being using.

The next line in C# will be used as a declaration of the namespace. In case you are unfamiliar, we should think of the terminology of namespace as a descriptor of the specific group or collection of classes.

In our Hello World example, we are looking at that phrase as the class (HelloWorld). Following the name space line, you will come to the class declaration. The class is going to inform you of the method of the class.

It is a typical characteristic of methods that there are multiple within one singular class. It is important to be familiar with the methods of a given class because you can then be more aware of the behavior of the class you are dealing with. In the Hello World example, we are looking only at one single term, or main method.

Each class has a main method that you can find in the line following the class declaration. Essentially, when you are looking at this line you can expect the main method to tell you what is going to happen when the code is executed. This point is often also referred to as the entry point in C# programs. When you see the `/*...*/` in the following line, you are looking at code that will be ignored by the actual compiler and instead is added in the program under the comments.

The last two phrases we are looking at in this example are `Console.WriteLine("Hello World")` and `Console.ReadKey()`. These lines are telling the system to both display your message and control how and when the program runs and closes. These are obviously both important parts of how and why the program would work and function and you want to make sure of a couple of key components that could affect your coding, just due to similar grammatical type errors you may find in the English

language.

To run the code, just click the Run/Execute button or press the F5 key and the project will be executed. The code should return:

```
Hello World!
```

The code begins with the *using* statement. What this statement does is that it adds the *System* namespace into the program. C# programs normally have multiple *using* statements.

In the next line, we have a *namespace* declaration. A namespace is simply a collection of many classes. Above, we have the *HelloWorld* namespace in which we have the *Hello* class.

In the next line, we have the class declaration. The *Hello* class has method and data definitions that the program will be using. A class normally has numerous methods. The purpose of a method is to state the behavior of a class. However, our *Hello* class above only has one method, the *Main* method.

In the next line of code, we have the definition of the *Main* method, which marks the entry point for all for C# programs. It is the *Main* method that states what happens after execution of a class.

Next, we have the */* ... */* line. This is a comment, and the C# compiler will ignore the line. The line is only meant to increase the readability of the code by human readers.

Next, we have the *Console.WriteLine(...)* line. *WriteLine* is a method that belongs to the *Console* class that is defined in the *System* namespace. This line will print the *Hello World!* message on the screen.

Lastly, we have the *Console.ReadKey();* statement. This line is for *VS.NET* users. The line will make the program to wait for a key press and prevent the screen from running then closing quickly once the program has been launched from the Visual Studio .NET.

Other than running the program from the Visual Studio, it is possible for you to run a C# program from the command prompt of the operating system. To do this, follow these sequences of steps:

- Open your text editor then add the code given above to it. An example of a text editor is Notepad.
- Save the file with the name *helloworld.cs*
- Launch the command prompt of the operating system then navigate to the

directory you have saved the file.

- Type the command `csc helloworld.cs` on the prompt then press the enter key to compile the code.
- If the code has no errors, the command prompt will take you to the next line then generates the file `helloworld.exe`. This file has executable code.
- Type `hello world` on the prompt to execute the program.
- The output *Hello World!* Will be printed on the screen.

You want to make sure that you note a subtle difference (if you are familiar with Java) that the class name and program file name can be different. Also, C# is both case-sensitive and, rather than ending sentences with a period or other notation, they are ended using a semicolon. Also, always remember that the main method is the point at which the program execution starts.

These four components are good to note in order to avoid annoying and tedious problem-solving when mistakes occur with your coding.

Play around with the above example within the virtual environment. Now we will discuss a little about the syntax of C# and how some of the concepts we have talked about with the structure really develop and exemplify the syntax. For starters, the `/*...*/` that allows you to comment on a program is an important feature to remember.

In addition to this, it is good to remember that when naming a class, the first character in the name can only be a letter, not a numerical digit. Although you can make up the class itself following the initial digit with letters and numbers, always remember this and that symbols are always not allowable in the naming of a class. Finally, the name cannot contain a C# keyword. There are two different categories of keywords, contextual and reserved, below I will list them all in their respective categories.

Contextual Keywords— `get`, `partial`, `from`, `orderby`, `let`, `dynamic`, `set`, `join`, `descending`, `ascending`, `into`, `select`, `alias`, `group`, `remove`, `global` and `add`.

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

Important Note: Make sure that all of your programs have formatted codes. This way, you can guarantee the readability of your codes.

A Breakdown of Different keywords^[4] in C#

Keywords are special predefined reserved words and are each assigned with a unique meaning. These keywords can be organized into categories useful for better understanding. Below is a list of keywords categorized into different types.

1) Keywords for class, method, field and property

- abstract
- extern
- internal
- new
- const
- override
- protected
- private
- public
- sealed
- readonly
- static
- virtual
- void

2) Keywords for type conversions

- explicit
- implicit
- as
- is
- operator
- sizeof

- typeof

3) Keywords useful for program flow control

- if
- else
- for
- foreach
- in
- case
- break
- continue
- return
- while
- goto
- default
- do
- switch

4) Keywords used for built in types and enumerations

- bool
- char
- class
- byte
- decimal
- enum
- double
- float
- interface
- long

- int
- object
- sbyte
- short
- string
- uint
- struct
- ulong
- ushort

5) Keywords used for exception handling

- try
- catch
- throw
- finally
- checked
- unchecked

6) Keywords used as literals, method passing parameters

- true
- false
- null
- this
- value
- out
- params
- ref

7) Keywords useful in function pointers, object allocation, unmanaged code

- delegate

- event
- new
- stackalloc
- unsafe

How to Format Codes Properly

Some rules to follow when formatting your codes include:

- You may indent methods within the class definition. C# allows you to use any number of tab characters (i.e. the character you'll get after hitting the Tab key of your keyboard) when formatting source codes.
- You should indent the method's contents within its own definition.
- Place the opening brace (i.e. {) right under the class or method to which it belongs. Also, it would be best if it will be the only character on its line.
- Make sure that the closing brace (i.e. }) is vertically aligned with the opening brace.
- The names of your class should begin with an uppercase letter.
- The names of your variables should start with a lowercase letter.
- The names of your methods should begin with an uppercase letter.
- Indent codes that are written inside another code.

Class Names

In C#, each program has one or more class definitions. Programmers often store each class definition inside a separate file that corresponds to the class's name. Also, the extension of these files should be ".cs". You can compile different classes into a single file without experiencing any technical problems. However, you'll have a hard time navigating your codes.

More Information about C#

The C# programming language is advanced, modern, multipurpose, and object-oriented. It has some similarities with other languages such as C, C++, and Java. The main difference is that C# offers easier programming and simplified syntaxes.

A C# program has one or more .cs files, which hold data types and class definitions. You need to compile those .cs files to get executable codes (i.e. files that end with .dll or .exe). For instance, if you will compile the HelloCSharp program, you'll get an

executable file named “HelloCSharp.exe”.

The Things You Need to Create C# Programs

Before you can write programs using C#, you need two important things—a text editor (e.g. Notepad) and the .NET framework. The text editor will help you in writing and editing codes. The .NET framework, on the other hand, will help you in compiling and executing your programs.

What is .NET?

Basically, .NET is a framework designed for the development and execution of computer programs. Most of the modern Windows systems have .NET as a built-in framework. Thus, you won’t have to install any software onto your computer if you are using Microsoft’s modern operating systems.

If you are using an old Windows OS, you need to download the .NET framework first. You may go to this [site^{\[5\]}](#) and get the most recent version of .NET.

Important Note: Make sure that your computer has .NET before reading the rest of this book. Otherwise, you’ll have problems compiling and executing the sample programs that you’ll see later.

Text Editors

You should use a text editor to write, edit, and save source codes. You can use any text editor installed on your computer. If you don’t want to download additional programs, you may simply use “Notepad” (i.e. the pre-installed text editor of Windows computers).

Compiling and Executing Programs

The actions you need to undertake include:

1. Generate a “.cs” file and name it as “HelloCSharp”.
2. Write the source code inside that .cs file.
3. Compile the code to get a .exe file.
4. Run the resulting program.

Turn on your computer and let’s start writing some codes.

Important Note: This eBook assumes that you are using a Windows computer.

Chapter 3: Demystifying Data Types

Now let's talk about data and the different types you will see in C#, which we have three basic categories of. These categories are value, pointer and reference types. If you are familiar with programming, you are definitely familiar with the different types of values that can be associated with data types.

Attributes

Data types have the following attributes:

- Name (e.g. int, char, bool, etc.)
- Size (e.g. 1 byte)
- Default Value (e.g. 1, 2, 3, etc.)

The Different Types of Data in C#

C# supports the following data types^[6]:

- Boolean
- Characters
- Objects
- Floating-Point
- Decimals
- Integers
- Strings

Since these data types are pre-installed into the C# language, programmers refer to them as “primitive types”.

Value types of data are, as the name would suggest, data types that consist of the value themselves. These are terms like Boolean data, byte or int. A good test to run within your virtual environment is to get the size of a given data type using the term `sizeof(type)`, only replace type with the data type you wish to know the size of. A complete list of the value data types are decimal, double, bool, float, char, byte, long, sbyte, short, uint, int, ushort and ulong.

Within the reference type of data, we are looking at (you probably guessed it!) a

reference to a type of data. These types of data are able to tell you where memory is located. There can even be multiple references to multiple variables. His type of data, we are looking at string, dynamic and object as our key words to describe the reference data types.

The last type of data, pointer, points to a different variable in order to direct you to the appropriate location. In order to reinforce that you do indeed know what a pointer is, here is a generic sample of what a declaration of a pointer variable should look like

—
type *var-name;

Go ahead and play around with the variables in your own dictionary and see what kind of output you can get for these. There is one other important phrase I think you should be aware of in your programming and coding. This is when you use the /unsafe command—

```
csc /unsafe darkprog.cs
```

In the above example, the program we would be specifying as unsafe is “darkprog.cs.

C# is a strongly typed language; hence it expects you to state the data type any time you are declaring a variable. Let us explore some of the common data types and how they work:

1. bool- this is a simple data type. It takes 2 values only, True or False. You need to use “bool” (i.e. a C# keyword) to declare this data type. Boolean values only result in “true” or “false”. In the C# programming language, Boolean values are automatically set as “false”. Programmers use this data type to store the result/s of logical statements. It is highly applicable when using logical operators like *if* statement.
2. int- this stands for *integer*. It is a data type for storing numbers with no decimal values. It marks the most popular data type for numbers. Integers also have several data types within C#, based on the size of the number that is to be stored.
3. string- this is used for storage of text, which is a sequence of characters. C# strings are immutable, meaning that you cannot change a string once it has been created. If you use a method that changes a string, the string will not be changed but instead, a new string will be returned.
4. char- this is used for storage of a single character. You need to use “char” (i.e. another C# keyword) to declare this type. Write your “char” values within a pair of apostrophes (e.g. “I”).

5. float- this is a data type used for storage of numbers that have decimal values in them.

Objects

Programmers regard objects as the most special type of data in the C# language. Basically, objects serve as the parent of other data types within .NET. This data type, which requires the “object” keyword during its declaration, may accept any value from other data types. You may think of objects as addresses that point to certain parts of your computer’s memory.

Real Type – Floating Points

In the C# language, “real type” refers to the actual numbers you’ve learned in mathematics. A floating-point real data type can either be “float” or “double”.

1. Float – Programmers call this subtype “single-precision real integer”. The default value of a float number is 0.0F (the “f” at the end is not case-sensitive). The letter “f” indicates that the value is a “float”.
2. Double – Programmers use the term “double-precision real integer” when referring to this subtype. Its default value is 0.0D (i.e. the letter “d” at the end is not case-sensitive.). C# automatically tags real numbers as “double” so you don’t really need to add “d” at the end of the value.

Real Type – Decimals

This programming language supports “decimal arithmetic”, a mathematical approach that uses decimal values to represent numbers. This approach preserves its accuracy regardless of the values it works on.

C# uses the “decimal” data type to represent this kind of real number. This data type is 128 bits long and is accurate up to the 29th decimal value. Because of its excellent accuracy, these are used by programmers for financial computations (e.g. payments, taxes, interest, etc.).

Integers

This data type consists of 8 subtypes, which are:

Ø SBYTE – An sbyte is a signed integer that is 8 bits long. That means it can contain up to 256 values (i.e. 2⁸). Additionally, sbytes can be either negative or positive.

Ø BYTE – This subtype contains unsigned integers that are 8 bits long. It is almost identical to sbyte. The only difference is that byte can never be negative.

- Ø SHORT – These integers are signed and 16-bits long.
- Ø USHORT – These are signed integers whose length is 16 bits.
- Ø INT – This is probably the most popular data type in C#. Programmers use this data type because it is perfectly compatible with 32-bit processors and big enough for typical computations. It is a signed integer whose length is 32 bits.
- Ø UINT – This 32-bit integer is unsigned. Thus, it can only be positive.
- Ø LONG – This integer is 64 bits long. It can be either positive or negative.
- Ø ULONG – This is similar to “long” integers. The only difference is that “ulong” can only assume positive values.

Strings

A string is a set or sequence of characters. C# requires you to use “string” (i.e. a keyword) when declaring values that belong to this data type. You have to write each string between a pair of quotation marks. This language allows you to perform various operations on strings (e.g. concatenation, character replacement, character search, etc.).

The *sizeof()* method allows us to know the size of a variable or data type. The size is returned in the form of bytes. Consider the following example^[7]:

using System;

namespace TypeApp {

class IntType {

static void Main(string[] args) {

Console.WriteLine("Size of int: {0}", sizeof(int));

Console.ReadLine();

}

}

}

The code should return:

```
Size of int: 4
```

Which means that an integer takes a storage size of 4 bytes?

Chapter 4: Working with Variables

You can use a variable to store, retrieve, and modify changeable data. As a programmer, you need to use variables^[8] in storing and processing information.

The Characteristics of a Variable

A variable has the following characteristics:

- Name (also known as “Identifier”)
- Data Type
- Value (i.e. the information you want to store).

Variables are sections of computer memory that have a name. They store values and allow a computer program to access their contents. You may place a variable inside the stack (i.e. the working memory of your computer program) or within the program’s dynamic memory.

Characters, Booleans, and Integers are known as value types since they keep their data within the program’s stack. On the other hand, strings, arrays, and objects are known as “reference types” because they don’t hold the values inside them. Instead, they serve as markers that point to the part of computer memory where the data is stored.

How to Name a Variable

You need to name your variables before using them. The variable’s name serves as an identifier: it helps the program in locating and specifying the variable. C# allows you to choose variable names freely. However, just like any other language, C# has some rules regarding variable names. These rules are as follows:

- You may use an underscore (i.e. “_”), letters, and numbers to create a variable name.
- You can’t use a number to start the name. Thus, FirstSample is valid while 1stSample is not. camelCase for local variables, such as cost, orderDetail, dateOfBirth, and firstName
- You can’t use C# keywords in naming your variables. This simple rule prevents compilation and runtime errors in your programs. If you really want to use keywords for your variables, you may begin the name using the “@” symbol (e.g. @bool, @int, @char, etc.).

- A meaningful or descriptive name that is neither too long nor too short to identify the information stored in a variable just by looking at it
- Can contain the letters a–z and A–Z, the numbers 0–9, and the underscore (_) character—other symbols are not allowed
- No spaces and cannot start with a number

Valid Names

- cost, name, order, order1, _order1, income
- order_Detail, orderDetail, dateOfBirth, hourlyRate, firstName, first_Name, isValid

Invalid Names

- 1 (*number*)
- class, while, if, protected (*keyword*)
- 1order, 1name (starts with a number)

How to Declare a Variable

You should declare a variable before using it. When declaring a variable:

1. Specify the variable's data type (e.g. char)
2. Specify the variable's name (e.g. sample)
3. Set the variable's initial value. This step is completely optional.

The format for variable declarations is:

data_type name = initial_value;

Here are some samples of valid variable declarations:

int year = 2000;

string = yourname;

char = test;

Important Note: You need to end your each of your C# statements using a semicolon. If you'll forget to add even a single semicolon character, your programs will generate undesirable and/or unexpected results.

How to Assign a Value

Just like any other computer language, C# allows you to assign a value to a variable.

This process, known as “value assignment”, requires you to use the equals sign. Programmers refer to this symbol as the “assignment operator”. Write the variable’s name on the left side of the operator. Then, specify the value you want to assign on the other side. Check the following examples:

```
test = x;
```

```
yourname = “John Doe”;
```

How to Initialize a Variable

In programming, “initialization” is the process of assigning a value to a new variable. Thus, you’ll set this “initial value” while declaring a variable.

The Default Value of a Variable^[9]

If you won’t assign any value for your variable, C# will perform an automated initialization (i.e. it will assign a default value to the empty variable). The following tables will show you the default value of each data type:

Data Type	Default Value	Data Type	Default Value
sbyte	0	float	0.0f
byte	0	double	0.0d
short	0	decimal	0.0m
ushort	0	bool	false
int	0	char	'\u0000'
uint	0u	string	null
long	0L	object	null
ulong	0u		

The code given below will show you how to declare variables and assign values.

```
// This is an example
```

```
byte years = 10;
```

```
short days = 31;
```

```
decimal pi = 3.14;
```

```
bool isittasty = true;
```

```
char sample = x;
```

```
string animal = “dog”;
```

Important Note: Lines that begin with two forward slashes (i.e. //) are “comments”. In programming, comments are descriptive text added to the code to improve its readability. Language compilers ignore these lines. Thus, comments won’t affect the functionality of your computer programs.

Creating an identifier

If you look through some of the suggestions that come with Microsoft, you will find that they will recommend that you use a Camel notation when you are working with the variables, and it will recommend that you work with the Pascal notation when you want to work with the methods. Working with the Camel notation will mean that you will keep the first letter of your name as lower case. If you are creating an identifier that has a compound word then you will make sure that the first letter of your second word will start with an uppercase letter:

payment *completePayment*

mathematics *firstClass*

The Pascal notation will take things a bit differently. This notation style requires you to start the first word using an uppercase letter. The first letter in all of the other words in the sequence should also be written in uppercase as well. Some examples of using the Pascal notation include:

WriteLine() *ReadLine()*

Start() *Main()*

When it comes to naming these identifiers, you can work with numbers and underscores as well. But, if for some reason you can't begin the name of these identifiers with a number, you can write out something like seven books, but you can't write out '7books.'

Now, these notations are not required, and you can change them around as much as you would like. But this is considered a proper coding practice if you're working with C# so it is best to follow these rules so the code will work exactly the way that you want it.

As you can see, C# is a pretty easy language to learn. You do not need to worry about a lot of rules that are too complicated to remember like the other programming languages, and yet you will still have a lot of the power and the flexibility to get the most out of coding.

Definite Assignment

C# enforces a definite assignment policy. This means that you will need to initialize the local variable with the value before using it. Suppose the following is written:

```
Console.WriteLine(name);
```

Error: Use of unassigned local variable 'name'

If you try to use a variable that hasn't been declared, your code won't compile. In this case, the compiler will tell you that something is wrong. Trying to utilize a variable minus having a value assigned to it also causes an error.

In order to initialize the variable with a value, you simply have to use the assignment operator "=". The variable name is located to the left side of the operator and on the right side is the value in the following manner.

```
name = "Pirzada"; age = 35; weight = 70; isMarried = true;
```

On declaring a variable with a type, it cannot be redeclared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an **int** and then assign it a Boolean value of True/False.

```
age = true;
```

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

If I try to assign a **string** to an **age** variable, it is declared an **int** datatype.

```
age = "Pirzada";
```

OR

```
age = name;
```

The above statement will give a compile-time error because the **string** value cannot be assigned to an **int** type variable.

You can combine the declaration and initialization statements at the same time on the same line as shown below, which is more convenient.

```
string name = "Pirzada"; int age = 35; int weight = 70; bool isMarried = true;
```

You can also use a mathematical expression.

```
int wowExp = (3 + 2) * 4; Console.WriteLine(wowExp);
```

OUTPUT:

20

The right side is being evaluated and then assigned to the variable on the left. Meaning 3+2 is 5 and 5 * 4 is 20, which will be assigned to the **wowExp** variable.

You can also assign the same value to multiple different variables all at the same time.

```
int a, b, c; a = b = c = 786; Console.WriteLine(a); Console.WriteLine(b); Console.WriteLine(c);
```

OUTPUT:

786

786

786

Using Type Inference

C# 3.0 introduced the implicitly typed variable with the **var** keyword. Now you can declare a local variable without giving an explicit or real type. The variable still receives a type at compile time, but the type is provided by the compiler.

Actually, the **var** keyword instructs the compiler to infer the type of variable from the expression on the right side of the initialization statement. The compiler is given the task to determine and assign the suitable type.

```
var variableName = variableValue; // Syntax
```

var is optional, and it's just for convenience.

Let's use the same variables as above.

```
var name = "Pirzada"; var age = 35; var weight = 70; var isMarried = true;
```

Now the variables are **implicitly typed local** variables, meaning that you don't have to explicitly specify the type. The compiler is tasked to determine and assign the suitable type. A **string** variable is indicated by double quotes, **char** variable is denoted by single quotes, and a **bool** type is denoted by true and false values.

Suggestion: Using **var** is convenient, but use it only when the type is obvious from the right side of the assignment.

Double is used to denote a literal number with a decimal point unless you add the **M/m** suffix to indicate a **decimal** variable or the **F/f** suffix to indicate **float** variable.

```
var value = 24899.45m; var number = 20.8f;
```

To initialize **number**, you need to add **f** as a suffix after 20.8 to explicitly tell the compiler to change 20.8 to a **float**. Similarly, to initialize **value**, you need to add **m** as a suffix to change 24899.45 into a **decimal** type.

A few rules you need to follow:

- **var** can only be declared and initialized in a single statement. Otherwise, the compiler doesn't have anything from which to infer the type.
- **var** is not supposed to be utilized on fields at class scope.

- The initializer cannot be null and must be an expression.
- Multiple, implicitly typed variables cannot be initialized in the same statement.
- An object cannot be initialized unless a new object is created in the initializer.

Default Value Expressions

Default value expressions are especially useful in combination with generic types when you don't know in advance what the default value for the given type will be.

default(T) Expression: Old Way

A default value expression **default(T)** returns the default value of a type **T**, where **T** is a reference type or a value type.

T variableName = default(T); // syntax

Example:

```
var name = default(string); var age = default(int); var weight = default(int); var
isMarried = default(bool);
```

default literal: New Way

The **default literal** is a new feature in C# 7.1 that is used to get the default value of the specified data type when the statement is executed. This feature works for value types as well as reference types.

Note: **default literal** is equivalent to default(T) where T is the inferred type.

```
string name = default; int age = default; int weight = default; bool isMarried =
default; Console.WriteLine(name); Console.WriteLine(age);
Console.WriteLine(weight); Console.WriteLine(isMarried);
```

In the output below, a null value is printed as a blank line.

OUTPUT for default literal & default(T):

0 0 False

new Operator:

Using the **new** operator invokes the default constructor of a value type. This allows you to create a variable using the **new** keyword, which automatically sets the variable to its default value.

Note: All values in C# are an instance of a specific type.

```
var age = new int(); var isMarried = new bool(); Console.WriteLine(age);  
Console.WriteLine(isMarried);
```

OUTPUT:

0

False

Chapter 5: What is Type Conversion?

In general, an operator works on arguments that belong to the same type. However, the C# language offers a wide range of data types that you can use for certain situations. To conduct any process on variables that belong to different types, you should convert one of those variables first. In C#, data type conversion^[10] can be implicit or explicit.

Each C# expression has a data type. This data type results from the literals, variables, values, and structures used inside the expression. Basically, you might use an expression whose type is incompatible for the situation. When this happens, you'll get one of these results:

- The program will give you a compile-time error.
- The program will perform an automated conversion. That means the program will get the right expression type.

Converting an “s” type to “t” type allows you to treat “s” as “t” while running your program. Sometimes, this process requires the programmer to validate the type conversion. Check the examples below:

- Converting “objects” to “strings” will need verification during the program's runtime. This verification ensures that you really want to use the values as strings.
- Converting “string” values to “object” values doesn't need any validation. That's because the “string” type is a branch of the “object” type. You can convert strings to their “parent” class without losing data or getting any error.
- You won't have to perform verification while converting int values to long values. The “long” data type covers all of the possible values of the “int” type. That means the data can be transformed without any error.
- Converting “double” values to “long” values involves validation. You might experience loss of data, depending on the values you're working on.

Important Note: C# has certain restrictions when it comes to changing data types. Here are the possible type conversions supported by this language:

Explicit Conversion

Use this conversion if there's a chance of information loss. For instance, you'll experience information loss while converting floating-point values to integer values (i.e. the fractional section will disappear). You might also lose some data while converting a wide-range type to a narrow-range type (e.g. long-to-int conversion, double-to-int conversion, etc.). To complete this process, you need to use the "type" operator. Check the examples below:

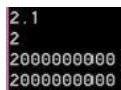
```
class Example
{
    static void Main()
    {
        double yourDouble = 2.1d;
        System.Console.WriteLine(yourDouble);

        long yourLong = (long)yourDouble;
        System.Console.WriteLine(yourLong);

        yourDouble = 2e9d;
        System.Console.WriteLine(yourDouble);

        int yourInt = (int)yourDouble;
        System.Console.WriteLine(yourInt);
    }
}
```

If compiled and executed correctly, that program will give you these results:



```
2.1
2
2000000000
2000000000
```

Implicit Conversion[\[11\]](#)

You can only perform this conversion if data loss is impossible. This conversion is

referred to as implicit because it doesn't require any operator. The C# compiler will perform implicit conversion whenever you assign narrow-range values to variables that have a wide-range.

String Conversion

C# allows you to convert any data type to "string". The compiler will perform this conversion process automatically if you will use "+" and at least one non-string argument. Here, the non-string argument will become a string and the "+" operator will produce a new value.

The implicit type conversion is performed to do actions like conversions that are from a base class to a derived class. These are performed in what is referred to as a "type safe" manner. The other type of conversion, explicit, require a cast operator in order to complete their function. There are 16 different functions within C# that are able to be used as a built in type of the conversion methods. Here are some examples you should practice becoming familiar with within the C# environment.

- ToChar- allows you to take a type and convert it to a single Unicode character
- ToByte- allows you to get a byte by converting it from a type
- ToDateTime- allows you to get a date time structure by converting a type (either string or integer)
- ToBoolean- allows you to get a Boolean value from converting a type
- ToDouble- allows you to get a double type from converting a type
- ToDecimal- allows you to get either an integer or decimal type converted from a floating point
- ToInt64- allows a 64 bit integer to be converted from a type
- ToInt32- allows a 32 bit integer to be converted from a type
- ToInt16- allows a 16 bit integer to be converted from a type
- ToSbyte- allows a signed byte type to be converted from a type
- ToString- allows a string to be converted from a type
- ToSingle- allows a small, floating point number to be converted from a type
- ToUInt64- allows an unsigned bit integer to be converted from a type
- ToUInt32- allows an unsigned long type to be converted from a type

- ToUInt16- allows an unsigned int type to be converted from a type
- ToType- allows a specified type to be converted from a type

For your exercise today, I encourage you to go find programming that does each and every one of these functions/conversions in order to get more familiar with what they involve and how they can assist you in your own programming.

Remember that although your coding may be long, your output will be short—but should always produce the answer you were trying to achieve from the coding in question that you have done.

Chapter 6: The Need for Operators

In this chapter, we will take some time to talk about the operators^[12] that you can work with inside the C# language.

By reading this material, you'll know how these operators work and how you can use them in your source codes. These operators are pretty simple to understand and use but this doesn't mean they're useless, in fact, they can help make sure that your code will perform the tasks that you require it to complete. Without using some of these operators, your program will lack functionality, and won't work all that well. These operators may be simple to understand, but they really add a ton of power to enhance the overall capability of your code.

Operators are important no matter which type of coding language that you are working with. We will go into detail on the different operators separately so that you can understand how each one is supposed to work and how you can work with each one to fulfill a variety of purposes and accomplish different tasks.

Operators – The Basics

Operators help you in processing objects and data types. They accept inputs and operands to produce a result. The C# language uses special characters (e.g. “.”, “+”, “^”, etc.) as operators. Also, C# codes can take up to three different operands.

The Different Types of Operators

C# programmers divide operators into the following types:

Assignment Operator

While there are some other options that you can go with when it comes to working with the assignment operators, this is the most common way that you will use these operators. In addition, there are a few different types of assignment operators that you can work with, but the most common one is the equal sign. Some of the operators that you can to work with inside of your C# code includes the following:

- =

The function of this operator is to allow you to perform simple assignment operations. It can assign the value to a variable that you are working on at that time. For example, writing `int sample = 100` will tell the program that you want to assign 100 to the variable that is called 'sample.' It won't perform any additional tasks on

this variable or on the value involved.

- +=

This is the additive assignment operator. Its function is to add up the values of your two operands and then assign the sum to your left-hand operand.

- -=

Programmers will often refer to this as the ‘subtractive’ assignment operator. Its function is to subtract the value of the operand on the right side from the one on the left side and then assign the difference to the left-hand operand.

- *=

The function of this operator is to multiply the values of each operand and then assign the product to the left-hand operand.

- /=

The function of this operator is to divide up the two variables and then take the result and assign it to the variable on the left.

When you are working with the assignment operators, make sure that you have both operands that belong to the same type of data. If you find that the various operands you are using are not compatible, some issues could arise which can result in an error in the program if you try to execute it and will require additional time and effort to fix.

In C#, you need to use the equals sign (i.e. =) to assign values. Here’s the syntax that you need to use when assigning a value:

name_of_operand = expression, literal, or second_operand

Here are some examples:

```
int y = 99;
```

```
string hello = “Good morning.”;
```

```
char sample = ‘z’;
```

Important Note: C# allows you to cascade (i.e. use several times within a single expression) the assignment operator. That means you won’t have to enter multiple equals signs if you are creating variables of the same data type. The examples given below will illustrate this rule:


```
int d = 1, e = 2, f = 3;
```

```
string hi = "Good Morning.", bye = "Farewell.", thanks = "Thank you.";
```

Logical Operators

A logical (also known as “Boolean”) operator accepts and returns Boolean values (i.e. true or false). Here are the four logical operators of C#:

- “&&” – Programmers refer to this operator as “Logical AND”. It will give you “true” if both of the operands are true.
- “^” – This is known as the “Exclusive OR” operator. It will give you true if one of the operands is true.
- “||” – This is the “Logical OR” operator. You will get true if at least one of the operators is true.
- “!” – This operator, known as the “Logical Negation” operator, reverses the value of an operand.

Study the code to understand the use of these operators^{[\[13\]](#)}:

```
class Test
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        bool x = true, y = false;
```

```
        System.Console.WriteLine(x && y);
```

```
        System.Console.WriteLine(x || y);
```

```
        System.Console.WriteLine(!x);
```

```
        System.Console.WriteLine(x ^ y);
```

```
// This is an example.
```

```
}
```

```
}
```

Once you have compiled and run this code, you will see the following results:

```
False
True
False
True
```

Arithmetic Operators

These operators help you in performing math operations. This category consists of the following operators:

- “+” – Use this operator to perform addition using two operands together or values (e.g. *int x = 1 + 1*)
- “-” – This operator allows you to perform subtraction in your codes. It deducts the value of the right operand from that of the left operand (e.g. *int x = 2 - 1*)
- “*” – With this operator, you can multiply the values of two operands (e.g. *int r = 6 * 6*)
- “/” – Use this operator to divide number values in your C# codes. Basically, “/” divides the value of the left-hand operand by that of the right-hand operand.
- “++” – This is known as the “increment operator”. It increases the value of an operand by one. Unlike the ones discussed above, this operator works on a single operand. Also, you may write it before or after the operand. (e.g. *99++*, *1++*, *++5*, etc.)
- “--” – This operator is the exact opposite of the increment operator. You can use it to decrease an operand’s value by one. You may write it before or after the operand you want to modify.
- “%” - This is often called the remainder or is commonly known as the ‘modulo’ operator. Its function is to divide the left-hand operand by the right operand and then returns the remainder.

The code given below will show you how these operators work:

```
class Test
```

```
{
```

```

static void Main()
{
    double d = 100, e = 3;

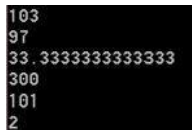
    System.Console.WriteLine(d + e);
    System.Console.WriteLine(d - e);
    System.Console.WriteLine(d / e);
    System.Console.WriteLine(d * e);
    System.Console.WriteLine(++d);
    System.Console.WriteLine(--e);

    // This is an example.

}
}

```

Once you compile and execute that code, you'll see this on your command prompt:



```

103
97
33.33333333333333
300
101
2

```

Binary Operators

These operators work on binary numbers. In the IT world, all of the information is expressed as a sequence of zeros and ones. That means you need to master binary operators if you want to be a successful programmer.

A binary operator works exactly like a logical one. Actually, you may think that binary and logical operators use different inputs but conduct the same processes. A logical operator uses Boolean values to produce results. A binary operator, on the other hand, works on numbers presented as binary digits. Here are the binary operators you'll encounter while using C#:

- “&” – This is called “Binary AND”. It indicates the position/s where both of the operands have “1”.
- “^” – Programmers refer to this operator as “Binary Exclusive OR”. It will place “1” in each position where the values are different.
- “|” – This is the “Binary OR” operator. It indicates the position/s where any of the operands has “1”.
- “~” – This acts as the negation operator for binary values. Just like “!”, it reverses the current value of a variable.
- “<<” – This is the “Binary Left Shift” operator. It moves the bits of a binary number to the left. This movement depends on the value that you’ll assign (e.g. $4 \ll 1$, $4 \ll 3$, $4 \ll 2$, etc.).

Relational/Comparison Operators

The next type of operator that you can use is known as the relational operator. These operators are very useful because they allow you to compare the values of two of your operands. Each of these operators gives a Boolean value as the final output. Because of this function, these operators are the best to use when you want to create some conditional statements. We will have a look at some of the relational operators that you can work with to make your C# code, but for these examples assume that ‘e = 150’ and that ‘d = 100.’

- “==” – This is the Equality operator. It checks whether the two operands are equal (e.g. $x == y$). The function of this the operator is to allow you to check the equality of two values. If the two values end up being equal, the operand will tell you it is true. Otherwise, the operand will tell you it is false. For example, saying the $d == e$ would show up as false.
- “>” – The function of this operator is to check whether the operand on the left is greater than the operand on the right. If it is, then the operator will tell you it is true. For example, saying that $e > d$ would be true.
- “<” – This is actually less than an operator, it allows you to check whether the operand on the left side is less than the operand on the right side. It will give you “true” if the left-hand operand’s value is less than that of the right-hand operand (e.g. $y < x$).
- “>=” – With this operator, you can determine whether the value of the left-hand operand is greater than or equal to that of the right-hand operand. The function of this operand is that it will say the value of the operand on the left side is truly greater than or equal to, the operand on the right side.

Otherwise, it will tell you the statement is false. For example, saying that `e >= d` evaluates as true.

- “<=” – This operator will give you true if the value of the left-hand operand is less than or equal to that of the right-hand operand.
- “!=” – The function of this operator is to allow you to test the inequality of two values/operands. If the values end up not being equal, it will tell you this is true. For example, `e != d` would result in the operator saying it’s true.

When you are working with these relational operators, always remember that you will get a Boolean result each time. What this means is that the answer you get will be either true or false. You also have to check if you are using two equal signs when you are working with the equality operator. If you end up getting these operators mixed up with your assignment operator, you will probably get an error message, and that program will not work out the way that you want.

Conditional Operator

C# users refer to “?” as the conditional operator. Basically, this operator uses the result of a Boolean expression to determine which statement should be processed. This is called “ternary operator” because it works on three operands. Here, the first value should be Boolean, while the remaining values need to belong to the same data type (e.g. characters, strings, numbers, etc.).

The syntax of this operator is:

first_operand ? second_operand : third_operand;

Here’s how it works: If “first_operand” is true, the program will work on “second_operand”. If “first_operand” is false, however, the program will work on “third_operand”.

Chapter 7: The Conditional Statements

C# has conditional statements, mostly used for controlling the flow of execution. The conditional statements expect the programmer to specify a condition or a set of conditions and the corresponding set of statements to be executed if a condition is found to be true. The programmer can also specify the set of statements that are to be executed if the condition is not true.

Let us discuss the various conditional statements supported in C#.

if Statement

This statement is used to evaluate a Boolean expression before a set of statements can be executed. If the condition stands true, then there will be execution of one set of statements, otherwise, another set of statements will be executed. Here is the syntax for this statement^[14]:

```
if(boolean_expression) {  
    /* statement(s) to execute if above boolean expression is true */  
}
```

Here is an example:

```
using System;  
  
namespace DecisionMaking {  
    class IfStatement {  
        static void Main(string[] args) {  
            /* defining a local variable */  
            int x = 5;  
  
            /* check a boolean condition via if statement */  
            if (x < 10) {  
                /* to be printed of the condition is true */  
                Console.WriteLine("x is less than 10");  
            }  
  
            Console.WriteLine("The value of x is : {0}", x);  
        }  
    }
```

```

        Console.ReadLine();
    }
}

```

The code returns the following output:

```

x is less than 10
The value of x is : 5

```

The condition was found to be true, that is, the value of variable x is less than 10, hence, and the statement just below the condition was executed. What if the condition was false?

using System;

namespace DecisionMaking {

class IfStatement {

static void Main(string[] args) {

/* defining a local variable */

int x = 15;

/* check a boolean condition via if statement */

if (x < 10) {

/* to be printed of the condition is true */

Console.WriteLine("x is less than 10");

}

Console.WriteLine("The value of x is : {0}", x);

Console.ReadLine();

}

}

}

The code returns the following:

```

The value of x is : 15

```

The condition was found to be false, hence the statement outside its block was

executed.

if-else Statement

This is simply a combination of an *if* statement with an *else* part. The *if* part is executed if the condition is true, while the *else* part is executed when the condition is false. The two parts cannot all be executed at once. Here is the syntax for the statement:

```
if(boolean_expression)  
{  
// code executes if the condition is true.  
}  
else  
{  
// code executed if the condition is false  
}
```

Consider the example given below:

using System;

```
namespace DecisionMaking {  
    class IfElseStatement {  
        static void Main(string[] args) {  
            /* defining a local variable */  
            int x = 5;  
  
            /* a check of the boolean condition */  
            if (x < 10) {  
                /* if the condition is true, the following will be printed */  
                Console.WriteLine("x is less than 10");  
            } else {
```



```

        /* if the condition is false, the following will be printed */
        Console.WriteLine("x greater than 10");
    }
    Console.WriteLine("The value of x is : {0}", x);
    Console.ReadLine();
}
}
}

```

The code returns the following:

```

x is less than 10
The value of x is : 5

```

The *if* condition evaluated to a true, hence the statement within its block has been executed. What if the condition was false?

using System;

namespace DecisionMaking {

class IfElseStatement {

static void Main(string[] args) {

/* defining a local variable */

int x = 15;

/* a check of the boolean condition */

if (x < 10) {

/* if the condition is true, the following will be printed */

Console.WriteLine("x is less than 10");

} else {

/* if the condition is false, the following will be printed */

Console.WriteLine("x greater than 10");

}

Console.WriteLine("The value of x is : {0}", x);

Console.ReadLine();

```
    }  
    }  
}
```

The code gives the following output:

```
x greater than 10  
The value of x is : 15
```

The *if* condition evaluated to a false, hence the statement within the *else* block has been executed. You must have noticed that part outside the two blocks has been executed in both cases. That is what happens.

else...if Statement

In some cases, we may be in need of checking a multiple number of conditions. In such a case, we can use the *else if* statement. It takes the syntax given below:

```
if(boolean_expression a) {  
    /* Runs if the boolean expression a is true */  
}  
else if( boolean_expression b) {  
    /* Runs if the boolean expression b is true */  
}  
else if( boolean_expression c) {  
    /* Runs if the boolean expression c is true */  
} else {  
    /* to run if none of above conditions is true */  
}
```

Here is an example:

```
using System;
```

```
namespace DecisionMaking {  
    class ElseIfStatement {  
        static void Main(string[] args) {
```

```

    /* defining a local variable */
    int x = 5;

    /* checking the boolean condition */
    if (x == 1) {
        /* print this statement if the condition is true */
        Console.WriteLine("Value of x is 1");
    }
    else if (x == 2) {
        /* print this statement if the if else if condition is true*/
        Console.WriteLine("Value of x is 2");
    }
    else if (x == 3) {
        /* print this statement if the if else if condition is true */
        Console.WriteLine("Value of x is 3");
    } else {
        /* print this statement if none of the above conditions is true */
        Console.WriteLine("All conditions are false");
    }
    Console.WriteLine("Exact value of x is: {0}", x);
    Console.ReadLine();
}
}
}

```

The code prints the following result:

```

All conditions are false
Exact value of x is: 5

```

All the conditions were found to be false; hence the block of code outside the

conditions has been executed. Let us set the value of variable x to 2 and see what happens with the code:

using System;

namespace DecisionMaking {

class ElseIfStatement {

static void Main(string[] args) {

/* defining a local variable */

int x = 2;

/* checking the boolean condition */

if (x == 1) {

/* print this statement if the condition is true */

Console.WriteLine("Value of x is 1");

}

else if (x == 2) {

/* print this statement if the if else if condition is true*/

Console.WriteLine("Value of x is 2");

}

else if (x == 3) {

/* print this statement if the if else if condition is true */

Console.WriteLine("Value of x is 3");

} else {

/* print this statement if none of the above conditions is true */

Console.WriteLine("All conditions are false");

}

Console.WriteLine("Exact value of x is: {0}", x);

Console.ReadLine();

}

}

```
}
```

The output is shown below:

```
Value of x is 2  
Exact value of x is: 2
```

An *else if* condition evaluated to a true hence the statement within its block was executed. The block of code outside all the conditions was also executed. You can play around with the code by modifying the value of x to various values and see what will happen.

Nested if Statements

C# allows us to nest conditional statements. We can nest both the *if* and the *else if* statements, which means that we use them inside another *if* or *else if* statement. The following syntax demonstrates how we nest the *if* statement in C#:

```
if( boolean_expression a) {  
    /* to execute if the boolean expression a is true */  
    if(boolean_expression b) {  
        /* to execute if the boolean expression b is true */  
    }  
}
```

The *else if* statement can be nested using the syntax given above.

Consider the example given below:

using System;

namespace DecisionMaking {

class NestedIf {

static void Main(string[] args) {

/* defining local variables */

int x = 1;

int y = 2;

/* checking the boolean condition */

if (x == 1) {

```

    /* if the condition is true, check the if condition below */
    if (y == 2) {
        /* if the condition is true, print the following */
        Console.WriteLine("Value of x is 1 and b is 2");
    }
}
Console.WriteLine("Exact value of x is : {0}", x);
Console.WriteLine("Exact value of y is : {0}", y);
Console.ReadLine();
}
}
}

```

The code returns:

```

Value of x is 1 and b is 2
Exact value of x is : 1
Exact value of y is : 2

```

Both *if* conditions evaluated to a true, hence the statement within the nested *if* were executed. If either or both conditions evaluated to a false, then this statement could not have been executed. Here is an example:

using System;

namespace DecisionMaking {

class NestedIf {

static void Main(string[] args) {

/* defining local variables */

int x = 1;

int y = 2;

/* checking the boolean condition */

if (x == 1) {

/* if the condition is true, check the if condition below */

```

    if (y == 3) {
        /* if the condition is true, print the following */
        Console.WriteLine("Value of x is 1 and b is 2");
    }
}

Console.WriteLine("Exact value of x is : {0}", x);
Console.WriteLine("Exact value of y is : {0}", y);
Console.ReadLine();
}
}
}

```

In the above code, the nested *if* condition checks whether the value of variable *y* is 3, which is false. The statement within this block will not be executed. The C# compiler will skip this section and proceed to execute the code outside the conditions. It returns the following:

```

Exact value of x is : 1
Exact value of y is : 2

```

switch Statement^[15]

This statement is the same as using multiple *if* statements. It is created with a list of possibilities, an action for every possibility and a default section to be execution in case any of the options doesn't evaluate to a true. Here is syntax for this statement:

```

switch(expression)
{
    case <value_1>
        // code
        break;
    case <value_2>
        // code
        break;
}

```

```
case <value_N>
    // code
    break;
default
    // code
    break;
}
```

Here is an example:

```
using System;
public class SwitchStatement
{
    public static void Main()
    {
        int a = 3;
        switch (a)
        {
            case 1:
                Console.WriteLine("The value of x is 1");
                break;
            case 2:
                Console.WriteLine("The value of x is 2");
                break;
            case 3:
                Console.WriteLine("The value of x is 3");
                break;
            default:
                Console.WriteLine("Unknown value of x");
```



```
        break;
    }
}
}
```

The code returns the following:

```
The value of x is 3
```

The value of variable was initialized to 3. After executing the code, the *case* code for 3 will be matched, hence the statement within its block will be executed. If none of the *case* conditions is true, then the *default* section will be executed. This is demonstrated below:

```
using System;

public class SwitchStatement
{
    public static void Main()
    {
        int a = 5;

        switch (a)
        {
            case 1:
                Console.WriteLine("The value of x is 1");
                break;
            case 2:
                Console.WriteLine("The value of x is 2");
                break;
            case 3:
                Console.WriteLine("The value of x is 3");
                break;
```

default:

```
    Console.WriteLine("Unknown value of x");  
    break;  
}  
}  
}
```

The code returns the output given below:

```
Unknown value of x
```

The case label within the switch statement has to be unique. The switch statement is usable with expressions of any type including strings, integers, bool, char, enum etc. Here is how to use it with a string:

using System;

public class SwitchStatement

```
{  
    public static void Main()  
    {  
        string firstName = "Nicholas";  
        switch (firstName)  
        {  
            case "Samuel":  
                Console.WriteLine("Your first name is Samuel");  
                break;  
            case "Nicholas":  
                Console.WriteLine("Your first name is Nicholas");  
                break;  
            case "Bismack":  
                Console.WriteLine("Your first name is Bismack");  
                break;  
        }  
    }  
}
```

```
}  
}  
}
```

The code returns the following output:

```
Your first name is Nicholas
```

Goto in switch[\[16\]](#)

In some cases when using the *switch* statement, we may need to skip to a certain case. This can be done using either a *jump* or *goto* statement. Consider the example given below:

```
using System;  
  
public class SwitchStatement  
{  
    public static void Main()  
    {  
        string firstName = "Nicholas";  
        switch (firstName)  
        {  
            case "text":  
                Console.WriteLine("is your first name");  
                break;  
            case "Samuel":  
                Console.WriteLine("Samuel");  
                break;  
            case "Nicholas":  
                Console.WriteLine("Nicholas");  
                goto case "text";  
                break;  
            case "Bismack":
```

```
Console.WriteLine("Bismack");
break;
}
}
}
```

The code returns the following output:

Nicholas
is your first name

Nested Switch

A switch statement can be created within another switch statement. This gives us a nested switch. Here is an example:

```
using System;
```

```
public class NestedSwitch
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        int x = 10;
```

```
        switch (x)
```

```
        {
```

```
            case 10:
```

```
                Console.WriteLine(10);
```

```
                switch (x - 1)
```

```
                {
```

```
                    case 9:
```

```
                        Console.WriteLine(9);
```

```
                        switch (x - 2)
```

```
                        {
```

```
                            case 8:
```

```
Console.WriteLine(8);  
break;  
}  
break;  
}  
break;  
case 20:  
Console.WriteLine(20);  
break;  
case 125:  
Console.WriteLine(25);  
break;  
default:  
Console.WriteLine(30);  
break;  
}  
}  
}
```

The output is as follows:

Chapter 8: Loops

As a programmer, you'll use a loop statement to run certain code blocks repeatedly. Loops play an important part in C# programming. Hence, you should study this material carefully as well if you want to master this language in a short while.

Loops – The Basics

While creating programs, you often need to execute code sequences multiple times. Typing the same blocks of code several times can be extremely boring. Thus, you need to find a quick and simple way to repeat codes. Fortunately, the C# language supports loop statements.

A loop is a tool that runs code fragments repeatedly. You can use it to run codes for a certain number of times or as long as a given condition is fulfilled. C# offers different types of loop statements. Let's analyze each type in detail:

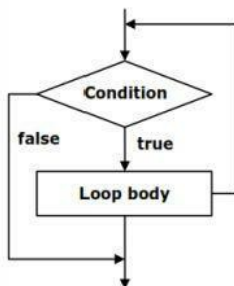
The While Loop^[17]

This is the simplest loop statement in C#. Its syntax is:

```
while (the_assigned_condition)  
{  
    the_loop's_body;  
}
```

In this syntax, “the_assigned_condition” is an expression that produces a Boolean value. This condition dictates how many times the “body” will run. The “body”, on the other hand, is the statement (or group of statements) that you want to execute.

The diagram given below will illustrate how while loops work:



When working on a “while” loop, a C# program checks the value of the Boolean

expression. If the value is “true”, the program will run all of the statements within the loop’s body. Then, the program will check the Boolean expression to see its value. If the value is still true, the program will rerun the loop’s body and go back to the first step (i.e. check the expression’s value). This process will go on until the Boolean expression evaluates to false. Once this happens, the program will process the code blocks right after the loop.

Important Note: Your C# program won’t run the while loop’s body if the Boolean expression is false. Thus, if the Boolean value is false when you launched the program, your in-loop statements will never be executed.

The following example will illustrate how while loops work:

```
int timer = 10;

while (timer >= 0)

{
    System.Console.WriteLine("Time remaining: " + timer);

    timer--;
}
```

If you will compile and execute this code, your command prompt will print this data:

```
Time remaining: 10
Time remaining: 9
Time remaining: 8
Time remaining: 7
Time remaining: 6
Time remaining: 5
Time remaining: 4
Time remaining: 3
Time remaining: 2
Time remaining: 1
Time remaining: 0
```

The Break Operator

You can use “break” (a C# operator) to exit a loop prematurely. Programmers use this operator if they don’t want to wait for the loop’s natural termination. Basically, a loop will end once it encounters a break operator. This situation forces the program to

jump to the code fragments right after the loop.

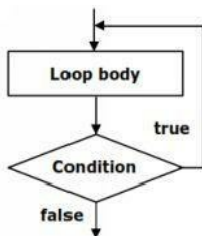
In C#, you can only use this operator inside the loop's body. That means "break" will only work while the loop is running.

The Do-While Loop^[18]

A do-while loop is almost identical to a while loop. The only difference between these loops is that the former analyzes the Boolean value after running the loop's body. Thus, you can rest assured that your codes will run at least once even if the Boolean expression is false. Here is the syntax of the do-while loop:

```
do
{
    statements;
} while (Boolean expression)
```

Do-while loops work according to this pattern:



The program will execute all of the statements within the loop's body. Then, it will check the condition (i.e. the Boolean expression). If the condition evaluates to true, the program will execute the loop's body again and perform the "value inspection". This process will go on until the condition becomes false. Thus, your codes might run forever if your assigned condition always evaluates to true.

The For Loop^[19]

In general, "for" loops are more complex than do-while and while loops. However, for loops can help you perform difficult tasks using fewer C# codes. The syntax of a "for" loop is:

```
for (initializer; Boolean_expression; update)
```



```
{  
    the_loop's_body;  
}
```

A “for” loop has an initializer (i.e. the initial value of the counter), a Boolean expression, a C# statement that updates the counter, and the loop’s body.

The “counter” (i.e. the initial value assigned to the loop) serves as a “for” loop’s most distinctive feature. In most cases, the counter’s value increases until it reaches the final value (e.g. 1 to 10). Also, programmers usually know how many times the “for” loop will iterate.

You may include one or more variables in your “for” loops. These variables may move in descending or ascending order. When writing programs, you may combine ascending and descending variables in a single “for” loop. In addition, an ascending variable can go from 1 to 1024 because “for” loops support arithmetic operations (e.g. addition, multiplication, etc.).

Important Note: All of the parts of a “for” loop are optional. You can create an infinite “for” loop by leaving blank spaces on the syntax. Here’s an example:

```
for ( ; ; )  
{  
    // Loop body  
}
```

At this point, you’re ready to learn more about the different sections of the “for” loop.

The Initialization

A “for” loop can possess an initializing fragment:

for (double sample = 1; ...; ...)

```
{  
  
    // You can use the “sample” variable here.  
  
}
```

// You can’t use the variable here.

C# programs execute this fragment once, right before running the “for” loop. Often, programmers use an initializing fragment to create a counter (also known as “loop variable”) and assign its initial value. This counter is available and usable when inside the loop’s body. C# allows you to declare multiple variables using a single initializing fragment.

The Condition

Obviously, a “for” loop needs a conditional expression. Here’s a sample:

```
for (double sample = 1; sample < 5; ...)
```

```
{  
    // This is the loop’s body.  
}
```

Computer programs evaluate the conditional expression before running the loop’s body. If the result is “true”, the body of the loop will run; otherwise, the involved program will jump to the statements written after the current loop.

The Update Statement

This is the final part of a “for” loop. Basically, an update statement updates the loop’s counter. Let’s use the code snippet given above:

```
for (double sample = 10; sample > 1; sample--)
```

```
{  
    // This is the loop’s body.  
}
```

The program executes this part after running the loop’s body. Keep in mind that this statement updates the counter’s value.

The Loop’s Body

This part consists of C# statements. It can utilize the variables you declared in the loop's initializing fragment. Check the example below:

```
for (double sample = 10; sample > 1; sample--)  
{  
    System.Console.WriteLine(sample);  
}
```

The Continue Operator

In some cases, you need to stop the active iteration without ending the loop itself. You can accomplish this task using the C# operator called “continue”. The example given below will illustrate how this operator works:

```
int n = int.Parse(Console.ReadLine());  
int sum = 0;  
for (int i = 1; i <= n; i += 2)  
{  
    if (i % 7 == 0)  
    {  
        continue;  
    }  
    sum += i;  
}  
Console.WriteLine("sum = " + sum);
```

This code computes the total of all the odd numbers within the range (1 to n), which produce remainders when divided by 7.

The “Foreach” Loop^[20]

Just recently, C# introduced the concept of “foreach” loops (i.e. extended “for” loops). This loop concept is also available in other programming languages such as C, VB, C++, PHP, etc. With this programming tool, you can run all of the elements of a list, array, or other groups of values. A “foreach” loop takes all of the existing elements even in non-indexed data groups.

When writing this kind of loop, use the following syntax:

```
foreach (type name_of_variable in name_of_group)  
{
```

```
        the_statements;  
    }
```

As you can see, this loop is much simpler than a typical “for” loop. A “foreach” loop can help you scan all of the objects inside a collection quickly. It’s no surprise that countless programmers use this loop in writing their codes.

The Nested For Loop

C# allows you to place a “for” loop inside another “for” loop. The loop at the innermost part of the code runs the most number of times. The one at the outermost section, on the other hand, gets the least number of repetitions. You should use the following syntax when “nesting” for loops:

```
for (initializing_fragment, condition, update_statement)  
{  
    for (initializing_fragment, condition, update_statement)  
    {  
        statements  
    }  
}
```

Here, the program runs the initial “for” loop, executes its body, and triggers the nested loop. The program will check the second loop’s condition and execute the codes inside it until the evaluation becomes false. Then, the program will make the necessary adjustments on the loops’ counters. The program will repeat the entire process until all of the conditions evaluate to false.

Chapter 9: C# Methods

This chapter will focus on the C# methods. Here, you'll learn how to declare and utilize methods in your own programs. You need to memorize the lessons contained in this chapter to be an effective C# programmer.

Methods – The Basics

For most programmers, methods are core aspects of any program. Methods can solve problems, accept user inputs (known as parameters), and produce results.

A method is simply a group of statements that have been put together to perform a common task. Every C# class has at least one method, which is the Main method. For you to be able to use a method, you must define it, and then call it to perform the action it was intended to perform. Methods complete their tasks by representing the data conversions done by the program. Additionally, methods are the areas where the actual processes are completed. This is the main reason why C# programmers consider methods as the basic units of any computer program.

The Benefits Offered by Methods

In this section, you'll discover the major benefits offered by C# methods. After reading this material, you'll know why you should use these tools in writing your programs.

Better Structure and Code Readability

Programming experts claim that you should use methods while writing computer programs. Methods, even the simplest ones, can improve the structure and readability of your C# codes.

Here's an important fact that you need to know: programmers spend 20% of their time on writing and checking their computer program. The remaining 80% is spent on maintaining and improving the software. Obviously, you'll have an easy time working on your program if your codes are readable and well-structured.

Prevention of Redundant Codes

Methods can help you avoid redundant codes in your programs. Redundant or duplicated codes often produce undesirable results in computer applications.

Better Code Repetition

If your program needs to use a certain code fragment multiple times, it's an excellent idea to transfer the said fragment into a C# method. You can invoke methods multiple times, which means you can repeat important codes without retyping them.

Declaring, Implementing, and Invoking C# Methods^[21]

Method Definition - A method definition is simply a declaration of the elements that form the structure of the method. Here is the syntax for method definition in C#:

```
<Access_Specifier><The_Return_Type><Your_Method_Name>  
(Parameter_List) {  
    Method Body  
}
```

At this point, you need to know the processes that you can perform on an existing method^[22]. These processes are:

- Declaration – In this process, you will link a method to a program. This process allows you to call the method in any part of your program.
- Implementation – This process involves entering codes to complete a certain task. The codes involved here exist inside the method you're using.
- Invocation – This is the process of calling a declared method. Here, you'll use the method to solve a problem or perform an action.

Method Declarations

In C#, you need to declare methods inside a class. Additionally, you're not allowed to "nest" methods (i.e. write a method inside the body of another method). The perfect example for this is Main(), the method you've used multiple times. The code given below will illustrate this:

```
class DeclaringMethods  
{  
    static void Main()  
{  
    System.Console.WriteLine("Hi C#!");  
}
```

```
}  
  
}
```

The Syntax

Use the following syntax when declaring a method:

static data_type_of_the_result name_of_method (list_of_parameters)

Let's analyze this syntax using the `Main()` method (i.e. *static void Main()*). `Main()` uses "void" as its return type since it doesn't generate any result. The word "Main" serves as its name. The parentheses, on the other hand, act as containers for the parameters that the programmer will provide.

Important Note: You have to follow this syntax when writing a C# program. Don't change the placement of the method's parts.

C# doesn't require you to include parameters in your declarations. That means you can leave the parentheses empty (e.g. `Main()`).

The Method's Name

You need to indicate the method's name during declarations, invocations, or implementations. Here are the rules that you need to remember when naming your methods:

- Make sure that the initial letter is in uppercase.
- Begin each word with an uppercase letter (e.g. `SampleMethod`, `NewMethod`, `MainLine`, etc.).
- Use verbs and nouns as names of your methods.

Important Note: The rules discussed above are completely optional. Follow these rules if you want to have excellent structure and readability in your C# codes.

Method Implementations

Declaring a method isn't enough. You also need to implement your methods if you want them to run inside your programs. Programmers use the term "body" when referring to the implementation of a method.

The Body

You'll find the method's body inside a pair of curly braces. This body consists of commands, expressions, and statements that you want to execute. Thus, the body is an important part of any method.

Important Note: Keep in mind that you cannot nest methods in C#. Don't write methods inside other methods.

Method Invocations

Basically, this is the process of running the statements within the method's body. Invoking a method is easy and simple: you just have to indicate its name, add a pair of curly braces, and terminate the line using a semicolon. Here's the syntax that you should use:

name_of_method();

The sample code given below will show you how to invoke a method:

```
class Animals
{
    static void Main()
    {
        Console.WriteLine("I love dogs.");
    }
}
```

If you'll compile and execute that code, your command prompt will print the following message:

I love dogs.

The Places Where You Can Invoke a Method

C# allows you to invoke methods in the following areas:

- Inside the program's main method (i.e. Main()).
- Inside another method.
- Inside the method's own body. This technique is called "recursion".

Recursive Method Call^[23]

It is possible for a method to call itself. This process is referred to as *recursion*^[24]. Consider the factorial example given below:


```

using System;
namespace MethodApplication {
    class NumberChecker {
        public int factorial(int x) {
            /* declaring a local variable */
            int answer;
            if (x == 1) {
                return 1;
            } else {
                answer = factorial(x - 1) * x;
                return answer;
            }
        }
    }
    static void Main(string[] args) {
        NumberChecker n = new NumberChecker();
        //let's now call the factorial method {0}", n.factorial(8));
        Console.WriteLine("The factorial of 8 is : {0}", n.factorial(8));
        Console.WriteLine("The factorial of 9 is : {0}", n.factorial(9));
        Console.ReadLine();
    }
}

```

We have created the factorial function and instance of the class named *n*. The code will return the result given below:

```

The factorial of 8 is : 40320
The factorial of 9 is : 362880

```

The code was able to give us the factorial of 8 and 9. Note that the factorial of a number is the multiplication of all the numbers below it except 0. For example, the factorial of 8 is $1 * 2 * 3 * 4 * 5 * 6 * 7 * 8$, which gives 40320 as shown in the

above result.

Passing Parameters to Methods^[25]

If a method was defined with parameters, then parameters should be passed to it during the call. Three methods can be used for passing parameters to methods. Let us discuss these methods.

Pass By Value

This method involves copying the actual value of an argument to the formal function parameter. This is the default mechanism of passing parameters to a method. With this mechanism, when calling a function, a new location is created in the memory for every value parameter. The values for the actual parameters are then copied into them. This means that the changes that are made to the parameter inside the method will have no effect on the argument.

Let us demonstrate this by an example:

using System;

namespace MethodApplication {

class NumberChecker {

public void swap(int a, int b) {

int temp;

temp = a; /* saving the value of a */

a = b; /* putting b into a */

b = temp; /* putting temp into b */

}

static void Main(string[] args) {

NumberChecker n = new NumberChecker();

/* defining a local variable */

int x = 10;

int y = 20;

Console.WriteLine("Before swapping, value of x is : {0}", x);

```

    Console.WriteLine("Before swapping, value of y is : {0}", y);

    /* let's swap the values by calling the function */
    n.swap(x, y);

    Console.WriteLine("After swapping, value of x is : {0}", x);
    Console.WriteLine("After swapping, value of y is : {0}", y);

    Console.ReadLine();
}
}
}

```

The code gives the following output:

```

Before swapping, value of x is : 10
Before swapping, value of y is : 20
After swapping, value of x is : 10
After swapping, value of y is : 20

```

What we did is that we changed the values within the function. However, the above output shows that this change did not take effect; hence it has not been reflected above.

That is how we pass parameters by value in C#.

Pass By Reference

A reference parameter references a memory location of a variable. When parameters are passed by reference, no creation of a new memory location, unlike what happens in the pass by value. Reference parameters actually reference the same memory location as the actual parameters being supplied to the method.

To declare reference parameters, we use the *ref* keyword. Let us demonstrate this using an example:

```

using System;

namespace MethodApplication {
    class NumberChecker {

```

```

public void swap(ref int a, ref int b) {
    int temp;

    temp = a; /* saving the value of a */
    a = b;    /* putting b into a */
    b = temp; /* putting temp into b */
}

static void Main(string[] args) {
    NumberChecker n = new NumberChecker();

    /* defining a local variable */
    int x = 10;
    int y = 20;

    Console.WriteLine("Before swapping, value of x is : {0}", x);
    Console.WriteLine("Before swapping, value of y is : {0}", y);

    /* let's swap the values by calling the function */
    n.swap(ref x, ref y);

    Console.WriteLine("After swapping, value of x is : {0}", x);
    Console.WriteLine("After swapping, value of y is : {0}", y);

    Console.ReadLine();
}
}
}

```

The code returns the following:

```
Before swapping, value of x is : 10  
Before swapping, value of y is : 20  
After swapping, value of x is : 20  
After swapping, value of y is : 10
```

In the pass by value, the values were not swapped. However, in this case, the values have been swapped as shown above. The above change is a reflection is reflected in the *Main* function.

Pass By Output^[26]

A return statement can help us to return a single value only from a function. However, by use of *output parameters*, it is possible for on to return two values from a function. Output parameters are the same as reference parameters, with the difference being that they transfer data out of the method instead of into it.

Here is an example demonstrating this:

using System;

namespace MethodApplication {

class NumberChecker {

public void GetValue(out int a) {

int temp = 20;

a = temp;

}

static void Main(string[] args) {

NumberChecker n = new NumberChecker();

/* defining a local variable */

int x = 10;

Console.WriteLine("Before calling the method, value of x : {0}", x);

/* call the function to get value */

n.GetValue(out x);

```

        Console.WriteLine("After calling the method, the of value of x is : {0}", x);
        Console.ReadLine();
    }
}
}

```

It returns the following:

```

Before calling the method, value of x : 10
After calling the method, the of value of x is : 20

```

The variable that is supplied for the output parameter should be assigned a value. Output parameters are very useful when one wants to return values from a method via the parameters without assigning initial value to the parameter. The following example will help you understand this better:

using System;

namespace MethodApplication {

class NumberChecker {

public void getValues(out int a, out int b) {

Console.WriteLine("Enter in your first value: ");

a = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter in your second value: ");

b = Convert.ToInt32(Console.ReadLine());

}

static void Main(string[] args) {

NumberChecker n = new NumberChecker();

/* defining a local variable */

int x , y;

```
/* call a function to obtain the values */  
n.getValues(out x, out y);  
  
Console.WriteLine("After calling the method, the value of x is : {0}", x);  
Console.WriteLine("After calling the method, the value of y is : {0}", y);  
Console.ReadLine();  
}  
}  
}
```

You will be prompted to enter the two values, so do those. You will then be provided with their values before and after calling the method.

Chapter 10: The Array World^[27]

We can define an array as a special data type that stores a fixed number of values in a sequential manner and by use of a special syntax^[28]. All the array elements must belong to the same data type such as a string, integer, double etc. You can think of an array as a collection of variables of the same type stored in contiguous memory locations. You use an index to get access to a given element within an array.

In the memory, the lowest address identifies the first element in the array while the highest address identifies the highest element in the array.

Declaring Arrays^[29]

The declaration of arrays in C# is done using the following syntax:

`datatype[] arrayName;`

The *datatype* helps us specify the data type of the elements that are stored in the array, not forgetting that all the array elements must belong to the same data type. The square brackets `[]` help in stating the rank of the array, where the rank denotes the size or the number of elements to be stored in the array. The *arrayName* denotes the name of the array. The following are valid examples of array declarations:

`int[] intArray; // may be used for storing int values`

`bool[] boolArray; // may be used for storing boolean values`

`string[] stringArray; // may be used for storing string values`

`double[] doubleArray; // may be used for storing double values`

`byte[] byteArray; // may be used for storing byte values`

`Employee[] customDepartmentArray; // may be used for storing instances of Employee class`

Array Initialization

When an array has been declared, it doesn't mean that it has already been initialized in the memory. After initializing an array, it is possible to assign values to it.

Array initialization can be done at the time of its declaration using the *new* keyword. When this keyword is used, an instance of the array is created. This is demonstrated below:

```
public class MyArray  
public static void Main()  
{  
int[] intArray_1 = new int[4];  
  
int[] intArray_2 = new int[4]{10, 20, 30, 40};  
  
int[] intArray_3 = {10, 20, 30, 40};  
}  
}
```

First, we have declared an array to store 4 integers. Note that the specification of the array size has been done within the square brackets. We have also done the same thing in our second statement, but values have also been assigned to the indices within the curly braces {}. In the third statement, we have declared an array and assigned values to it without specifying its size.

Late Initialization

It is possible for us to initialize an array after it has been declared. This means that it is not a must for us to do both the declaration and the initialization of an array at the same time. Here is an example:

```
using System;  
public class MyArray  
{  
public static void Main()  
{  
string[] array1, array2;
```

```
array1 = new string[4]{ "Nicholas",  
    "Michelle",  
    "John",  
    "Claire",  
};
```

```
array2 = new string[]{"Nicholas",  
    "Michelle",  
    "John",  
    "Claire",  
};  
Console.WriteLine(array1.Length);  
Console.WriteLine(array2.Length);  
}  
}
```

The above mechanism is known as *late initialization*. The initialization in this case must be done using the *new* keyword. We cannot initialize the array by simply assigning values to it. The following example shows an invalid initialization of an array:

```
string[] myarray;
```

```
myrray = {"Nicholas", "Michelle", "John", "Claire"};
```

Accessing Array Elements

The elements of an array were assigned during the initialization time. However, it is possible for us to assign values to an array using the individual indexes. This is demonstrated below:

```
using System;
```

```
public class MyProgram
```

```
{  
public static void Main()  
{  
int[] array1 = new int[5];  
  
array1[0] = 1;  
  
array1[1] = 2;  
  
array1[2] = 3;  
  
array1[3] = 4;  
array1[4] = 5;  
Console.WriteLine(array1[0]);  
Console.WriteLine(array1[1]);  
Console.WriteLine(array1[2]);  
Console.WriteLine(array1[3]);  
Console.WriteLine(array1[4]);  
}  
}
```

The code will return the following result:

```
1  
2  
3  
4  
5
```

The retrieval or access to the array values is also done using the indexes. The index of the element is passed within brackets. Consider the example given below:

```
array1[0];  
array1[1];
```

```
array1[2];
```

```
array1[3];
```

```
array1[4];
```

That is how we can access all the elements of the above array named *array1*.

Using *for* Loop

We can use a *for* loop to access the elements of an array. The loop will iterate through the elements of the array while accessing the required ones via their indices. For example:

```
using System;
```

```
namespace ArrayApp {
```

```
    class MyArray {
```

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

```
            int [] n = new int[10]; /* n is an array of 10 integers */
```

```
            int x, y;
```

```
            /* initialize the elements of the array n */
```

```
            for ( x = 0; x < 10; x++ ) {
```

```
                n[ x ] = x + 50;
```

```
            }
```

```
            /* output each array element's value */
```

```
            for (y = 0; y < 10; y++ ) {
```

```
                Console.WriteLine("Element at index[{0}] = {1}", y, n[y]);
```

```
            }
```

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

```
        }
```

```
    }
```

```
}
```

The *for* loop for the variable *x* helped us fill 10 values into the array, with the first element being 50 and the last one being 59. The *for* loop for variable *y* helped us access all the elements of the array right from index 0 to the last index. The output

from the program is given below:

```
Element at index[0] = 50
Element at index[1] = 51
Element at index[2] = 52
Element at index[3] = 53
Element at index[4] = 54
Element at index[5] = 55
Element at index[6] = 56
Element at index[7] = 57
Element at index[8] = 58
Element at index[9] = 59
```

Using *foreach* Loop

You have known how to use a *for* loop to access all the elements of an array. A *foreach* loop can also help you to iterate through the elements of an array. The following example demonstrates how to use a *foreach* loop to iterate through the elements of an array:

using System;

namespace ArrayApp {

class MyArray {

static void Main(string[] args) {

int [] n = new int[10]; /* n is an array of 10 integers */

//int x, y;

/* initialize the elements of the array n */

for (int x = 0; x < 10; x++) {

n[x] = x + 100;

}

/* output the values of all array elements */

foreach (int y in n) {

int x = y-100;

Console.WriteLine("Element[{0}] = {1}", x, y);

}

Console.ReadKey();

}

}

```
}
```

The code gives the following result:

The *for* loop helped us fill 10 values into the array, with the first element being 100 and the last one being 109. The *foreach* loop helped us access all the elements of the array right from index 0 to the last index. Note that the variable *x* has been used for filling the array values while the variable *y* has been used for iterating through the elements of the array. The code returns the output given below:

```
Element[0] = 100  
Element[1] = 101  
Element[2] = 102  
Element[3] = 103  
Element[4] = 104  
Element[5] = 105  
Element[6] = 106  
Element[7] = 107  
Element[8] = 108  
Element[9] = 109
```

multidimensional Arrays

The concept of multidimensional arrays is supported in C#. Such arrays are also known as rectangular arrays. A multidimensional array is a two-dimensional series organized in the form of rows and columns. The following is an example of a multidimensional array:

```
int[,] array1 = new int[3,2]{  
                {1, 5},  
                {2, 5},  
                {5, 7}  
            };  
  
// or  
  
int[,] array1 = { {1, 5}, {2, 5}, {5, 7} };
```

As you can see in the above example, to initialize a multidimensional array, you must define it in terms of the number of rows and the number of columns. The [3,2] means that the array will have 3 rows and 2 columns.

To access the elements of a multidimensional array, we use two indexes. The first index identifies the row while the second one identifies the column. Note that both indexes begin from 0. Let us demonstrate this using an example:

```
using System;
public class MyArray
{
    public static void Main()
    {
        int[,] array1 = new int[3,2]{
            {1, 5},
            {2, 5},
            {5, 7}

        };

        Console.WriteLine(array1[0, 0]);

        Console.WriteLine(array1[0, 1]);

        Console.WriteLine(array1[1, 0]);

        Console.WriteLine(array1[1, 1]);

        Console.WriteLine(array1[2, 0]);

        Console.WriteLine(array1[2, 1]);
    }
}
```

The code gives us the following output:

1
5
2
5
5
7

That is how we can access the elements. The *array1[2,1]* returns the element located at row 2 and column 1 of the array.

Jagged Arrays

A jagged array is simply an array of arrays. These arrays are storing arrays instead of data type values directly. To initialize a jagged array, we use two square brackets []. In the first bracket, we specify the size of the array while in the second bracket, we specify the dimension of the array which is to be stored as values. Here is an example to show the declaration and initialization of a jagged array:

using System;

public class MyArray

{

public static void Main()

{

int[][] jaggedArray = new int[2][];

jaggedArray[0] = new int[4]{1, 2, 3, 4};

jaggedArray[1] = new int[3]{5, 6, 7};

Console.WriteLine(jaggedArray[0][0]);

Console.WriteLine(jaggedArray[0][2]);

Console.WriteLine(jaggedArray[1][1]);

}

}

We have declared and initialized our jagged array in the above code. The code will give you the following result:

```
1
3
6
```

A jagged array can also store a multidimensional array as a value. We can use [,] in the second bracket to indicate a multi-dimension. Here is an example:

using System;

public class MyArray

{

public static void Main()

{

int[,] jaggedArray = new int[3][,];

jaggedArray[0] = new int[3, 2] { { 1, 5 }, { 2, 6 }, { 4, 8 } };

jaggedArray[1] = new int[2, 2] { { 3, 5 }, { 4, 6 } };

jaggedArray[2] = new int[2, 2];

Console.WriteLine(jaggedArray[0][1,1]);

Console.WriteLine(jaggedArray[1][1,0]);

Console.WriteLine(jaggedArray[1][1,1]);

}

}

The code will return the following:

```
6
4
6
```

In case an additional bracket is added, then this will become an array of array of array. This is demonstrated in the following example:

using System;

```

public class MyArray
{
    public static void Main()
    {
        int[][][] jaggedArray = new int[2][][]
        {
            new int[2][]
            {
                new int[3] { 2, 3, 5},
                new int[2] { 1, 5}
            },
            new int[1][]
            {
                new int[3] { 7, 6, 8}
            }
        };

        Console.WriteLine(jaggedArray[0][0][0]);

        Console.WriteLine(jaggedArray[0][1][1]);

        Console.WriteLine(jaggedArray[1][0][2]);
    }
}

```

The code returns the following:

```

2
5
8

```

Note that we have used three square brackets `[][][]`, which means an array of array of

array. The *jaggedArray* will have 2 elements, which are 2 arrays. Each of these arrays will also have a single dimension array.

Chapter 11: Classes^[30]

A class can be seen as a blueprint for an object. Objects in the real world have characteristics like shape, color and functionalities. For example, X6 is an object of car type. A car has characteristics like color, speed, interior, shape etc. This means that any company that creates an object with the above characteristics will be of type car. This means that the Car is a class while each object, that is, a physical car, will be an object of type Car.

In object-oriented programming (not forgetting that C# is an object-oriented programming language) a class has fields, properties, methods, events etc. A class should define the types of data and the functionality that the objects should have.

With a class, you can create your own custom types by grouping variables of other types together as well as methods and events.

In C#, we use the *class* keyword to define a class^[31]. Here is a simple example of this:

```
public class TestClass
{
    public string field1 = string.Empty;

    public TestClass()
    {
    }

    public void TestMethod(int param1, string param2)
    {
        Console.WriteLine("The first parameter is {0}, and second parameter is {1}",
                               param1, param2);
    }

    public int AutoImplementedPropertyTest { get; set; }
    private int propertyVar;
```

```

public int PropertyTest
{
    get { return propertyVar; }
    set { propertyVar = value; }
}
}

```

The *public* keyword before the class is an Access Specifier, specifying how the class will be accessed. By being public, it means that it will be accessible by all other classes within the same project. We have given the class the name *TestClass*.

We have also defined a field in the class named *field1*. Below this, we have created a constructor for the class. Note that the constructor takes the same name as the class itself, hence the constructor's name is *TestClass()*. Inside this class, we have also defined a method named *TestMethod()*, and this method takes in two parameters, *param1* and *param2*, with the former being an integer and the latter being a string.

Here is another example demonstrating how to declare and use a class:

```

using System;
namespace CubeApplication {
    class Cube {
        public double length; // Length of the cube
        public double breadth; // Breadth of the cube
        public double height; // Height of the cube
    }

    class Cubetester {
        static void Main(string[] args) {
            Cube Cube1 = new Cube(); // Declare Cube1 of type Cube
            Cube Cube2 = new Cube(); // Declare Cube2 of type Cube
            double volume = 0.0; // Store the cube volume here
            // cube 1 specification
            Cube1.height = 4.0;
        }
    }
}

```

```

    Cube1.length = 5.0;
    Cube1.breadth = 8.0;
    // cube 2 specification
    Cube2.height = 8.0;
    Cube2.length = 12.0;
    Cube2.breadth = 14.0;
    // volume of cube 1
    volume = Cube1.height * Cube1.length * Cube1.breadth;
    Console.WriteLine("Volume of Cube1 : {0}", volume);
    // volume of cube 2
    volume = Cube2.height * Cube2.length * Cube2.breadth;
    Console.WriteLine("Volume of Cube2 : {0}", volume);
    Console.ReadKey();
}
}
}

```

The code will return the following result:

```

Volume of Cube1 : 160
Volume of Cube2 : 1344

```

Encapsulation and Member Functions

A member function for a class is simply a function with a definition or prototype within the definition of the class in the same way as any other function. Such a function can operate on any object of the class in which it is a member, and it can access all the class members for the object.

Member functions are simply the attributes of the object (from a design perspective) and they are defined as *private* so as to implement the concept of encapsulation. We can only access such variables using public member functions. Let's demonstrate how we can set and access the various members of a class in C#:

using System;

namespace CubeApplication {

```

class Cube {
    private double length; // Length of a cube
    private double breadth; // Breadth of a cube
    private double height; // Height of a cube
    public void setLength( double len ) {
        length = len;
    }
    public void setBreadth( double brea ) {
        breadth = brea;
    }
    public void setHeight( double heig ) {
        height = heig;
    }
    public double getVolume() {
        return length * breadth * height;
    }
}

class Cubetester {
    static void Main(string[] args) {
        Cube Cube1 = new Cube(); // Declare Cube1 of type Cube
        Cube Cube2 = new Cube();
        double volume;

        // Declare Cube2 of type Cube
        // cube 1 specification
        Cube1.setLength(4.0);
        Cube1.setBreadth(6.0);
    }
}

```

```

    Cube1.setHeight(8.0);

    // cube 2 specification
    Cube2.setLength(10.0);
    Cube2.setBreadth(14.0);
    Cube2.setHeight(12.0);
    // volume of cube 1
    volume = Cube1.getVolume();
    Console.WriteLine("Volume of Cube1 is: {0}" ,volume);

    // volume of cube 2
    volume = Cube2.getVolume();
    Console.WriteLine("Volume of Cube2 is: {0}" , volume);
    Console.ReadKey();
}
}
}

```

Here is the output from the code:

```

Volume of Cube1 is: 192
Volume of Cube2 is: 1680

```

We used the *setter* methods to set the values of the various attributes of our two cubes. The *getVolume()* function has been called to calculate the volumes of the two cubes.

Constructors

A constructor is simply a special member function of a class that is run anytime that we create new objects of the class. A constructor assumes the class name and it should not have a return type. The following example demonstrates how to use a constructor in C#:

```

using System;

```



```

namespace ConstructorApplication {
    class Person {
        private double height; // height of the person

        public Person() {
            Console.WriteLine("We are creating an object");
        }

        public void setHeight( double heig ) {
            height = heig;
        }

        public double getHeight() {
            return height;
        }

        static void Main(string[] args) {
            Person p = new Person();
            // set the person's height
            p.setHeight(7.0);
            Console.WriteLine("The height of the person is: {0}", p.getHeight());
            Console.ReadKey();
        }
    }
}

```

The code should return the following:

```

We are creating an object
The height of the person is: 7

```

A default constructor has no parameters, but it is possible to add parameters to a constructor. Such a constructor is known as a *parameterized constructor*. With such a technique, it is possible for one to assign an initial value to an object during the time of its creation. Here is an example:

```

using System;
namespace ConstructorApplication {
    class Person {
        private double height; // Height of the person
        public Person(double heig) { // A parameterized constructor
            Console.WriteLine("We are creating an object, height = {0}", heig);
            height = heig;
        }
        public void setHeight( double heig ) {
            height = heig;
        }
        public double getHeight() {
            return height;
        }
        static void Main(string[] args) {
            Person p = new Person(8.0);
            Console.WriteLine("The height of the person is : {0}", p.getHeight());
            // set the height
            p.setHeight(7.0);
            Console.WriteLine("The height of the person is : {0}", p.getHeight());
            Console.ReadKey();
        }
    }
}

```

Here is the output from the code:

```

We are creating an object, height = 8
The height of the person is : 8
The height of the person is : 7

```

Destructors

A destructor refers to a special member function of a class that is run anytime an object of the class goes out of scope. A destructor takes the same name as a class but it should be preceded by a tilde (~). A destructor cannot take parameters neither can it return a value.

A destructor is a useful tool for releasing the memory resources before leaving a program. You can overload or inherit a destructor. The following example demonstrates how to use a destructor:

using System;

namespace ConstructorApplication {

class Person {

private double height; // Height of a person

public Person() { // A constructor

Console.WriteLine("We are creating an object");

}

~Person() { //A destructor

Console.WriteLine("We are deleting an object");

}

public void setHeight(double heig) {

height = heig;

}

public double getHeight() {

return height;

}

static void Main(string[] args) {

Person p = new Person();

// set the height of the person

p.setHeight(7.0);

Console.WriteLine("The height of the person is : {0}",

```
p.getHeight());
```

```
    }
```

```
    }
```

```
}
```

Here is the output from the function:

```
We are creating an object  
The height of the person is : 7  
We are deleting an object
```

Static Members

To define a class member as static, we use the *static* keyword. When a class member is declared as static, it means that regardless of the number of objects of the class that are created, there exists only one copy of the static member.

The use of the *static* keyword means that there is only one instance of a member existing in the class. We use this keyword when we need to declare constants since their values can be retrieved by invocation of the class without the creation of an instance of the same. We can initialize static variables outside a class definition or a member function. Static variables can also be initialized inside a class definition.

Let us demonstrate the use of static variables using an example:

using System;

```
namespace StaticApp {
```

```
    class StaticVariables {
```

```
        public static int x;
```

```
        public void count() {
```

```
            x++;
```

```
        }
```

```
        public int getX() {
```

```
            return x;
```

```
        }
```

```
    }
```

```
    class StaticTester {
```

```

static void Main(string[] args) {
    StaticVariables var1 = new StaticVariables();
    StaticVariables var2 = new StaticVariables();
    var1.count();
    var1.count();
    var1.count();
    var2.count();
    var2.count();
    var2.count();
    Console.WriteLine("Variable x for var1 is: {0}", var1.getX());
    Console.WriteLine("Variable x for vars is: {0}", var2.getX());
    Console.ReadKey();
}
}
}

```

The code returns the following result:

```

Variable x for var1 is: 6
Variable x for vars is: 6

```

A member function can also be declared as static. Such a function will only be able to access static variables. Static functions exist even before the creation of the object. Static functions can be used as demonstrated in the following example:

```

using System;

namespace StaticAppli {
    class StaticVariable {
        public static int x;
        public void count() {
            x++;
        }
        public static int getX() {

```

```

        return x;
    }
}

class StaticTester {
    static void Main(string[] args) {
        StaticVariable var = new StaticVariable();

        var.count();
        var.count();
        var.count();
        Console.WriteLine("Variable x is: {0}", StaticVariable.getX());
        Console.ReadKey();
    }
}

```

The code will return the following:

```
Variable x is: 3
```

Chapter 12: Structure

A structure in C# is a value type data type. With a structure, you can use a single variable to hold related data belonging to different data types. In C#, we use the *struct* keyword to create a structure^[32].

Structures are used for tracking records. A good example is when you need to keep a record of all class students or all company employees. For the case of storing student records, some of the details that you may need to track including the names, age, course, date of enrollment, date of completion, amount of fee paid, fee balance etc, guardian address and mobile phone number etc. All these details belong to different data types, meaning that you will be required to create a variable for each of them. However, by use of a structure, you can define a single variable and keep these details together as one.

Here is an example of declaring a structure^[33]:

```
struct Employee{  
    public string name;  
    public string department;  
    public string title;  
    public int age;  
    public double salary;  
};
```

To initialize a struct, we can choose the *new* keyword or not. The members of a struct can be assigned values as shown below:

```
using System;  
  
public class MyStruct  
{  
    public static void Main()  
{
```

```
Employee emp = new Employee()
emp.name = "Nicholas Samuel";
emp.department = "Computing";
emp.age = 26;
emp.salary = 5000;
Console.WriteLine(emp.name);
Console.WriteLine(emp.department);
Console.WriteLine(emp.age);
Console.WriteLine(emp.salary);
}
}
```

```
public struct Employee
{
    public string name { get; set; }
    public string department { get; set; }
    public string title { get; set; }
    public int age { get; set; }
    public double salary { get; set; }
}
```

The code gives the following result:

```
Nicholas Samuel
Computing
26
5000
```

The struct named *Employee* was defined using the *struct* keyword. The various details of this struct have been declared within this. At the top of the class, we created an instance of this struct and we gave it the name *emp*. This instance has been used to access the various attributes of the struct for display on the screen.

Note that a struct is a value type, and this makes it faster when compared to a class object. This has made them good for use in game programming. However, one can

easily transfer a class object than a struct. This means that a struct should not be used when one is need of transferring data to other classes.

Here is another example of a struct in C#:

using System;

struct Cars {

public string model;

public int cc;

public int passengers;

public int year;

};

public class carStructure {

public static void Main(string[] args) {

Cars Premio; /* Declare prenio of type Car */

Cars X6; /* Declare X6 of type Car */

/* premio specification */

Premio.model = "Saloon";

Premio.cc = 1800;

Premio.passengers = 5;

Premio.year = 2010;

/* X6 specification */

X6.model = "Saloon";

X6.cc = 3500;

X6.passengers = 5;

X6.year = 2012;

/* print premio info */

Console.WriteLine("Premio model: {0}", Premio.model);

Console.WriteLine("Premio CC : {0}", Premio.cc);

Console.WriteLine("Premio passengers : {0}", Premio.passengers);

```

    Console.WriteLine("Premio year :{0}", Premio.year);

    /* print X6 info */

    Console.WriteLine( "X6 model: {0}", X6.model);

    Console.WriteLine("X6 CC : {0}", X6.cc);

    Console.WriteLine("X6 passengers : {0}", X6.passengers);

    Console.WriteLine("X6 year :{0}", X6.year);

    Console.ReadKey();

}

}

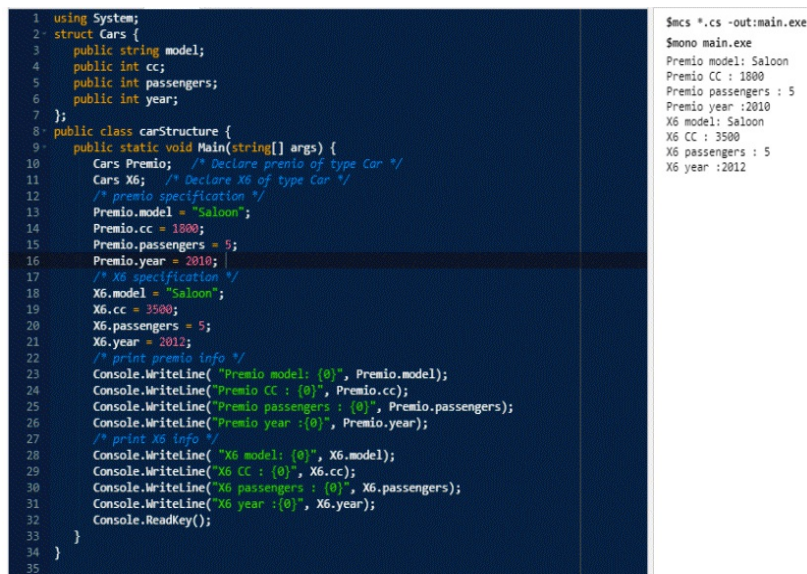
```

The code will return the following details^[34]:

```

Premio model: Saloon
Premio CC : 1800
Premio passengers : 5
Premio year :2010
X6 model: Saloon
X6 CC : 3500
X6 passengers : 5
X6 year :2012

```



The screenshot shows a code editor with C# code on the left and its output on the right. The code defines a struct named `Cars` with two instances, `Premio` and `X6`. It then prints the details of these instances. The output shows the details for both cars, with `Premio` having a year of 2010 and `X6` having a year of 2012.

```

1 using System;
2 struct Cars {
3     public string model;
4     public int cc;
5     public int passengers;
6     public int year;
7 };
8 public class carStructure {
9     public static void Main(string[] args) {
10         Cars Premio; /* Declare premio of type Car */
11         Cars X6; /* Declare X6 of type Car */
12         /* premio specification */
13         Premio.model = "Saloon";
14         Premio.cc = 1800;
15         Premio.passengers = 5;
16         Premio.year = 2010;
17         /* X6 specification */
18         X6.model = "Saloon";
19         X6.cc = 3500;
20         X6.passengers = 5;
21         X6.year = 2012;
22         /* print premio info */
23         Console.WriteLine("Premio model: {0}", Premio.model);
24         Console.WriteLine("Premio CC : {0}", Premio.cc);
25         Console.WriteLine("Premio passengers : {0}", Premio.passengers);
26         Console.WriteLine("Premio year :{0}", Premio.year);
27         /* print X6 info */
28         Console.WriteLine("X6 model: {0}", X6.model);
29         Console.WriteLine("X6 CC : {0}", X6.cc);
30         Console.WriteLine("X6 passengers : {0}", X6.passengers);
31         Console.WriteLine("X6 year :{0}", X6.year);
32         Console.ReadKey();
33     }
34 }
35

```

```

$mscs *.cs -out:main.exe
$mono main.exe
Premio model: Saloon
Premio CC : 1800
Premio passengers : 5
Premio year :2010
X6 model: Saloon
X6 CC : 3500
X6 passengers : 5
X6 year :2012

```

We have created a single struct named *Cars*. In this structure, we have stored the two members, *Premio* and *X6*. These two instances are instances of the structure. The properties for these had been defined inside the structure. The two instances share properties, but these will take different values. We have then printed the values of these properties as shown in the above output.

Characteristics of Structures

So far, you have known how to use structures and some of their characteristics. The structures in C# have a great difference from the structures supported in C and C++. C# structures have the features described below:

- Structures may have fields, indexers, methods, properties, operators and events.
- A structure can have defined constructors, not destructors. However, in a structure, we are not allowed to define a default constructor. The reason is that this is defined automatically and one cannot change it.
- A structure cannot inherit another structure or a class.
- With a structure, we can implement one or more interfaces.
- The members of a structure cannot be specified as virtual abstract or protected.
- After the creation of a Struct object via the *New* keyword, the struct object is created and the necessary constructor is called. Unlike classes, we can instantiate a struct without the use of the *New* keyword.
- If we don't use the *New* operator, the fields will not be assigned and it will not be possible to use the object until an assignment has been done to all the fields.

Struct vs. Class

Here the differences between Structs and Classes:

- A Struct is explained as a value type. On the other side, a class is defined as a reference type.
- Structs don't support inheritance. Classes do.
- A Struct cannot have a default constructor. A class can have.

Let us create an example in relation to the above differences:

using System;

struct Cars {

private string model;

private int cc;

private int passengers;

```

    private int year;
    public void getValues(string m, int c, int p, int yr) {
        model = m;
        cc = c;
        passengers = p;
        year = yr;
    }
    public void show() {
        Console.WriteLine("Car model : {0}", model);
        Console.WriteLine("CC : {0}", cc);
        Console.WriteLine("Passengers : {0}", passengers);
        Console.WriteLine("Year :{0}", year);
    }
};

public class myStructure {
    public static void Main(string[] args) {
        Cars Premio = new Cars(); /* Declare Premio of type Cars */
        Cars X6 = new Cars(); /* Declare X6 of type Cars */
        /* Premio specification */
        Premio.getValues("Saloon",
            1800, 5, 2012);
        /* X6 specification */
        X6.getValues("Saloon",
            3500, 5, 2012);
        /* print Premio info */
        Premio.show();
        /* print X6 info */
    }
}

```

```
X6.show();  
Console.ReadKey();  
}  
}
```

The code will give the following output after execution:

```
Car model : Saloon  
CC : 1800  
Passengers : 5  
Year :2012  
Car model : Saloon  
CC : 3500  
Passengers : 5  
Year :2012
```

Chapter 13: Encapsulation

Encapsulation is the process of enclosing items within a logical or physical package. In object-oriented programming, encapsulation is used to prevent access to the implementation details.

Encapsulation and Abstraction are closely related features in object-oriented programming. The purpose of abstraction is to make the relevant details visible to users while the purpose of encapsulation is to enable a programmer to implement the required level of abstraction.

To implement encapsulation^[35], we use *access specifiers*. The role of an access specifier is to state the visibility and scope of a class member. There are different types of access specifiers that are supported in C#. They include the following:

- Public
- Protected
- Private
- Protected internal
- Internal

Private Access Specifier

With a private access specifier, a class is able to hide its member functions and member variables from other objects and functions. Only functions of a similar class are able to access its private members. An instance of the class is not able to access the private members of the class. Consider the example given below:

using System;

namespace AccessApplication {

class Figure {

// The class member variables

private double width;

private double length;

public void GetDetails() {

```

        Console.WriteLine("Enter the Length: ");
        length = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Enter the Width: ");
        width = Convert.ToDouble(Console.ReadLine());
    }

    public double CalculateArea() {
        return length * width;
    }

    public void Show() {
        Console.WriteLine("The Length of the figure is: {0}", length);
        Console.WriteLine("The Width of the figure is: {0}", width);
        Console.WriteLine("The Area of the figure is: {0}", CalculateArea());
    }
}

//end the class Figure
class RunFigure {
    static void Main(string[] args) {
        Figure f = new Figure();
        f.GetDetails();
        f.Show();
        Console.ReadLine();
    }
}
}

```

Run the code and enter the measurements of the figure, both the width and the length. You will get the area of the figure.

The member variables, that is, the *width* and the *length* have been defined as *private*. This means that we cannot access them from the `Main()` function. The `GetDetails()` and `Show()` member functions have been declared as *public*, hence they are able to access these variables. We can also access them from the `Main()` function after

creating an instance of the *Figure* class. This instance has been given the name *f*.

Public Access Specifier

With a public access specifier, a class is able to expose its member functions and member variables to the other objects and functions. Any class member declared as public can be accessed from outside of that class. Consider the following example:

using System;

namespace FigureApplication {

class Figure {

// The class member variables

public double width;

public double length;

public double CalculateArea() {

return length * width;

}

public void Show() {

Console.WriteLine("The Length of the figure is: {0}", length);

Console.WriteLine("The Width of the figure is: {0}", width);

Console.WriteLine("The Area of the figure is: {0}", CalculateArea());

}

}//end the Figure class

class RunFigure {

static void Main(string[] args) {

Figure f = new Figure();

f.length = 5.4;

f.width = 3.2;

f.Show();

Console.ReadLine();

}


```
}  
}
```

The following is the output from the code:

```
The Length of the figure is: 5.4  
The Width of the figure is: 3.2  
The Area of the figure is: 17.28
```

The class member variables *width* and *length* have been declared as *public*, which means that we are able to access them from the *Main()* function after creating an instance of the class. We have created an instance of the *Figure* class and given it the name *f*.

The *CalculateArea()* and *Show()* member functions are also able to access these member variables without using an instance of the class. The *Show()* member function has also been declared as *public*, meaning that we are able to access it from the *Main()* function using an instance of the class.

Protected Access Specifier

This type of access specifier makes it possible for a child class to access the member functions and member variables that have been defined in the base class. This way, it is easier to implement inheritance. This will be discussed in inheritance.

Internal Access Specifier

This type of access specifier allows a class to expose its member functions and member variables to the other objects and functions in the current assembly. This means that any member that has been created using the internal access specifier is accessible from any class or method that has been defined within the application where the member has been defined. This is demonstrated in the following example:

using System;

namespace AccesserApplication {

class Figure {

// The member variables

internal double width;

internal double length;

double CalculateArea() {

```
        return length * width;
    }
    public void Show() {
        Console.WriteLine("The Length of the figure is: {0}", length);
        Console.WriteLine("The Width of the figure is: {0}", width);
        Console.WriteLine("The Area of the figure is: {0}", CalculateArea());
    }
} //end the class Figure
```

```
class RunFigure {
    static void Main(string[] args) {
        Figure f = new Figure();
        f.length = 6.5;
        f.width = 4.8;
        f.Show();
        Console.ReadLine();
    }
}
}
```

The code should return the following result:

```
The Length of the figure is: 6.5
The Width of the figure is: 4.8
The Area of the figure is: 31.2
```

Chapter 14: Inheritance

With inheritance, it is possible for a programmer to define one class in terms of another class. This also makes it possible for us to reuse code and shorten the time taken to implement an application.

During the creation of a class, instead of having to create completely new member functions and data members, the programmer is able to designate that the new class inherits the members of an already existing class. The *base class* is the name for the existing class while the new class is known as the *derived class*.

Inheritance is simply an implementation of IS-A relationship^[36]. For example, Cow is a Mammal.

It is possible for a class to inherit from more than one class or interfaces, meaning that it can inherit data and functions from many base interfaces or classes. The following example demonstrates the concept of the base and derived class:

using System;

namespace InheritanceApp {

class Figure {

public void setWidth(int wid) {

width = wid;

}

public void setHeight(int heig) {

height = heig;

}

protected int width;

protected int height;

}

// A Derived class

class Rectangle: Figure {

public int calculateArea() {

```

        return (width * height);
    }
}

class TestInheritance {
    static void Main(string[] args) {
        Rectangle R = new Rectangle();
        R.setWidth(6);
        R.setHeight(8);
        // Print the rectangle's area.
        Console.WriteLine("The area is: {0}", R.calculateArea());
        Console.ReadKey();
    }
}

```

The code should return the following output:

```
The area is: 48
```

We have defined the *Figure* class with two properties namely *width* and *height*. This class also has two methods, *setWidth()* and *SetHeight()*. We have then defined a class named *Rectangle*. Notice the syntax we have used to create this class:

class Rectangle: Figure {

The use of the full colon signals that the *Rectangle* class is inheriting the *Figure* class. The *Rectangle* class only has one method, the *calculateArea()* method. However, since it has inherited the *Figure* class, it means that it has all the properties of the *Figure* class, like the width and the height.

We have then created the *TestInheritance* class. Within the method *Main()*, an instance of the *Rectangle* class has been created and given it the name *R*. We have used this instance to access the properties that have been defined in both the *Rectangle* and the *Figure* classes. That is how powerful inheritance is!

Base Class Initialization

The derived class inherits the member methods and member variables defined in the

base class. This means that we should create the super class object before creating the subclass. Instructions for initialization of the superclass can be given in the initialization of the list of members.

This is demonstrated in the program given below:

using System;

namespace InheritanceApp {

class Object {

// The member variables

protected double width;

protected double length;

public Object(double len, double wid) {

width = wid;

length = len;

}

public double CalculateArea() {

return length * width;

}

public void Show() {

Console.WriteLine("The Width of the figure is: {0}", width);

Console.WriteLine("The Length of the figure is: {0}", length);

Console.WriteLine("The Area of the figure is: {0}", CalculateArea());

}

}//end class Figure

class Carpet : Object {

private double cost;

public Carpet(double len, double wid) : base(len, wid) { }

public double CalculateCost() {

double cost;

```

        cost = CalculateArea() * 120;
        return cost;
    }

    public void Show() {
        base.Show();

        Console.WriteLine("The toatl cost for the caprpet is: {0}",
CalculateCost());
    }
}

class RunObject {
    static void Main(string[] args) {
        Carpet c = new Carpet(7.5, 9.5);
        c.Show();
        Console.ReadLine();
    }
}

```

The code will return the following output:

```

The Width of the figure is: 9.5
The Length of the figure is: 7.5
The Area of the figure is: 71.25
The toatl cost for the caprpet is: 8550

```

Multiple Inheritance

Multiple inheritance is not supported in C#. However, with interfaces, it is possible for us to implement multiple inheritance. Consider the following example:

using System;

namespace InheritanceApp {

class Object {

public void setWidth(int wid) {

width = wid;

```
}  
  
public void setHeight(int heig) {  
    height = heig;  
}  
  
protected int width;  
protected int height;  
}
```

// PaintCost for the Base class

```
public interface PaintCost {  
    int calculateCost(int area);  
}
```

// The derived class

```
class Board : Object, PaintCost {  
    public int calculateArea() {  
        return (width * height);  
    }  
  
    public int calculateCost(int area) {  
        return area * 120;  
    }  
}
```

```
class InheritanceTester {  
    static void Main(string[] args) {  
        Board B = new Board();  
        int area;  
        B.setWidth(5);  
        B.setHeight(7);  
    }  
}
```

```
    area = B.calculateArea();  
    // Show the area of the object  
    Console.WriteLine("The area of the object is: {0}", B.calculateArea());  
        Console.WriteLine("The toal printing cost is: ${0}" ,  
B.calculateCost(area));  
    Console.ReadKey();  
}  
}  
}
```

The code gives the following output:

```
The area of the object is: 35  
The toal printing cost is: $4200
```


Chapter 15: Polymorphism

Polymorphism^[37] is a term that means taking many forms. In programming, it is expressed as “**one interface, many functions**”. Polymorphism can take two forms, static or dynamic.

For the case of static dynamism, response to the function is determined during compile time. In dynamic polymorphism, response to the function is determined during runtime.

Static Polymorphism

As stated above, the response to a function in this type of polymorphism is determined during compile time. The process of linking a function to an object during compile time is known as *early binding*. It is also known as *static binding*. In C#, static polymorphism can be implemented in two ways:

- Function overloading
- Operator overloading

Function Overloading

It is possible for us to have multiple definitions for the same function name within one scope. The differences between these functions are implemented by types or the number of arguments that the functions take. Function declarations cannot be overloaded by differing the return types only.

The following example demonstrates how to overload a function in C#:

using System;

```
namespace PolymorphismApp {  
    class DisplayData {  
        void display(int x) {  
            Console.WriteLine("Printing an int value: {0}", x );  
        }  
        void display(double y) {
```

```

        Console.WriteLine("Printing a float value: {0}" , y);
    }
    void display(string z) {
        Console.WriteLine("Printing a string value: {0}", z);
    }
    static void Main(string[] args) {
        DisplayData dd = new DisplayData();

        // Call display function to return an integer value
        dd.display(10);
        // Call display function to return a float value
        dd.display(11.385);
        // Call display function to return a string value
        dd.display("Hello Sir/Madam");
        Console.ReadKey();
    }
}

```

The code will return the following once executed:

```

Printing an int value: 10
Printing a float value: 11.385
Printing a string value: Hello Sir/Madam

```

We have three definitions of the function *display()*. In this case, overloading has been implementing by varying the types of parameters taken by the function. In one instance, the function is taking an integer value, a float in another instance and a string in the last instance.

Operator Overloading

Operator overloading refers to the use of a single operator to perform various operations. With operator overloading, additional functionalities can be added to the C# operators during their application on user-defined types. This is the case when either one or both the operands belong to a user-defined class.

Here is an example that demonstrates this:

using System;

namespace PolymorphismApp {

class Object {

private double breadth; // The breadth of the box

private double length; // The Length of the box

private double height; // The Height of the box

public double calculateVolume() {

return breadth * length * height;

}

public void setLength(double l) {

length = l;

}

public void setBreadth(double b) {

breadth = b;

}

public void setHeight(double h) {

height = h;

}

// Overload the + operator to add 2 objects.

public static Object operator+ (Object obj1, Object obj2) {

Object ob = new Object();

ob.length = obj1.length + obj2.length;

ob.breadth = obj1.breadth + obj2.breadth;

ob.height = obj1.height + obj2.height;

return ob;

}

}

class BoxTester {

static void Main(string[] args) {

Object Object1 = new Object(); // Declare Object1 of type Object

Object Object2 = new Object(); // Declare Object2 of type Object

Object Object3 = new Object(); // Declare Object3 of type Object

double volume = 0.0; // Store volume of the object here

// object 1 dimensions

Object1.setLength(7.0);

Object1.setBreadth(8.0);

Object1.setHeight(10.0);

// Object 2 dimensions

Object2.setLength(6.0);

Object2.setBreadth(9.0);

Object2.setHeight(11.0);

// Calculate the volume of object 1

volume = Object1.calculateVolume();

Console.WriteLine("The volume of Object1 : {0}", volume);

// Calculate the volume of object 2

volume = Object2.calculateVolume();

Console.WriteLine("The volume of Object2 : {0}", volume);

// Add the two objects

Object3 = Object1 + Object2;

// Calculate the volume of object 3

volume = Object3.calculateVolume();

Console.WriteLine("The volume of Object3 : {0}", volume);

```

        Console.ReadKey();
    }
}

```

The code returns the following output:

```

The volume of Object1 : 560
The volume of Object2 : 594
The volume of Object3 : 4641

```

We have overloaded the + operator. The operator is being utilized for the addition of two numeric types. However, in the above example, we have used the operator to add the two objects together, that is, Object1 and Object2. This has given us Object3. The Object1 and Object2 are user-defined types, meaning that we have used the + operator to add user-defined types.

Note that it is true that we can overload operators in C#, but not all operators can be overloaded. For example, you cannot overload the comparison operators like ==, <, >, !=, >= and <=. For the case of the conditional logic operators like && and ||, we can overload them, but not directly.

Dynamic Polymorphism

With C#, we can create abstract classes that are good for a partial class implementation of an interface. The completion of this is implemented once after a derived class has inherited from it. An abstract class has abstract methods, and the derived class implements these. However, the derived class has a more specialized functionality.

Note that you cannot create an instance of any class that is abstract. An abstract method cannot also be declared outside an abstract class. If a C# class is declared as *sealed*, it means that the class cannot be inherited. However, you are not allowed to declare an abstract class *sealed*. Here is an example of an abstract class:

```
using System;
```

```
namespace PolymorphismApp {
```

```
    abstract class Object {
        public abstract int area();
    }

```

```
    class Object1: Object {

```

```

    private int width;
    private int length;
    public Object1( int x = 0, int y = 0) {
        length = x;
        width = y;
    }
    public override int area () {
        Console.WriteLine("The Object1 area is:");
        return (width * length);
    }
}
class Object1Tester {
    static void Main(string[] args) {
        Object1 obj = new Object1(8, 6);
        double x = obj.area();
        Console.WriteLine("The area is: {0}",x);
        Console.ReadKey();
    }
}

```

The code will print the following result:

```

The Object1 area is:
The area is: 48

```

After defining a function in a class that you want to implement in an inherited class, you should use a *virtual* function.

To implement dynamic polymorphism, we use *abstract classes* and *virtual functions*. Let us demonstrate this using an example:

```

using System;
namespace PolymorphismApp {

```

```
class Object {
    protected int width, height;
    public Object( int x = 0, int y = 0) {
        width = x;
        height = y;
    }
    public virtual int area() {
        Console.WriteLine("The area of the parent class is :");
        return 0;
    }
}

class Object1: Object {
    public Object1( int x = 0, int y = 0): base(x, y) {

    }
    public override int area () {
        Console.WriteLine("The area of Object1 class is:");
        return (width * height);
    }
}

class Object2: Object {
    public Object2(int x = 0, int y = 0): base(x, y) {
    }
    public override int area() {
        Console.WriteLine("The area of Object2 class is:");
        return (width * height / 2);
    }
}
```

```

    }
    class Caller {
        public void CallArea(Object obj) {
            int x;
            x = obj.area();
            Console.WriteLine("The Area is: {0}", x);
        }
    }
    class TesterClass {
        static void Main(string[] args) {
            Caller c = new Caller();
            Object1 ob1 = new Object1(10, 7);
            Object2 ob2 = new Object2(10, 5);
            c.CallArea(ob1);
            c.CallArea(ob2);
            Console.ReadKey();
        }
    }
}

```

The code returns the following output:

```

The area of Object1 class is:
The Area is: 70
The area of Object2 class is:
The Area is: 25

```


Chapter 16: Regular Expressions^[38]

A regular expression is explained as a pattern that we can match against an input text. In the .Net framework, there is a regular expression engine that we can use to perform that matching. A pattern is made up of one or even more operators, character literals or constructs.

Regex Class

This is a class used for representation of regular expressions. This class comes with a number of inbuilt functions such as the following:

1. `public bool IsMatch(string input)`- the method specifies whether the specified regular expression in the `Regex` constructor gets a match in the specified input string.
2. `public bool IsMatch(string input, int startPoint)`- the method specifies whether the specified regular expression in the `Regex` constructor gets a match in the specified input string from the specified location in the input string.
3. `public static bool IsMatch(string input, int pattern)`- the method specifies whether the specified regular expression gets a match in the specified input string.
4. `public MatchCollection Matches(string input, string replacement)`- this replaces a match in the input string with the specified replacement string.
5. `public string[] Split(string input)`- this function helps in splitting an input string into an array of substrings at positions that have been defined by the regular expression pattern that is specified in a `Regex` constructor.

Consider the example given below:

```
using System.Text.RegularExpressions;
```

```
using System;
```

```
namespace RegularExpApp {
```

```
    class MyProgram {
```

```

private static void findMatch(string input, string expre) {
    Console.WriteLine("The Expression is: " + expre);
    MatchCollection mc = Regex.Matches(input, expre);
    foreach (Match x in mc) {
        Console.WriteLine(x);
    }
}

static void Main(string[] args) {
    string s = "Splitting an Input String into Substrings";

    Console.WriteLine("Match words that begin with 'S': ");
    findMatch(s, @"\bS\S*");
    Console.ReadKey();
}
}

```

After the match, the code will return the following:

```

Match words that begin with 'S':
The Expression is: \bS\S*
Splitting
String
Substrings

```

The output shows that three words have been matched from the input string. We were matching any words that begin with S and three of them were found.

Consider the next example given below:

```

using System.Text.RegularExpressions;
using System;
namespace RegularExpApp {
    class MyProgram {
        private static void findMatch(string input, string expre) {

```

```

        Console.WriteLine("The input expression is: " + expre);
        MatchCollection mc = Regex.Matches(input, expre);
        foreach (Match x in mc) {
            Console.WriteLine(x);
        }
    }

    static void Main(string[] args) {
        string s = "so she was the same";

        Console.WriteLine("Match words beginning with 's' and ending with
'e:");
        findMatch(s, @"\"bs\S*e\b");
        Console.ReadKey();
    }
}

```

The code will give the following result:

```

Match words beginning with 's' and ending with 'e':
The input expression is: \"bs\S*e\b
she
same

```

In the above example, we are matching the words that begin with s and end with e. We have successfully matched two words in the input string. Here is another example:

```

using System;

using System.Text.RegularExpressions;

namespace RegularExpApp {
    class MyProgram {
        static void Main(string[] args) {
            string text = "Hi learner ";

```

```
string pattern = "\\s+";  
string substitute = " ";  
Regex r = new Regex(pattern);  
string output = r.Replace(text, substitute);  
Console.WriteLine("The input string is: {0}", text);  
Console.WriteLine("The string after replacement is: {0}", output);  
Console.ReadKey();  
}  
}  
}
```

The code returns the following:

```
The input string is: Hi   learner  
The string after replacement is: Hi learner
```

What we are doing in the code is that we are replacing the extra white space. There is a big space between “Hi” and “learner”. The extra one has been replaced or removed.

Chapter 17: The Process of Handling

Exceptions^[39]

Applications usually encounter errors during execution. After the occurrence of an error, the program throws an exception with more information regarding the error. Exceptions should be handled to prevent a program from crashing.

In C#, exceptions are handled using 4 main keywords:

1. *try*- this keyword identifies the block of code in which particular exceptions have been activated. It is then followed by either one or more *catch* blocks.
2. *catch*- a program should catch an exception with an exception handler at a place within the program where you need to handle the problem. The *catch* is a keyword that indicates a place where the exception will be caught.
3. *finally*- the finally block is used for executing a set of statements regardless of whether an exception has been thrown or not. For example, a file must be closed after being opened, whether an exception is thrown or not.
4. *throw*- an exception is normally thrown after the occurrence of a problem. This is done via the *throw* keyword.

A combination of *try* and *catch* is used for catching exceptions. The *try/catch* block has to be placed around a code that may raise an exception. Such code is said to be *protected*, and here is the syntax for using these keywords:

```
try {  
    // statements raising the exception  
} catch( ExceptionName exception1 ) {  
    // The error handling code  
} catch( ExceptionName exception2 ) {  
    // The error handling code  
} catch( ExceptionName eexceptionN ) {  
    // The error handling code
```

```
} finally {  
    // statements to execute  
}
```

One can use many catch statements with the goal of catching many different exceptions if many exceptions are raised by the *try* block.

In C#, exceptions are represented using classes. The *System.Exception* acts as the base class for all exception classes in C# since all other classes are derived from it, either directly or indirectly.

Handling Exceptions

In C#, exceptions can be handled using the *try* and *catch* blocks. With these blocks, we can separate the core program statements from statements for handling errors. We can handle errors using the *try*, *catch* and *finally* keywords.

A division by zero should, for example, raise an exception as it not mathematically supported. Let us create some code to handle this:

```
using System;  
  
namespace ExceptionHandlingApp {  
    class DivisionClass {  
        int answer;  
        DivisionClass() {  
            answer = 0;  
        }  
        public void division(int x, int y) {  
            try {  
                answer = x / y;  
            } catch (DivideByZeroException exception) {  
                Console.WriteLine("Exception caught: {0}", exception);  
            } finally {  
                Console.WriteLine("Answer: {0}", answer);  
            }  
        }  
    }
```

```

    }
    static void Main(string[] args) {
        DivisionClass dc = new DivisionClass();
        dc.division(12, 0);
        Console.ReadKey();
    }
}

```

The code should return the exception given below:

```

Exception caught: System.DivideByZeroException: Attempted to divide by zero.
  at ExceptionHandlingApp.DivisionClass.division (System.Int32 x, System.Int32 y) [0x000000]
Answer: 0

```

We created the *division()* function that takes two arguments, x and y, of type integer. We have then passed 12 and 0 to the function, meaning that we will be dividing 12 by 0. However, this has generated an exception since division by zero is not allowed.

User-Defined Exceptions

C# allows programmers to define their own exceptions. We derive such user-defined exception classes from the *Exception* class. Consider the example given below:

using System;

namespace ExceptionHandlingApp {

class TemperatureTest {

static void Main(string[] args) {

TemperatureClass tc = new TemperatureClass();

try {

tc.displayTemperature();

} catch(TempIsZeroException exception) {

Console.WriteLine("TempIsZeroException: {0}", exception.Message);

}

Console.ReadKey();

}

```

    }
}
public class TempIsZeroException: Exception {
    public TempIsZeroException(string msg): base(msg) {
    }
}
public class TemperatureClass {
    int temp = 0;

    public void displayTemperature() {

        if(temp == 0) {
            throw (new TempIsZeroException("We found Zero Temperature"));
        } else {
            Console.WriteLine("Temperature is: {0}", temp);
        }
    }
}

```

The code will return the following:

```
TempIsZeroException: We found Zero Temperature
```

Nested try-catch

C# allows us to create a block of nested try-catch. In such a case, the exception will be caught in the catch block following the try block in which the exception occurred. Consider the following example:

```

using System;

public class NestedtryCatch
{
    public static void Main()

```



```
{  
Employee emp = null;  
  
try  
{  
try  
{  
emp.EmployeeName = "";  
}  
catch  
{  
Console.WriteLine("The inner catch");  
}  
}  
catch  
{  
Console.WriteLine("The outer catch");  
}  
}  
}
```

```
public class Employee{  
  
public string EmployeeName { get; set; }  
}
```

The code prints the following:

```
The inner catch
```

In case there is no inner catch block with the necessary exception type, the exception will flow to outer catch block until an appropriate exception filter is found. Here is an example:

```
using System;  
public class NestedtryCatch  
{  
public static void Main()  
{  
Employee emp = null;  
  
try  
{  
try  
{  
// This will throw a NullReferenceException  
emp.EmployeeName = "";  
}  
catch (InvalidOperationException innerException)  
{  
Console.WriteLine("The inner catch");  
}  
}  
catch  
{  
Console.WriteLine("The outer catch");  
}  
}
```

```
public class Employee{
```

```
public string EmployeeName { get; set; }
```

```
}
```

In the example given above, the statement *emp.EmployeeName* will generate a `NullReferenceException`, but we don't have a catch block that handles a `NullReferenceException` or an `Exception` type. The outer block will handle this. The code returns the following:

```
The outer catch
```

Chapter 18: File I/O^[40]

A file is characterized by a name and a directory path. Once you open a file, it becomes a *stream*. Files are normally opened for reading or writing purposes.

The stream is simply the sequence of bytes that pass through the communication path. Streams are of two types:

- Input stream
- Output stream

The input stream helps us to read data from a file, that is, a read operation, while the output stream helps us to write data into a file, that is, a write operation.

Input/ Output Classes

C# provides us with a number of classes that we can use for working with files. These classes can be used for accessing directories, files, creating new files, opening existing files and moving files from one directory to another. These classes are defined in the System.IO class. Let us discuss some of these classes:

- BinaryReader- this class helps us in reading primitive data from a binary stream.
- BinaryWriter- this class helps us in writing primitive data in a binary format.
- BufferedStream- this acts as a temporary storage for bytes of streams.
- Directory- this class helps us to manipulate the structure of a directory.
- DriveInfo- this class provides us with information about the drives.
- File- this class is used for manipulation of files.
- FileInfo- helps in performing operations on files.
- FileStream- used for writing and reading from any location in a file.
- MemoryStream- helps in randomly accessing data kept in a memory.
- Path- for performing operations on path information.
- StreamReader- reads characters from a stream of bytes.
- StreamWriter- writes characters to a stream.

- StringReader- reads from a string buffer.
- StringWriter- writes into a string buffer.

FileStream Class

This class is defined in the System.IO namespace and it helps us to read from and write to files. We can also use it to close files^[41].

For you to be able to use this class, you have to create its object. This instance can then be used for creating new files and opening existing files. Here is how you can create an instance of the FileStream class:

```
FileStream <object> = new FileStream( <file>, <FileMode Enumerator>,
<FileAccess Enumerator>, <FileShare Enumerator>);
```

For example, suppose we need to read a file names *names.txt*. We can create a FileStream object named *FS* and use it for this purpose. This is demonstrated below:

```
FileStream FS = new FileStream("names.txt", FileMode.Open, FileAccess.Read,
FileShare.Read);
```

The FileMode is an enumerator that defines a number of methods that can be used for opening files. These methods include the following:

- Append – this method opens an existing file then puts the cursor at the end of the file, or it creates a new file if the specified file doesn't exist.
- Create – for creating a new file.
- CreateNew – It instructs the operating system to create a new file.
- Open – for opening an existing file.
- OpenOrCreate – It instructs the operating system to open a file if it is in existence or create a new file if it doesn't exist.
- Truncate – for opening an existing file and truncating its size to zero bytes.

The FileAccess is an enumerator that comes with a number of methods including Read, Write and ReadWrite.

The FileShare enumerator comes with the following members:

- Inheritable – helps a file handle in passing inheritance to child processes.
- None – It disables sharing of the current file.
- Read – It opens a file for reading.

- ReadWrite – opens a file for reading and writing.
- Write – opens a file for writing.

The FileStream class can be used as demonstrated below:

```
using System.IO;
using System;
namespace FileApp {
    class MyProgram {
        static void Main(string[] args) {
            FileStream FS = new FileStream("names.dat", FileMode.OpenOrCreate,
                FileAccess.ReadWrite);
            for (int x = 1; x <= 10; x++) {
                FS.WriteByte((byte)x);
            }
            FS.Position = 0;
            for (int x = 0; x <= 10; x++) {
                Console.Write(FS.ReadByte() + " ");
            }
            FS.Close();
            Console.ReadKey();
        }
    }
}
```

The code will return the following output after execution:

```
1 2 3 4 5 6 7 8 9 10 -1
```

Appending Text Lines^{[\[42\]](#)}

Sometimes, you may need to append a number of text lines to a file. This can be done by calling the *AppendAllLines()* method. Here is an example:

```
string multipleLines = "The first line." + Environment.NewLine +
```

```
"The second line." + Environment.NewLine +  
"The third line.";
```

// To open the file named Myfile.txt then append the above lines. If the file does not exist, a new one will be created.

```
File.AppendAllLines(@"C:\MyFile.txt",  
multipleLines.Split(Environment.NewLine.ToCharArray()).ToList<string>());
```

The file will be opened and the specified lines will be appended to the file.

Appending a String

The *File.AppendAllText()* method can allow you to append a string to a file in only a single line of code. This is demonstrated below:

//Open the file named MyFile.txt then append text to it. If the file does not exist, create a new one and open it for writing.

```
File.AppendAllText(@"C:\ MyFile.txt", "This string will be written into the  
file");
```

Overwriting Text

If you need to overwrite a file, use the *File.WriteAllText()* method. The method will delete the text in the file and replace it with the one that you specify. This method can be used as demonstrated below:

// To open the file MyFile.txt and write the text into it. If the file does not exist, a new one will be created and opened for writing.

```
File.WriteAllText(@"C:\MyFile.txt", "This text will be used for replacing the  
current text in the file.");
```

With the Static File Class, one can perform various operations. This is demonstrated below:

//Check whether the file exists at the specified location or not

```
bool isFileExists = File.Exists(@"C:\ MyFile.txt"); // it will return false
```

//Copy MyFile.txt as the new file MyNewFile.txt

```
File.Copy(@"C:\MyFile.txt", @"D:\MyNewFile.txt");
```

// Check when the file was lastly accessed

DateTime lastAccessTime = File.GetLastAccessTime(@"C:\MyFile.txt");

//get the last time the file was written

DateTime lastWriteTime = File.GetLastWriteTime(@"C:\MyFile.txt");

// Transfer the file to a new location

File.Move(@"C:\MyFile.txt", @"D:\MyFile.txt");

//Open file and returns FileStream for reading bytes from the file

FileStream fs = File.Open(@"D:\MyFile.txt", FileMode.OpenOrCreate);

//Open the file and return a StreamReader for reading a string from the file

StreamReader sr = File.OpenText(@"D:\MyFile.txt");

//Delete the file

File.Delete(@"C:\MyFile.txt");

The above shows that with the Static File class, it becomes easy for us to work with physical files. However, for more flexibility, one can use the *FileInfo* class.

StreamReader Class

This class helps us to read data from a text file. This class inherits the Stream base class. It also inherits the TextReader class, an abstract base class for reading a series of characters. The following are the popular methods provided by this class:

- public override void Close()- the method closes the StreamReader object and the stream, then any system resources that were being used by the reader are released.
- public override int Peek()- this method will return the next character that is available without consuming it.

- `public override int Read()`- this method will read the next character in an input stream then advance the position of the character by 1.

Let us demonstrate how we can use this class to read from a file named *names.txt*:

using System.IO;

using System;

namespace IOApplication {

class MyProgram {

static void Main(string[] args) {

try {

// Create a StreamReader instance for reading from a file.

// The using statement will close the StreamReader.

using (StreamReader sr = new StreamReader("c:/names.txt")) {

string line;

// Read the show lines from the file until

// you reach the end of the file.

while ((line = sr.ReadLine()) != null) {

Console.WriteLine(line);

}

}

} catch (Exception exception) {

// Information the user about what went wrong

Console.WriteLine("File couldn't be read:");

Console.WriteLine(exception.Message);

}

Console.ReadKey();

}

```
}  
}
```

The code will read the text written in the file *names.txt* and print it.

StreamWriter Class^[43]

This class inherits the `TextWriter` abstract class that represents a write, capable of writing a series of characters. Here are the popular methods for this class:

- `public override void Close()`- this method will close the current object of the `StreamWriter` as well as the underlying stream.
- `public override void Flush()`- this method will clear buffers for current writer and makes any buffered data to be written to the stream.
- `public virtual void Write(bool value)`- this method writes a textual representation of a Boolean value into the text stream or string.
- `public override void Write(char value)`- for writing a character to a stream.
- `public virtual void Write(decimal value)`- this method writes a textual representation of a decimal value to a text stream or string.
- `public virtual void Write(double value)`- writes a text representation of 8-byte floating point value into a text stream or string.
- `public virtual void Write(int value)`- writes a text representation of 4-byte signed integer into a text stream or string.
- `public override void Write(string value)`- for writing a string to a stream.
- `public virtual void WriteLine()`- for writing a line terminator to a text stream or string.

Let us demonstrate how we can use the `StreamWriter` class to write text data:

```
using System.IO;
```

```
using System;
```

```
namespace IOApp {
```

```
class MyProgram {
```

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

```
    string[] students = new string[] { "Nicholas Samuel", "Michelle Boss" };
```

```

using (StreamWriter sw = new StreamWriter("students.txt")) {

    foreach (string x in students) {
        sw.WriteLine(x);
    }
}

// Read and display every line from the file.
string text = "";
using (StreamReader sr = new StreamReader("students.txt")) {
    while ((text = sr.ReadLine()) != null) {
        Console.WriteLine(text);
    }
}
Console.ReadKey();
}
}

```

The code will print the following result after execution:

```

Nicholas Samuel
Michelle Boss

```

BinaryReader Class^[44]

This class can be used for reading binary data from a file. We should first create a BinaryReader object by passing an object of FileStream to its constructor. This class comes with a number of methods including the following:

- public override void Close()- for closing the BinaryReader object as well as the underlying stream.
- public virtual int Read()- for reading characters from the underlying stream and advancing the stream's current position.

- public virtual bool ReadBoolean()- for reading a Boolean value from the current stream and advancing the stream's current position by a byte.
- public virtual byte ReadByte()- for reading the next byte from current stream into byte array and advancing current position by a similar number of bytes.

BinaryWriter Class^[45]

This class helps in writing binary data into a stream. To create a BinaryWriter object, we pass a FileStream object to the constructor. This class comes with a number of methods including the following:

- public override void Close()- for closing the BinaryWriter object as well as the underlying stream.
- public virtual void Flush()-this method will clear buffers for current writer and makes any buffered data to be written to the device.
- public virtual void Write(bool value)- for writing a one-byte Boolean value into the current stream. 1 represents true while 0 represents false.

Let us create an example that demonstrates how to read and write binary data:

using System;

using System.IO;

namespace BinaryFileApplication {

class MyProgram {

static void Main(string[] args) {

BinaryWriter bw;

BinaryReader br;

int x = 12;

double db = 1.24658;

bool bl = false;

string st = "Hello world";

//create a file

```
try {
    bw = new BinaryWriter(new FileStream("testdata", FileMode.Create));
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to create the file.");
    return;
}

//write to the file
try {
    bw.Write(x);
    bw.Write(db);
    bw.Write(bl);
    bw.Write(st);
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to write to the file.");
    return;
}

bw.Close();

//read from the file
try {
    br = new BinaryReader(new FileStream("testdata", FileMode.Open));
} catch (IOException exception) {
    Console.WriteLine(exception.Message + "\n Unable to open the file.");
    return;
}

try {
    x = br.ReadInt32();
```

```

        Console.WriteLine("Integer data: {0}", x);
        db = br.ReadDouble();
        Console.WriteLine("Double data: {0}", db);
        bl = br.ReadBoolean();
        Console.WriteLine("Boolean data: {0}", bl);
        st = br.ReadString();
        Console.WriteLine("String data: {0}", st);
    } catch (IOException exception) {
        Console.WriteLine(exception.Message + "\n Unable to read from the
file.");
        return;
    }
    br.Close();
    Console.ReadKey();
}
}
}

```

The code will give the following result after execution:

```

Integer data: 12
Double data: 1.24658
Boolean data: False
String data: Hello world

```

Chapter 19: Delegates

We can have a function with more than one parameters from different data types. However, we may sometimes need to pass a certain function as a parameter. How can C# handle the event handler or callback functions? This is done via *delegates*^[46].

A delegate is taken as a pointer to a function. It is a reference data type that holds reference of a method. All delegates are derived from *System.Delegate* class implicitly. We use delegates to implement events and callback methods^[47].

Declaring a Delegate

The way we declare a delegate determine the methods that the delegate can reference. A delegate may refer to a method, which has the same signature as the delegate^[48]. We use the *delegate* keyword to declare a delegate. The following syntax is used for the declaration:

<access modifier> delegate <return type><delegate_name>(<parameters>)

Here is an example:

public delegate void Display(int value);

Above, we have declared a *Display* delegate. We can use this delegate to point to a method with same return type and parameters that have been declared with the *Display* delegate.

Consider the example given below:

using System;

public class MyProgram

{

public delegate void Display(int num);

public static void Main()

{

// Display delegate points to the DisplayNumber

Display displayDel = DisplayNumber;

```

    displayDel(100000);
    displayDel(200);

    // Display delegate points to DisplayMoney
    displayDel = DisplayMoney;

    displayDel(50000);
    displayDel(20);
}

public static void DisplayNumber(int x)
{
    Console.WriteLine("Number: {0,-12:N0}",x);
}

public static void DisplayMoney(int amount)
{
    Console.WriteLine("Money: {0:C}", amount);
}
}

```

The code will print out the following:

```

Number: 100,000
Number: 200
Money: $50,000.00
Money: $20.00

```

We have created a delegate named *Display* that accepts a parameter of type integer and returns void. With the `Main()` method, we have created a variable of type *Display* and a method named *DisplayNumber* has been assigned to it. After invoking the *Display* delegate, the *DisplayNumber* method will be called also, if the *Display* delegate variable is assigned to *DisplayMoney* method, the *DisplayMoney* method will be invoked.

Also, it is possible for us to create a delegate object using the *new* keyword. In this case, we have to specify the name of the method as shown below:

```
Display displayDel = new Display (DisplayNumber);
```

Invoking a Delegate

A delegate is a reference to a method; hence we can invoke it simply like a method. When a delegate is invoked, the method that is referred to will, in turn, be invoked.

There are two ways through which we can invoke a delegate: using the () operator or using *Invoke()* method of delegate. Let us demonstrate this using an example:

```
Display displayDel = DisplayNumber;
```

```
displayDel.Invoke(5000);
```

```
//or
```

```
printDel(10000);
```

The two ways that we can use to invoke a delegate have been shown above.

Passing a Delegate as a Parameter

A method may have a parameter of delegate type and it can invoke the delegate parameter. Here is an example of a delegate parameter:

```
public static void DisplayHelper(Display delegateFunc, int numToDisplay)  
{  
    delegateFunc(numToDisplay);  
}
```

In the example given above, the DisplayHelper function has a delegate parameter of Display type and it has been invoked as a function using the statement given below:

```
delegateFunc(numToDisplay);
```

Let us give another example demonstrating how to use the *DisplayHelper* method and a delegate type parameter:

```
using System;
```

```
public class MyProgram
```

```

{
public delegate void Display(int num);

public static void Main()
{
DisplayHelper(DisplayNumber, 500);
    DisplayHelper(DisplayMoney, 200);
}

public static void DisplayHelper(Display delegateFunc, int numToDisplay)
{
    delegateFunc(numToDisplay);
}

public static void DisplayNumber(int val)
{
    Console.WriteLine("Number: {0,-12:N0}",val);
}

public static void DisplayMoney(int amount)
{
    Console.WriteLine("Money: {0:C}", amount);
}
}

```

The code will return the following output:

```

Number: 500
Money: $200.00

```

Multicast Delegate

It is possible for a delegate to point to multiple functions. Any delegate that points to many functions is referred to as a *multicast delegate*. Use the + operator to add a function to a delegate object and the – operator to remove an existing function from a delegate object.

Consider the example given below:

```
using System;  
  
public class MyProgram  
{  
  
public delegate void Display(int value);  
  
  
public static void Main()  
{  
Display displayDel = DisplayNumber;  
displayDel += DisplayHexadecimal;  
displayDel += DisplayMoney;  
  
displayDel(1000);  
displayDel -= DisplayHexadecimal;  
displayDel(2000);  
}  
  
public static void DisplayNumber(int x)  
{  
Console.WriteLine("Number: {0,-12:N0}",x);  
}  
  
  
public static void DisplayMoney(int amount)  
{  
Console.WriteLine("Money: {0:C}", amount);
```

```
}  
  
public static void DisplayHexadecimal(int dec)  
{  
    Console.WriteLine("Hexadecimal: {0:X}", dec);  
}  
}
```

The code will print the following as the result:

```
Number: 1,000  
Hexadecimal: 3E8  
Money: $1,000.00  
Number: 2,000  
Money: $2,000.00
```

In the above example, we have the Display pointing to three methods namely DisplayNumber, DisplayMoney and DisplayHexadecimal. This makes it a multicast delegate. This means that when the displayDel is invoked, it will, in turn, invoke all the three methods sequentially.

Chapter 20: Multithreading in C#

A thread defines a control flow. Thread is a basic unit that the operating system assigns a thread. The execution of a thread is independent within a program.

A single process is executed using one thread. Such process is known as single – threaded process. Only one task can be performed at a time. The user has to wait for the task to complete before executing new task.

For executing more than one thread at a time, multiple threads are created. The process creating two or more threads is known as multithreading^[49].

Life cycle of a thread

The life cycle begins when the object of **System.Threading.Thread** class is created. The life ends as soon as the task is completed. There are various states in the life cycle of a thread.

- **Unstarted State:** When the instance of the **Thread** class is created, the thread enters in unstarted state.
- **Ready State:** The thread is in this state until the program calls the **Start()** method.
- **Not Runnable State:** Some of the elements that make a thread not to be in a runnable state include:
 1. **Waiting:** The **Wait()** method is called to make the thread for a specified condition
 2. **Blocked:** The thread is blocked by an I/O operation
 3. **Sleeping:** The **Sleep()** method is called to put the thread in sleeping mode.
- **Dead State:** Once the thread completes its execution or aborted, it is placed in a dead state

Main thread

When working with threads, we make use of the **System.Threading.Thread** class. The main thread is created as soon as the program starts execution. The **Thread** class is used for creating threads. They are known as child threads. The user can access the main thread by using the **CurrentThread** property of the Thread class.

Example:

```
using System;
namespace thread
{
    class MainThread
    {
        static void Main(string[] args)
        {
            Thread t1 = new Thread();
            t1.Name="Thread1";
            Console.WriteLine("Thread is:{0}",t1.Name);
            Console.Read();
        }
    }
}
```

On compiling and executing the code, the output is:

Thread is: Thread1

Properties and methods of the Thread class

Properties^[50]:

- **IsAlive:** The value showing the execution status of the current thread
- **CurrentThread:** The current running thread is retrieved
- **CurrentContext:** The current context in which the thread is executing is retrieved
- **ExecutionContext:** The ExecutionContext object contains information about different contexts
- **Name:** Gets or sets the name of the thread

- **ThreadState:** The value containing states of the current thread
- **Priority:** It gets or sets the value showing the scheduling priority of a thread

Methods:

- **public static void BeginThreadAffinity():** The host is to about to execute instructions depending on the current physical operating system thread.
- **public void Abort():** The ThreadAbortException is raised in the thread on which it is invoked.
- **public void interrupt():** The thread present in the WaitSleepJoin state is interrupted
- **public static AppDomain GetDomain():** A unique domain identifier is returned
- **public static void MemoryBarrier():** The processor executes the current thread. The instructions cannot be reordered.
- **public void Start():** It starts the thread
- **public static bool Yield():** The calling thread to yield execution to another thread which is ready to run on the processor

Creating and managing threads^[51]

The extended thread class creates a thread. The extended thread class calls the **Start()** method to start the child thread execution.

Example:

```
using System;
using System.Threading;

namespace MultipleThread
{
    class ThreadProgram
    {
```

```

public static void CallChild()
{
    Console.WriteLine("Start child thread");
}

static void Main(string[] args)
{
    ThreadStart child1 = new ThreadStart(CallChild);
    Console.WriteLine("Creating child thread");
    Thread child2 = new Thread(child1);
    child2.Start();
    Console.Read();
}
}
}

```

On compiling and executing the code, the output is:

Start child thread

Creating child thread

Managing Threads

When there is a need to pause a thread for a period of time so that another thread can execute, the **Thread.Sleep()** method is used. The method takes a single argument stating time in milliseconds.

Example:

```

using System;
using System.Threading;

namespace Multithreaded
{

```



```
class Program
{
    public static void ChildThread()
    {
        Console.WriteLine("Start child thread");
        int sleeptime = 4000;

        Console.WriteLine("Thread sleeping for {0}  seconds",sleeptime /
1000);

        Thread.Sleep(sleeptime);
        Console.WriteLine("Resume child thread");

    }
    public static void Main()
    {
        ThreadStart t1 = new ThreadStart(ChildThread);
        Console.WriteLine("child thread created");
        Thread child1 = new Thread(t1);
        child1.Start();
        Console.Read();
    }

}
}
```

On compiling and executing the code, the output is:

Start child thread

Thread sleeping for 4 seconds

Resume child thread

child thread created

Destroying threads

The **Thread.Abort()** method is used to destroy the thread. The **ThreadAbortException** is thrown when the thread is destroyed. The exception is not caught and is sent to the **finally** block.

Example:

```
using System;
using System.Threading;

namespace ThreadDemo
{
    class Program
    {
        public static void ChildThread()
        {
            try
            {
                Console.WriteLine("Child Thread started");
                for(int j = 0; j <= 10; j++)
                {
                    Thread.Sleep(1000);
                    Console.WriteLine("Child thread finished");
                }
            }
        }
    }
}
```

```
    }  
}  
catch(ThreadAbortException e)  
{  
    Console.WriteLine("Exception caught");  
}  
finally  
{  
    Console.WriteLine("Exception is not handled");  
}  
}  
  
public static void Main()  
{  
    ThreadStart t1 = new ThreadStart(ChildThread);  
    Console.WriteLine("Creating child thread");  
    Thread t2 = new Thread(t1);  
    t1.Start();  
  
    //main thread is stopped  
    Thread.Sleep(2000);  
  
    //child thread aborted  
    Console.WriteLine("Aborting child thread");  
    t2.Abort();  
    Console.Read();  
}  
}
```

```
}
```

c, the output is:

Creating child thread

Child Thread started

0

1

Aborting child thread

Exception caught

Exception is not handled

Chapter 21: Event

An event is something that is expected to happen. In C#, events are user actions like click, key press, mouse movements or occurrences such as system-generated notifications^[52]. An application should respond to events once they occur. A good example is the occurrence of an interrupt. Events are used to facilitate inter-process communications.

Events are declared in a class and associated with an event handler via delegates in the same class or in another class. The class with the event is used for publishing the event. Such is known as the *publisher* class. The class accepting the event is known as the *subscriber class*. This means that events rely on a publisher-subscriber model.

The publisher is the object with the definition of both the event and the delegate. The association between the event and the delegate is defined in this object. An object of the publisher class invokes the event which is in turn notified to the other objects.

The subscriber is the object that accepts the event and offers the event handler. The publisher class has a delegate that invokes the method (event handler) of the subscriber class.

Event Declaration

For an event to be declared inside a class, you should first declare a delegate type for the event. It is after this that you can declare the event using the *event* keyword^[53]. Here is an example:

```
public delegate void myEvent();
```

```
public event myEvent myEvent;
```

Thus, we have used the *event* keyword to make it an event. Here is a complete example:

```
using System;
```

```
public class DisplayHelper
```

```
{
```

```
    // declare the delegate
```

```
    public delegate void BeforeDisplay();
```

```
//declare an event of type delegate
public event BeforeDisplay beforeDisplayEvent;

public DisplayHelper()
{

}

public void DisplayNumber(int x)
{
    //call the delegate method before moving to display
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Number: {0,-12:N0}", x);
}

public void PrintDecimal(int dec)
{
    if (beforeDisplayEvent != null)
        beforeDisplayEvent();

    Console.WriteLine("Decimal: {0:G}", dec);
}

public void DisplayMoney(int amount)
{
```

```

        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Money: {0:C}", amount);
    }

    public void DisplayTemperature(int x)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Temperature: {0,4:N1} F", x);
    }

    public void DisplayHexadecimal(int dec)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Hexadecimal: {0:X}", dec);
    }
}

```

The *DisplayHelper* is a publisher class responsible for publishing the *beforeDisplay* event. After every *display* method, it first checks to determine whether the *beforeDisplayEvent* is not null then it calls the *beforeDisplayEvent()* method.

We now need to create a subscriber. Consider the class given below:

```

using System;

public class MyProgram

```

```

{

public static void Main()
{
    NumberClass nc = new NumberClass(200);
    nc.DisplayMoney();
    nc.DisplayNumber();
}
}

class NumberClass
{
    private DisplayHelper _displayHelper;

    public NumberClass(int x)
    {
        _value = x;

        _displayHelper = new DisplayHelper();
        //subscribe to the beforeDisplayEvent event
        _displayHelper.beforeDisplayEvent += displayHelper_beforeDisplayEvent;
    }
    //beforeDisplayevent handler
    void displayHelper_beforeDisplayEvent()
    {
        Console.WriteLine("BeforeDisplayEventHandler: DisplayHelper will print
a value");
    }
}

```



```
private int _value;
```

```
public int Value
```

```
{
```

```
    get { return _value; }
```

```
    set { _value = value; }
```

```
}
```

```
public void DisplayMoney()
```

```
{
```

```
    _displayHelper.DisplayMoney(_value);
```

```
}
```

```
public void DisplayNumber()
```

```
{
```

```
    _displayHelper.DisplayNumber(_value);
```

```
}
```

```
}
```

```
public class DisplayHelper
```

```
{
```

```
    // declare a delegate
```

```
    public delegate void BeforeDisplay();
```

```
    //declare an event of type delegate
```

```
    public event BeforeDisplay beforeDisplayEvent;
```

```
public DisplayHelper()
```

```
{
```

```
}
```

```
public void DisplayNumber(int y)
```

```
{
```

```
//call a delegate method before printing
```

```
    if (beforeDisplayEvent != null)
```

```
        beforeDisplayEvent();
```

```
    Console.WriteLine("Number: {0,-12:N0}", y);
```

```
}
```

```
public void DisplayDecimal(int dec)
```

```
{
```

```
    if (beforeDisplayEvent != null)
```

```
        beforeDisplayEvent();
```

```
    Console.WriteLine("Decimal: {0:G}", dec);
```

```
}
```

```
public void DisplayMoney(int amount)
```

```
{
```

```
    if (beforeDisplayEvent != null)
```

```
        beforeDisplayEvent();
```

```

        Console.WriteLine("Money: {0:C}", amount);
    }

    public void DisplayTemperature(int y)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Temperature: {0,4:N1} F", y);
    }

    public void DisplayHexadecimal(int dc)
    {
        if (beforeDisplayEvent != null)
            beforeDisplayEvent();

        Console.WriteLine("Hexadecimal: {0:X}", dc);
    }
}

```

The code returns the following result:

```

BeforeDisplayEventHandler: DisplayHelper will print a value
Money: $200.00
BeforeDisplayEventHandler: DisplayHelper will print a value
Number: 200

```

All subscribers must have a handler function which will be called after the publisher has raised an event.

Chapter 22: Hints and Important Resources

Although I believe that if you follow the steps of this book you will have a very thorough and firm grasp on C# in a short period, there are undoubtedly still questions you may have that haven't been addressed.

The wonderful thing about the tech-savvy world that we live in today is that this book is not your only resource to learning and mastering the programming and coding of C#.

As I believe you now feel ready and able to take on many challenges you wish to address with your coding in the C# format, you may come to a crossroads where you are unsure of what the next potential step in problem-solving is. This chapter should serve as your guide to finding any answers to questions that you have not received within the pages of this book.

First and foremost, I encourage you to look through your code for simple errors in spacing or capitalization.

Sometimes if you are used to coding in a different language, you may have accidentally put a period where a semicolon should be in C# or even put brackets where they should not be.

These are the types of errors that I find to be the most frustrating and typically are the mistakes that I have made. Essentially, always double check your work when you get to a frustrating point that seems to keep giving you an error message.

Also, if you try this tactic and it doesn't work and you are ready to give up—take a break! Come back to your project after you have let yourself relax and take your mind off your project. When you come back and look at it with refreshed eyes you may find the error of your ways.

Another resource that I utilize as well are online forums for C# in particular. Although some programming forums offer great general advice, I encourage you to Google things like “C# help forum” in order to find others with potentially sage advice on how to solve an issue.

There is a likelihood that you may not be the only individual to ever come across a certain speed hump that may be preventing you from completing a portion of your project. These can be great and accessible ways to quickly find an answer by using the power of the internet.

Although I already mentioned these next resources, I want to reiterate just how important code editors and libraries can potentially be.

Just as within any programming language, you can find seemingly endless answers in the libraries that will allow you to code for many different situations. As I suggested before, however, please have a firm grasp on the process of coding and programming before you venture into the world of using code.

If something changes within the code and you can't find the mistake, you could potentially have irreparable damage to your program or project. Code editors can help you find these mistakes before making your code live, so I highly suggest that you consistently utilize this resource until you are a C# master—maybe even then just in case you fall victim to human error.

If you find that you need a better visual for any of the content within this book, or outside of the scope—I encourage you to visit my favorite video site—YouTube!

Just search for a key phrase or word that you are attempting to figure out during your coding and programming and I guarantee that you will find a helpful video. Sometimes, it may not be the very first video that you come across but it certainly will be there.

You may already have a preferred YouTuber that you go to for examples in a different language—maybe go look at the other users he or she subscribes to and see if their content can be helpful to you.

Never think that just because you are struggling it is the end of your project as a whole.

I can tell you from personal experience that allowing yourself to fully explore and problem-solve when issues like this come up gives you an opportunity to create new pathways in your brain that will serve you well with your struggles to come.

Think back on the ways that you have solved other issues in your life, or programming experiences, and allow yourself to strategize how and why you can apply it to the situation at hand.

You are your own best ally when it comes to knowing how you think and what you want to accomplish. Use your resources to the best of your ability and become as knowledgeable as you can—the power of knowledge and learning is quite a remarkable skill.

Programming and coding can be seen as a problem-solving matter that you use to create programs that perform a specific purpose that can make things easier for you and the lives of others. Always work towards an achievable goal for your skill set,

while continuing to push yourself to learn more and get to the next level of your functioning.

You can get more detailed information about the C# language from the following reference links.

- [Visual Studio Application](#) – The IDE for creating C# applications.
- [C# \(Programming guide \)](#) – An overview of C# programming language
- [C# Programming](#) – The information about the C# features using .NET framework is explained
- [Mono](#) – Cross-platform applications can be easily created using the software.

Conclusion

Now that this book has come to a close, you have all the information and knowledge of skills in order to be an excellent C# programmer and coder. It is important to me that you use this book as the foundation for your skills. I have worked hard to provide what I think is a guide that is comprehensive and easy to understand. Without languages like C#, we could not have the amazing technology both in our personal and business lives.

When you think about the opportunities that technological skills such as programming create in your life, I believe it is an invaluable skill. Knowing about the very basics of the computer system setup is something that can open doors for you in a professional sense, with more complex and high paying jobs.

However, in your personal life, you can use the critical thinking and problem-solving skills to help enhance your own gaming experience on your computer or other fields outside of technology. With the quickness you learned C#, you should feel very confident and fulfilled in your accomplishment!

Not everyone can learn an entirely new language in a month's time, which you have just simply and easily done through exploring a new virtual world. It is an accomplishment that you should be happy about, and really start to create the projects that you desire. As an object-oriented language, I am sure you have some excellent new ideas for contributions within the software community. Or perhaps you just want to be able to manipulate the Windows computer you have!

The descriptions and sections that I have broken down in this book should have allowed you to grasp and connect different pieces and parts of C# in a way that, when you reached this final conclusion, you are able to feel as though you can pass along knowledge to someone else.

I hope this language showed you that it encourages gameplay programming, through the many different avenues used in this language to define and separate different aspects that may touch (classes and statements for example) but are not interchangeable.

It may appear to be an overwhelming confusion as you get used to the nuances— the more you test out your skills and put examples of code to the test, the more you will come to realize the full impact and advantages these offer you.

Thank you for picking up my book and giving it a chance to teach you something

new! I would like to make the basics of every topic something that any person can learn quickly and easily. I hope you use this as a reference guide for your future projects and that you have been able to absorb the material and concepts with the help of the examples within the book. I want to end the book with a few more pearls of wisdom I find important to reiterate from my own experiences.

Always test your projects in a test zone before putting them to life, especially if irrevocable damage could be done! Remember that you should be enjoying yourself when you are programming and coding because you are creating something complex by putting the pieces together yourself. Lastly, have fun and enjoy yourself while doing it!

Bibliography

AG, A. (n.d.). *Delegates and Events in C# / .NET*. [online] Akadia.com. Available at: https://www.akadia.com/services/dotnet_delegates_and_events.html [Accessed 15 Feb. 2019].

Bodnar, J. (2019). *Methods in C# - working with CSharp methods*. [online] Zetcode.com. Available at: <http://zetcode.com/lang/csharp/methods/> [Accessed 15 Feb. 2019].

Bodnar, J. (2019). *Operators in C# - describing CSharp operators and expressions*. [online] Zetcode.com. Available at: <http://zetcode.com/lang/csharp/operators/> [Accessed 15 Feb. 2019].

Codescracker.com. (n.d.). *C# Program Structure*. [online] Available at: <https://codescracker.com/c-sharp/c-sharp-program-structure.htm> [Accessed 15 Feb. 2019].

Csharp.net-informations.com. (n.d.). *C# File Operations Tutorial*. [online] Available at: <http://csharp.net-informations.com/file/csharp-file-tutorial.htm> [Accessed 15 Feb. 2019].

Csharp.net-tutorials.com. (n.d.). *Variables - The complete C# tutorial*. [online] Available at: <https://csharp.net-tutorials.com/basics/variables/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Arrays - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Casting and type conversions - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Delegates - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *goto statement - C# Reference*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2015). *Passing Parameters - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Classes - C# Programming Guide*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Default values table - C# Reference*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (2018). *Using threads and threading*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *BinaryReader Class (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *BinaryWriter Class (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

Docs.microsoft.com. (n.d.). *File.AppendText(String) Method (System.IO)*. [online] Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.io.file.appendtext?view=netframework-4.7.2> [Accessed 15 Feb. 2019].

Download.microsoft.com. (n.d.). *Microsoft Download Center: Windows, Office, Xbox & More*. [online] Available at: <http://download.microsoft.com/> [Accessed 15 Feb. 2019].

En.wikipedia.org. (2019). *C Sharp (programming language)*. [online] Available at: [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)) [Accessed 15 Feb. 2019].

GeeksforGeeks. (n.d.). *C# | Arrays - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-arrays/> [Accessed 15 Feb. 2019].

GeeksforGeeks. (n.d.). *C# | Methods - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/c-sharp-methods/> [Accessed 15 Feb. 2019].

Guru99.com. (n.d.). [online] Available at: <https://www.guru99.com/c-sharp-data-types.html> [Accessed 15 Feb. 2019].

Heydrick, C. (2014). *C# Struct Sizes*. [online] Chris Heydrick: Serial Hobbyist.

Available at: <https://chrisheydrick.com/2014/12/17/c-struct-sizes/> [Accessed 15 Feb. 2019].

Jonskeet.uk. (n.d.). *Parameter passing in C#*. [online] Available at: <http://jonskeet.uk/csharp/parameters.html> [Accessed 15 Feb. 2019].

Khorshidnia, S. (2013). *Recursive methods using C#*. [online] Codeproject.com. Available at: <https://www.codeproject.com/Articles/142292/Recursive-methods-in-Csharp> [Accessed 15 Feb. 2019].

Mayo, J. (n.d.). *Lesson 5: Methods - C# Station*. [online] C# Station. Available at: <https://csharp-station.com/Tutorial/CSharp/Lesson05> [Accessed 15 Feb. 2019].

Mkhitaryan, A. (2017). *Why Is C# Among The Most Popular Programming Languages in The World?*. [online] Medium. Available at: <https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb> [Accessed 15 Feb. 2019].

Mono-project.com. (2019). *Home | Mono*. [online] Available at: <http://www.mono-project.com/> [Accessed 15 Feb. 2019].

Msdn.microsoft.com. (2017). *C# Programming Guide*. [online] Available at: <https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx> [Accessed 15 Feb. 2019].

o7planning.org. (n.d.). *C# Multithreading Programming Tutorial*. [online] Available at: <https://o7planning.org/en/10553/csharp-multithreading-programming-tutorial> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# foreach loop (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/foreach-loop> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# if, if...else, if...else if and Nested if Statement (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/if-else-statement> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# Operators: Arithmetic, Comparison, Logical and more..*. [online] Available at: <https://www.programiz.com/csharp-programming/operators> [Accessed 15 Feb. 2019].

Programiz.com. (n.d.). *C# switch Statement (With Examples)*. [online] Available at: <https://www.programiz.com/csharp-programming/switch-statement> [Accessed 15 Feb. 2019].

Spasojevic, M. (2018). *C# Basics - C# Type Conversions (Implicit and Explicit Conversion)*. [online] Code Maze. Available at: <https://code-maze.com/csharp-basics->

type-conversion/ [Accessed 15 Feb. 2019].

Tutorialspoint.com. (n.d.). *Online Csharp Compiler - Online Csharp Editor - Online Csharp IDE - Csharp Coding Online - Practice Csharp Online - Execute Csharp Online - Compile Csharp Online - Run Csharp Online*. [online] Available at: https://www.tutorialspoint.com/compile_csharp_online.php [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# Files & Directories*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-file> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# for loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-for-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# keywords*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-keywords> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *C# while loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-while-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Do-While loop*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-do-while-loop> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Event in C#*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-event> [Accessed 15 Feb. 2019].

Tutorialsteacher.com. (n.d.). *Struct in C#*. [online] Available at: <https://www.tutorialsteacher.com/csharp/csharp-struct> [Accessed 15 Feb. 2019].

Visual Studio. (n.d.). *Downloads | IDE, Code, & Team Foundation Server | Visual Studio*. [online] Available at: <https://www.visualstudio.com/downloads/download-visual-studio-vs> [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Arrays*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_arrays.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Classes*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_classes.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Delegates*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_delegates.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Encapsulation*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Events*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_events.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Exception Handling*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# File I/O*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_file_io.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Inheritance*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_inheritance.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Multithreading*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_multithreading.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Polymorphism*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Regular Expressions*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_regular_expressions.htm [Accessed 15 Feb. 2019].

www.tutorialspoint.com. (n.d.). *C# Structures*. [online] Available at: https://www.tutorialspoint.com/csharp/csharp_struct.htm [Accessed 15 Feb. 2019].

[1] <https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb>

[2] <https://codescracker.com/c-sharp/c-sharp-program-structure.htm>

[3] https://www.tutorialspoint.com/compile_csharp_online.php

[4] <https://www.tutorialsteacher.com/csharp/csharp-keywords>

[5] <http://download.microsoft.com/>

[6] <https://www.guru99.com/c-sharp-data-types.html>

[7] <https://chrisheydrick.com/2014/12/17/c-struct-sizes/>

[8] <https://csharp.net-tutorials.com/basics/variables/>

[9] <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/default-values-table>

[10] <https://code-maze.com/csharp-basics-type-conversion/>

[11] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions>

[12] <https://www.programiz.com/csharp-programming/operators>

[13] <http://zetcode.com/lang/csharp/operators/>

- [14] <https://www.programiz.com/csharp-programming/if-else-statement>
- [15] <https://www.programiz.com/csharp-programming/switch-statement>
- [16] <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/goto>
- [17] <https://www.tutorialsteacher.com/csharp/csharp-while-loop>
- [18] <https://www.tutorialsteacher.com/csharp/csharp-do-while-loop>
- [19] <https://www.tutorialsteacher.com/csharp/csharp-for-loop>
- [20] <https://www.programiz.com/csharp-programming/foreach-loop>
- [21] <https://csharp-station.com/Tutorial/CSharp/Lesson05>
- [22] <https://www.geeksforgeeks.org/c-sharp-methods/>
- [23] <http://zetcode.com/lang/csharp/methods/>
- [24] <https://www.codeproject.com/Articles/142292/Recursive-methods-in-Csharp>
- [25] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-parameters>
- [26] <http://jonskeet.uk/csharp/parameters.html>
- [27] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>
- [28] https://www.tutorialspoint.com/csharp/csharp_arrays.htm
- [29] <https://www.geeksforgeeks.org/c-sharp-arrays/>
- [30] https://www.tutorialspoint.com/csharp/csharp_classes.htm
- [31] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes>
- [32] https://www.tutorialspoint.com/csharp/csharp_struct.htm
- [33] <https://www.tutorialsteacher.com/csharp/csharp-struct>
- [34] https://www.tutorialspoint.com/compile_csharp_online.php
- [35] https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm
- [36] https://www.tutorialspoint.com/csharp/csharp_inheritance.htm
- [37] https://www.tutorialspoint.com/csharp/csharp_polymorphism.htm
- [38] https://www.tutorialspoint.com/csharp/csharp_regular_expressions.htm
- [39] https://www.tutorialspoint.com/csharp/csharp_exception_handling.htm
- [40] https://www.tutorialspoint.com/csharp/csharp_file_io.htm
- [41] <http://csharp.net-informations.com/file/csharp-file-tutorial.htm>
- [42] <https://www.tutorialsteacher.com/csharp/csharp-file>
- [43] <https://docs.microsoft.com/en-us/dotnet/api/system.io.file.appendtext?view=netframework-4.7.2>
- [44] <https://docs.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=netframework-4.7.2>
- [45] <https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter?view=netframework-4.7.2>
- [46] https://www.tutorialspoint.com/csharp/csharp_delegates.htm
- [47] <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/>
- [48] https://www.akadia.com/services/dotnet_delegates_and_events.html
- [49] https://www.tutorialspoint.com/csharp/csharp_multithreading.htm
- [50] <https://docs.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading>

[51] <https://o7planning.org/en/10553/csharp-multithreading-programming-tutorial>

[52] https://www.tutorialspoint.com/csharp/csharp_events.htm

[53] <https://www.tutorialsteacher.com/csharp/csharp-event>