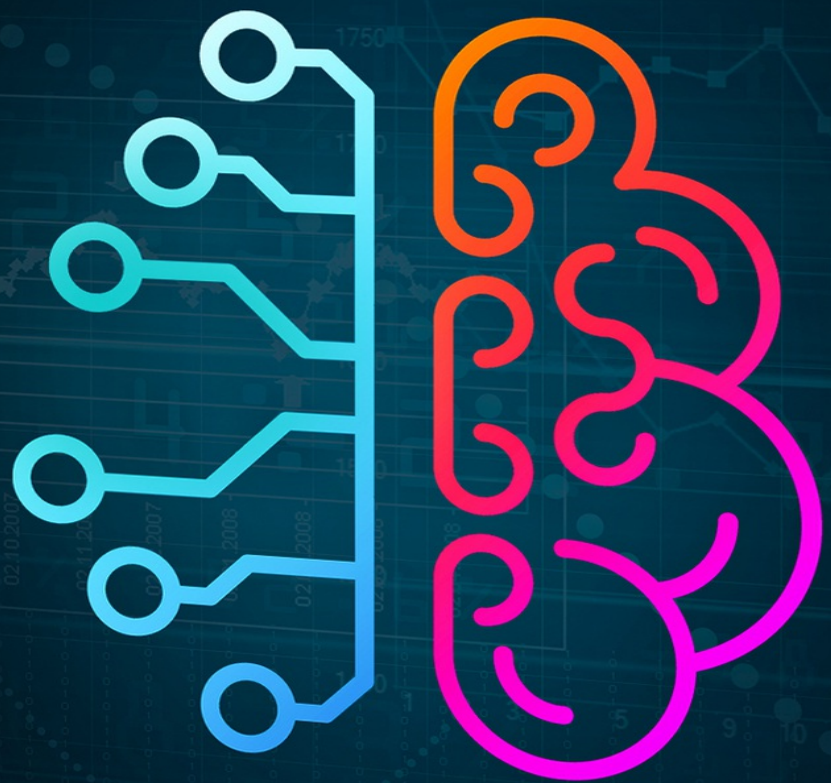
C#

THE ULTIMATE BEGINNERS GUIDE TO LEARN C# PROGRAMMING STEP-BY-STEP



MARK REED

C#

The Ultimate Beginners Guide to Learn C# Programming Step-by-Step

Mark Reed

Table of Contents

Introduction

If you are taking up programming, learning C# is one of the best decisions you will make. Knowledge of C# helps you lay the right foundation for your career in programming. With this knowledge, it is easier to advance into modern programming languages. You will also find it easier to learn some software development technologies, too. The principles, concepts, and knowledge learned in this book are useful even for programmers who do not wish to venture deeper into C# programming. You will realize that most of the concepts are cross-platform in that you can still apply and implement them in other languages.

An important point of note when reading this book is that the basic concepts and principles of writing computer programs have remained unchanged for more than a decade. Even as new programming languages enter the fray and technologies advance, the principles of programming have remained the same. Therefore, as a beginner, you should learn to think instinctively from the start. While learning about C# programming, you should also start thinking about algorithms and solving problems. Programming is essentially about making the world better, one line of code at a time.

This book is ideal not just for beginners, but also for anyone who has already ventured into programming, but is looking to go further. It is written in a manner such that you will start from scratch and grow into C#, mastering the fundamentals along the way. While it might not turn you into a software developer or software engineer overnight, it will help you lay the foundation upon which you can scale up your knowledge in programming and technology, which you can then use to map the way forward for your career.

For readers who can already write some simple lines of code, programs, or even if you just started a programming course in school, do not assume anything. This is the best attitude to get you through the world of programming. Never underestimate anything you come across, or its potential. While this book is written for beginners, you will learn important skills and concepts that many expert programmers do not have.

If you have some programming experience, you can also read this book in-depth and remind yourself of the concepts you have learned. This will help you advance into object-oriented programming and working with algorithms.

Computer science and knowledge in other information technologies is not mandatory prior to reading this book, but will help you comprehend the content better.

One of the most important skills you need in programming is the desire to learn. This is what makes the difference between successful and failed programmers. There are many lawyers, engineers, and experts in other fields who have since advanced into programming because of their desire. You will encounter several challenges in your programming career, and without the desire, you might lack the motivation to push through them. This also explains why these days you come across many good programmers and developers in the industry without a degree in computer science. Basic computer literacy is, however, a must have.

Using variables, working with data structures, organizing logical statements, writing conditional statements, loops, and arrays are some of the fundamental programming skills you will learn in this book. This knowledge underpins other complex concepts you will learn later on in your programming career, such as exception handling and string processing.

C# is your gateway to object-oriented programming, which is a prerequisite for modern programming and software development. Over time you will go on to write high-quality programs and write code to solve complex problems in the real world. This is the ultimate call for any programmer: algorithms and solving problems.

Learning C#, you will also need to familiarize yourself with Microsoft's .NET platform. C# is a Microsoft project which has since been used as a platform for most of the successful Microsoft projects we are accustomed to, including websites, desktop applications, web applications, and office and mobile applications. As a high level language, C# features in the same league as the likes of C++ and Java.

Over the years, C# has grown to become one of the leading languages preferred by programmers and developers all over the world. This is partly due to its versatility in the development ecosystem, and also because of Microsoft's backing as their go-to platform for developing and program execution. This means that if you ever plan to work in any of the companies that are backed by Microsoft, learning C# will give you a competitive edge over your peers. Considering Microsoft's position in the world of technology,

you can also take pride knowing that C# will continue to receive backing from one of the powerhouses in the industry for years to come.

Compared to C and C++, C# is one of the easiest modern programming languages you can learn. Millions of programmers all over the world code in C#. As an object-oriented programming language, you will be learning how to work with real world issues. Check any job listing website and you will find C# and .NET among the most requested skills by clients and employers, rivaling the likes of Java and PHP.

As you venture into programming, remember that knowing how to program is more important than the language you use. Whatever technology or language you need, you will master it faster if you know how to program, which means mastery of the basics. You will also realize that programming is essentially built around specific principles that evolve gradually with time. It is such principles that will make up the core of this book.

When reading this book, prepare yourself for lots of practice. You will never learn programming by reading alone. You must apply the knowledge learned to sharpen your skills. The more programming challenges you attempt, the faster you will learn and acquaint yourself. Write code all the time. There is no better way to learn C# or any other programming language.

Chapter 1:
C# An Overview

From the onset, programming is about working with computers at different levels. You are getting into a world of issuing commands and orders that your computer will execute through compilers. In the programming world, this is referred to as issuing instructions. Programming, therefore, is an act of organizing assignments issued to a computer through instruction sequences. This is where algorithms come in. At a later stage in your programming career, you will delve deeper into algorithms. An algorithm is simply a sequence of steps the computer follows to accomplish an assignment.

There are different programming languages in the market today, which are responsible for the programs we use on our computers, smartphones, and other devices. Everything you use online is written in some programming language. Programming languages are written to control computers at different levels. We have languages that are written to instruct computers at the lowest levels. An example of these is an assembler. We also have languages that are written to interact with computers at system level. Such languages instruct the computer through the operating system running. A good example is C. Beyond this we have programming languages that are used to interact with computers through programs and applications. These are the languages in which the programs are written. Such languages are known as high level languages, and in this category we have the likes of C#, PHP, Python, C++, Ruby, Java, and Visual Basic among many others.

Programming in a high level language like C# involves having access to, and controlling, most computer services, either through the operating system or by directly assigning computing resources to the services. Before we jump into C# programming, this overview will give you a glimpse of different aspects of software development, helping you understand the programming process better.

Any programmer you come across might tell you how time-conscious and complex their assignments are. There are times when you have to work with other programmers from different departments, or even collaborate online with programmers in different locations. To meet the collaboration needs, programmers use different practices and methods to make their work easier.

In any of the methods applicable, you will go through the following steps:

Step 1: Understanding the assignment and gathering all the information and resources necessary

At the onset, you only have an idea of the project you are about to write. The idea includes a list of instructions, requirements, or the kind of interaction expected between the computer and the user. This information is properly defined. At this point in time, there is no programming expected of you. It is about identifying and clearly defining the problem at hand.

Step 2: Preparation and planning the structure and design of your program

Having understood the requirements, your next step is to prepare and plan the program design. You come up with a technical plan that will guide the implementation of your project. In this case, the technologies, platform, and architecture of the program are designed. This is the first point in the development process where your creativity comes in.

In this stage, some of the decisions you make will include the type of application, for example a client-server, desktop application, or web application. Today a lot of clients seek mobile applications to bring their products closer to the target audience.

You also have to think about the development architecture, whether you plan on using single, double, or multiple layer architecture and so on. You will also think about the programming language within which you will write the program, including the likes of C#, Python, Ruby, or Java.

Having settled on the programming language, you also have to think about the technologies and development frameworks. In terms of technologies, you will be looking at platforms like .NET, a database server such as MySQL or Oracle, and user interface technologies like ASP.NET and Flash. There are lots of other technologies you can consider for your project, depending on your implementation process and the parts of the software system you wish to build.

On the technical front, you will also have to consider the development framework for your project. Say you are writing a program in Python; you might want to consider Django, or Rails if you are programming in Ruby.

Away from the technical aspects of the project, think about the composition

of your development team. How many developers are part of the project, and how confident are you that their skills will suit the project development requirements?

Step 3: The implementation stage (writing the program)

This is the programming stage. You write the program in line with the architecture, design, and client instructions. You will write the source code for the program. Most of this book is written with a view of helping you master this important stage in creating your project.

Step 4: Product testing and trials

Testing and trials is an important stage once you have written the program. The idea here is to ascertain whether all the program requirements are met accordingly. While you can test your program manually, it is always advisable to automate tests. Automated tests are small programs written to make the trial process seamless and efficient.

Quality assurance engineers often work with programmers in this stage to identify and fix bugs in the program. Note that in this stage, the objective is to identify defects in your code, and for the most part, you will not write new code.

Step 5: Deployment

After passing the testing and trials stage, the program moves to deployment. In this stage, the product goes into exploitation. The complexity of the program and number of people it serves determine how fast this process will be, and the costs involved. In most cases, you will create a small program known as an installer. Installers ensure easy and fast installation of the program.

Once the program is deployed successfully, the next step is to train users on how to use it effectively. This is where deployment experts come in. Deployment experts include system administrators, system engineers, and database administrators. While you don’t need to write new code at this stage, you can tweak and configure the existing code accordingly to ensure it meets the client requirements for successful deployment.

Step 6: Maintenance and support

It is prudent to expect errors and problems at deployment. Such problems arise from different factors, especially at configuration and software errors.

More often, problems arise because user requirements change, making it almost impossible for the program to meet the client needs because as currently constituted, the program can no longer perform the tasks for which it was created.

It does not matter how good the project is, maintenance and support are needed throughout the life of the project. The nature of support often depends on the kind and extent of changes made to the project, and the number of people involved in the maintenance and support process.

The development cycle of every program involves the six steps above. Beyond that, you have the documentation stage which basically sums up all the stages above. Documentation is important in that it connects all the steps, and is useful in development and program support. This stage is performed by developers and other experts involved in building the program.

At this point, you can already see that there is more to programming than writing code. Let's move on and write our first C# program. From here, we can delve deeper and learn more along the way. A typical C# program looks like this:

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

This is a simple instruction to print " Hello World " at output. It is still too early in the development cycle to execute the program, but we will use this excerpt to help you understand the program structure.

From the instruction above, you can deduce the following three important logical parts of the program:

- **Class definition Hello World;**

This is the first line of the program, defining the class Hello World . The simplest class definition will always include the keyword *class*, followed by

the name of the class. The class in our example is Hello World , whose contents are enclosed in a block of code within curly brackets {}.

- **Method definition Main();**

The third line of this program defines the method with the name Main() . This is where the program begins. All programs written in C# must start from this method, followed by the signature (title) below:

static void Main(string[] args)

This is how we declare methods in C#. Methods should be static and void, and include Main and in the list of parameters, have only one parameter of the type array . In the example above, we have the parameter args , though we can omit it since it is not mandatory. We can, therefore, simplify the code above as follows:

static void Main()

Take note of the instructions above because if any of them are not met, your program will never execute. The program might still compile, but since the starting point is not properly defined, it will not execute.

- **Method contents Main().**

When writing programs in C#, you include the content of every method after the signature, enclosed within the curly brackets. As you can see above, we have used the system object System.Console and the method WriteLine() to print the message ”Hello World” . In this method, you can write any manner of expressions, and they will be compiled and executed in the sequence order you assign to them. We will discuss more of these concepts as we go deeper into the book.

Going forward, you must also keep in mind that C# is a case-sensitive language. Therefore, when writing code in C#, be careful with your upper and lower case code. In C#, static and Static do not mean the same thing. This rule applies to all aspects of your program, including class names, variable names, and keywords.

This brings us to formatting rules. There are a few simple guidelines you should always remember when formatting code in C#:

1. All method names must begin with a capital letter

2. All variable names must begin with a lowercase letter
3. All class names must begin with a capital letter
4. Methods must be indented within class definitions using the Tab character on your keyboard.
5. The method contents must be indented within the method definition
6. You must place the opening curly bracket { on its own line, and under the class or method to which it refers.
7. You must place the closing curly bracket } on its own line, and directly under the relevant opening bracket, using the same indentation.

Code indentation, as used in the point above is another important rule you must adhere to when writing code in C#. When writing some new code inside another set of code, you must indent it to the right using a single Tab. For example, if you are nesting a method within a class, the method indents further to the right from the class.

When coding in C#, each program must have at least one class definition. Every class is further defined in a different file, with the file name corresponding to the class name. They must also have a **.cs** extension. From the example we used above, we have the class HelloWorld , and for that reason, the file is saved in a file named HelloWorld.cs .

C# and The .NET Platform

Microsoft released the first version of C# under the .NET platform. This platform was built to enhance the software development platform for the Windows ecosystem through managed code and the virtual machine concept. C# is a high-level object-oriented, general purpose programming language. While the C# syntax is similar to C++ and C, most of their features are not supported in C#. The idea behind this is to make programming in C# cleaner and easier.

Programs written in C# are compiled through the C# compiler, in the process

creating assemblies. Assemblies are files that share similar names with the program, but have a different extension. You can identify assemblies with the extensions **.dll** or **.exe** . For example, if we create a Hello World program and compile HelloWorld.cs , you will see a file name HelloWorld.exe or HelloWorld.dll . There are lots of other files that can be created in the process.

Why is the .NET framework important? You get an error message when you try to run C# code on a computer without the .NET framework installed. Other than that, one of the benefits of the .NET framework is its automatic memory management. This is useful to programmers because without it, you would have to manually allocate memory to objects, and determine the appropriate time to release the memory from objects that no longer require such resources. It is safe to say that with the automatic memory management feature, .NET helps to improve the quality of programs and at the same time, enhance your productivity.

The .NET framework has a unique component known as the garbage collector. This is an automated memory cleaning system that checks when memory apportioned to different variables is not in use, and releases it, making it available for other objects that might need that memory.

Another benefit of the .NET framework is that you can easily exchange code with other programmers, as long as they code in any of the .NET supported languages. Therefore, given that you are coding in C#, you can use code written by other programmers who write in F#, Managed C++ or VB.NET. All this is possible because all .NET languages use similar infrastructure for execution, data types, and assemblies.

The .NET framework runs effortlessly because of the common language runtime (CLR). This is the environment within which managed code is executed in C#. It is because of the CLR that you can execute .NET programs on different operating systems and hardware systems. The CLR is essentially a virtual machine. It therefore supports memory access, instructions, registries and input-output operations. Through the CLR, you are able to execute .NET programs using the competencies of the operating system and processor.

What we refer to as the .NET platform, therefore, is a combination of the CLR, C# programming language, and a host of libraries and auxiliary

instruments.

Chapter 2:
Types and Variables

When coding in C#, data types and variables are important concepts you must understand. Understanding their characteristics makes it easier to discern the appropriate instances where you can use them without your program yielding unsavory results. When it comes to variables, you need to know the unique features and how to declare each variable correctly in your code. Other than that, you must also know how to assign values to variables.

In programming, you will often use random values whose properties can change while in execution. You can, for example, create a program that measures the body mass index (BMI). Obviously, since the calculations depend on the individual's mass and height, you can expect different entries from each user. Therefore, when writing the program, you do not know the values that each user will introduce at input, so the best option is to write a program that can process all possible values users can enter.

Each time someone enters either the mass or height values to determine their BMI, they are temporarily stored in the computer's RAM. It is such values that change at execution depending on the user's input that we collectively refer to as variables.

Data Types

Data types refer to a range of values that share similar characteristics. For example, the ***byte*** data type refers to integers within the range of 0 and 255. Data types are identified according to their names, size (memory allocation) and default values. In C#, the basic data types are also known as primitive data types because they are built-in. They are hard-coded into the C# language at the fundamental levels. Let's look at the primitive data types below:

Integer Types

These types refer to integer values and are as follows:

- **sbyte**

These are signed 8-bit integers, meaning that they can only handle up to 256

values (2^8), including negative and positive values. From the default value 0, the minimum and maximum values stored by sbyte can be expressed as follows:

SByte.MinValue = -128 (-2^7)

SByte.MaxValue = 127 (2^7-1)

- **byte**

Unlike sbyte, these are unsigned 8-bit integers. They can also handle up to 256 integer values. However, bytes cannot be negative values. The default byte value is 0, and the minimum and maximum values stored in bytes can be expressed as follows:

Byte.MinValue = 0

Byte.MaxValue = 255 (2^8-1)

- **short**

These are signed 16-bit integer values. The default value for the short type is 0, while the minimum and maximum values stored can be expressed as follows:

Int16.MinValue = -32768 (-2^{15})

Int16.MaxValue = 32767 (2^{15}-1)

- **ushort**

These are unsigned 16-bit integer values. The default value for ushort is 0, while the minimum and maximum values stored can be expressed as follows:

UInt16.MinValue = 0

UInt16.MaxValue = 65535 (2^{16}-1)

- **int**

These are 32-bit signed integer values. By now you can see that as the bits grow, the number of possible values that can be stored in the data type also grows. The default value for int is 0, while the minimum and maximum values stored can be expressed as follows:

Int32.MinValue = -2,147,483,648 (-2^{31})

Int32.MaxValue = 2,147,483,647 (2^{31}-1)

Of all the data types, int is the most commonly used in most programming languages. This is because it naturally fits a 32-bit microprocessor, and the fact that it is large enough to handle most of the calculations performed in normal life.

- **uint**

These are 32-bit unsigned integer values. The default value for uint is **0u**, which can also be written as **0U**. This is one of the few exceptions in C# programming where upper or lower case mean the same thing. When using this data type, you must be careful to include the letter U, otherwise the compiler will interpret it as an int data type. The minimum and maximum values stored can be expressed as follows:

UInt32.MinValue = 0

UInt32.MaxValue = 4,294,967,295 (2^{32}-1)

- **long**

These are 64-bit signed integer types whose default value is **0l**, which can also be written as **0L**. When using this data type, it is often advisable to use the upper case L because the alternative can be misconstrued for 1. Another common mistake people make is to forget the L, in which case the compiler interprets the values as int, yet we know the number of values held within cannot match.

The minimum and maximum values stored can be expressed as follows:

LInt64.MinValue = -9,223,372,036,854,775,808 (-2^{63})

LInt64.MaxValue = 9,223,372,036,854,775,807 (2^{63}-1)

- **ulong**

This is by far the largest integer type in C# programming. The 64-bit unsigned type has the default value as **0U** or **0u**. Without the U, compilers will interpret the data type as long values, so you must also be careful with that. The minimum and maximum values stored can be expressed as follows:

UInt64.MinValue = 0

UInt64.MaxValue = 18,446,744,073,709,551,615 (2^{64}-1)

Let’s use an example to explain how you can declare several variables of the integer data types mentioned above.

```
// To declare loan repayment variables

byte years = 20;

ushort weeks = 1042;

uint days = 7300;

ulong hours = 175200;

// Print output to console

Console.WriteLine(years + " years are " + weeks + " weeks, or " + days + " days, or " + hours + " hours.");

// output:

// 20 years are 1042 weeks, or 7300 days, or 175200 hours.

ulong maxIntValue = UInt64.MaxValue;

Console.WriteLine(maxIntValue); // 18446744073709551615
```

Real Floating Point Types

These data types refer to the real numbers you use every day. They are real numbers you use in mathematics all the time. Such data types are identified by floating points as per the IEEE 754 standard, and can either be a float or a double.

- **Real type floats**

This is a 32-bit float, also referred to as a single precision real number. The default value for this data type is **0.0f**, or **0.0F**. The ‘f’ character at the end of the value denotes the float type. This is important because all real numbers are often considered doubles by default.

When using this data type, you can express accuracy up to seven decimal points, rounding off all values accordingly. For example, the value 3.323588725723 when stored as a float is rounded off to 3.3235887.

We also have real type float values that are not necessarily real numbers, but they still fit mathematical concepts. These include negative infinity, positive

infinity, and uncertainty (**Single.NaN**), which occurs when you perform an invalid operation using real numbers (for example when you try to calculate the square root of negative values).

- **Real type doubles**

These data types are also known as double precision real numbers, and are 64-bit data types whose default value is expressed as **0.0d** or **0.0D** . Unlike other data types, you can omit the 'd' because in C#, all real numbers are considered doubles by default.

When using real type doubles, you can express values up to 15 or 16 decimal points. Bearing this in mind, the minimum and maximum real type double values can be expressed as follows:

Double.MinValue = -1.79769e+308

Double.MaxValue = 1.79769e+308

We also have the Double.Epsilon = 4.94066e-324 , which is the closest positive double type to zero. As we saw in real type floats, we can also have unique values in real type doubles, like Double.Nan, Double.NegativeInfinity, and Double.PositiveInfinity . Let's use an example below to show the difference between these two, and the accuracy levels:

```
// To declare random variables

float floatPI = 4.248273754726743285f;

double doublePI = 4.24827375472674328785;

// Print output to console

Console.WriteLine("Float PI is: " + floatPI);

Console.WriteLine("Double PI is: " + doublePI);

// Console output:

// Float PI is: 4.248274

// Double PI is: 4.24827375472674
```

As you can see above, we declared a random float value to the 7th digit, but when declaring the same value as a double, we declare it to the 15th digit. This is proof that the real type doubles allow more precision than floats, so if you are writing a program that demands specificity beyond the decimal point,

it makes sense to use real type doubles.

Real Decimal Precision Types

The decimal floating point method is commonly used in C#, where the decimal numbering system is used instead of the binary system. Using the decimal precision points, real numbers are expressed as 128-bit decimal types, allowing up to 28 or 29 decimal places. The default values using this system are **0m** or **0M** . The ‘m’ in the number is an explicit indicator for decimal types. This is important because by default, all values are considered of the double type as discussed earlier. Let’s look at an example below:

```
decimal declaration = 6.386284284624928472 64m;

Console.WriteLine(declaration); // 6.38628428462492847264
```

In the example above, the number used has not been rounded because it falls within a 21-digit precision, which naturally fits the decimal without rounding off. The decimal type is ideal for calculations that require a high level of accuracy, for example in financial calculations.

Boolean Types

This data type is declared using the keyword **bool** . Boolean types only return two possible values, true or false, false being the default Boolean value. Boolean types are ideal when storing calculations involving logical expressions. Let’s explain this with an example below:

```
int x = 5;

int y = 10;

// Which one is greater?

bool greaterXY = (x > y);

// Is 'x' equal to 5?

bool equalX5 = (x == 5);

// Print the results on the console

if (greaterXY)

{

Console.WriteLine("X > Y");
```

```
}
else
{
Console.WriteLine("X <= Y");
}
Console.WriteLine("greaterXY = " + greaterXY);
Console.WriteLine("equalX5 = " + equalX5);

// Console output:
// X <= Y
// greaterXY = False
// equalX5 = True
```

In the example above, we declared two variables of the type **int**, and by comparison, we assign the resulting value using the **bool** type, and variable **greaterXY** . In the same manner, we also declare for the variable **equalX5** . If the variable **greaterXY** returns **true** , then the console will print **X > Y** . If that is not the case, it prints **X <= Y** .

Character Types

These are single 16-bit characters. They are declared using the keyword **char** . The smallest possible char variable is 0, while the largest value is 65535. This data type represents values in the form of letters and other characters, and are always enclosed within an apostrophe.

In the next example, we will declare a variable of the type char, initializing it with the value ‘x’, then ‘y’ and finally ‘X’, printing the outcome to the console.

```
// Declare a variable
char ch = 'x';
// Print the results on the Console.WriteLine(
"The code of '" + ch + "'is: " + (int)ch);
ch = 'y';
```

```
Console.WriteLine(
"The code of '" + ch + "'is: " + (int)ch);
ch = 'X';
Console.WriteLine(
"The code of '" + ch + "'is: " + (int)ch);

// Console output:
// The code of 'x' is: 76
// The code of 'y' is: 77
// The code of 'X' is: 43
```

String Types

String characters are declared in C# using the keyword **string**, and their default value is **null**. They are always expressed within quotation marks, and can be used to process different text functions. For example, two strings can be joined together, a process known as concatenation, or split using a separator, and so on.

In the example below, we declare variables of different string types, and print the outcome to the console:

```
// Declare variables
string firstName = "Mazda";
string lastName = "Demio";
string fullName = firstName + " " + lastName;
// Print output to the console
Console.WriteLine("Welcome to, " + firstName + "!");
Console.WriteLine("You are currently viewing a " + fullName + ".");
// Console output:
// Welcome to, Mazda!
// You are currently viewing a Mazda Demio.
```

Object Types

These are special data types which house all the data types within the .NET framework. They are declared using the keyword **object** , and can assume values from any of the data types we have discussed above.

Object types are basically reference types, used to identify memory areas where actual data types are stored. In the example below, we will declare different variables of the **object** type, and print the outcome to the console:

```
// Declare variables

object containerX = 10;

object containerY = "Ten";

// Print the output on the console

Console.WriteLine("The value of containerX is: " + containerX);

Console.WriteLine("The value of containerY is: " + containerY);

// Console output:

// The value of containerX is: 10

// The value of containerY is: Ten.
```

From the example above, you can see that the **object** type can be used to store any other data types. Therefore, **object** types are universal data containers.

Variables, Value and Reference Types

A variable in C# refers to a container with some information. The information in question is not static, so its value can change from time to time, with respect to information storage, and retrieving the stored information. You can also use variables to modify the information.

Any variable is identifiable and distinguished from others by the following features:

1. The name of the variable (its unique identifier), such as color
2. The type of variable (the kind of information held in the variable), such as string
3. The value (information stored in the variable), such as 3.124

Since variables are a specific area in the memory, they store data values in a specific memory area only accessible by the name of the variable. Primitive data types like **bool, char** and **numbers** are value types because their value is stored within the program. On the other hand, reference data types like **arrays** , **objects** , and **strings** point to the dynamic memory where their value is stored. Such data types can be released or allocated, meaning that their size is not fixed as is the case with the primitive data types.

When naming variables, there are specific rules that you must adhere to. This is important to avoid unnecessary errors when you pass some information to the compiler. While you have freedom of naming choice with variables, they must follow the following rules in C#:

1. Variables can only be named using the characters **A-Z** , **a-z** , digits **0-9** , and the character' **_** '.
2. You cannot start a variable name with a digit
3. While you are free to name variables as you wish, the names should never be similar to the innate C# keywords. For example, your variables cannot have any of the following names: **null, this, int, default, object, char, base** , or any of the other keywords you will come across in C#.

That being said, the proper way of naming a variable can be any of the following:

1. **name**
2. **name_name**
3. **_name37**

If you write the variable name in any manner that contravenes the instructions above, you will get a compilation error.

Literals

Earlier on we mentioned primitive types, as unique data types that come prebuilt into C#. The values of such data types are specified within the program's source code. Here are some examples to explain this:

```
bool result = false;

char capitalD = 'D';

byte b = 600;

short s = 15000;

int i = 820000;
```

In the example above, the literals are **false** , **D** , **600** , **15000** , and **820000** . Their values are set within the program's code. The following are literal types you will use in C#:

1. **String**

These literals are used to represent the **string** data type. A string can be preceded by the character **@** , which denotes a verbatim string (quoted string).

2. **Character**

These literals are single characters which are enclosed within an apostrophe. They are used to represent **char** value types.

3. **Integer**

These are digit sequences, including the signs **+** and **-** , used as prefixes or suffixes.

4. **Real**

These are digit sequences, the signs **+** and **-** , suffixes and the decimal point. They are used with the decimal, double, and float value types.

5. **Boolean**

They return true or false, for example, **bool result = false;**

Chapter 3:
Operators and Expressions

Having learned about the different data types used in C#, the next step is to understand operators, expressions, and more importantly, when to use them on data. Primarily, you need to know the priorities assigned to different operators, their types, and arguments within which they can be used in programming. Another important aspect we will look at later on is how to convert different data types where necessary.

Operators

In programming, operators are used in every programming language to perform specific actions on data. The role of operators is to enable you to process objects and primitive data types. Operators are applied to one or more operands, returning a result.

Other than their indigenous roles, operators can also be classified based on the number of arguments that you can pass through them. According to this, we have the following operator types:

- Unary operators - they take one operand
- Binary operators - they take two operands
- Ternary operators - they take three operands

In C# programming, we calculate operations from left to right, a system referred to as left-associative. The only exception to this rule is for the assignment operators. Assignment and conditional operators in C# are right-associative. It is also important to note that unary operators are not associative, meaning that you can calculate them from either side.

Another point you will learn about operators is that they perform different tasks depending on the data types to which they are applied. Take the operator `+`, for example. When used on strings, it joins the relevant strings together to return a new string (concatenation). On the other hand, if you use the operator on numeric data types, it performs an arithmetic addition, yielding a new result. Let’s explain this in an example below:

```
int x = 5 + 10;
```

```
Console.WriteLine(x); // 15
string firstName = "Mazda";
string lastName = "Demio";
// Do not forget the space between them
string fullName = firstName + " " + lastName;
Console.WriteLine(fullName); // Mazda Demio
```

From the example above, you can see how the operator performs a different role depending on the operation in question.

Another important point you will need to learn about operators is the precedence. Operator precedence is purely about priorities. This refers to the order in which operations are carried out, depending on the operators applicable. For example, when carrying out arithmetic operations, the multiplication operator takes precedence over addition. Building on this information, operators that have a higher precedence are always calculated ahead of those with a lower precedence. While you're at it, you can use the operator `()` to change the precedence of an operation just as you do in math.

If you write an expression that must use more than one operator, or is naturally complex, it is advisable to introduce parentheses into the expression. This makes it easier to understand, and reduces difficulties for anyone who might use the code later. Let's look at an example below:

```
// The expression below is ambiguous
a + b / 25
// You can write it better
a + (b / 25)
```

Always remember that parentheses are useful when you need to change the operator precedence. Let's have a look at the different types of operators below:

- **Arithmetic Operators**

In C# programming, arithmetic operators are similar to those you use in math. Arithmetic operators are used to perform addition, subtraction, division, and multiplication operations on integer values to return a numeric output.

The division operator does not have the same effect on real numbers and integers as other arithmetic operators. For example, when dividing two integers, the result must be an integer. If the result has a decimal point, you ignore the decimal. It is also by convention that you do not round off the value to the nearest whole integer. This is referred to as integer division, and you can see an example of this below:

9 / 4 = 2

On the same note, you cannot perform an integer division by zero. This would return the runtime exception **DivideByZeroException** . You can, however, divide two real numbers, a process known as real division. In this case, the result can be an integer or a real number with a fraction. Using the example above, we have the following:

9 / 4 = 2.25

In this concept, you can divide real numbers by zero, returning any of the following infinity values, depending on the operand used: **NaN** (Invalid value), -Infinity (- ∞), or Infinity (∞).

You will also come across the increment operator (++), that adds one unit to the variable value, and the decrement operator (--) that subtracts one unit from the variable value. The effect of these operators depends on their placement in the expression. When used as prefixes to the variable, you calculate the new value, then return the result. However, when used after the variable, you return the original value of the operand first, then perform the decrement or increment.

- **Logical Operators**

These operators take and return Boolean values and results (**True** or **False**). The basic operators here are **logical negation** (**!**), **OR** (||), **exclusive OR** (^), and **AND** (**&&**). In the table below, we look at how the operators can be used and the logical results:

a	**b**	**!a**	**a && b**	**a || b**	**a ^ b**
true	true	fasle	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true

false	false	true	false	false	false

From the table above, we can see that the logical AND, &&, can only return a true when both variables are true, while the logical OR, ||, can only return a true when either of the operands are true.

Building on that, you can change the argument value using the logical negation operator, !. If, for example, your operand has a true value followed by the negation operator, the operation returns a false.

The exclusive OR, ^ can only return a true if one of the operands is true. If the operands do not have the same values, the exclusive OR returns a true output, or a false if the values are similar. Let's explain these statements in the example below:

```
bool x = true;
bool y = false;
Console.WriteLine(x && y);              // False
Console.WriteLine(x || y);              // True
Console.WriteLine(!y);                  // True
Console.WriteLine(y || true);           // True
Console.WriteLine((3 > 5) ^ (x == y))  // False
```

- **Concatenation Operators**

In programming, the operator + can be used to join different strings, returning a new string as the result. Assuming that at least one of the arguments within the expression is of the string type, while other operands in the expression are of different data types other than string, the operator automatically converts them into the string type, hence string concatenation. Let's explain this using an example below:

```
string csharp = "C#";
string dotnet = ".NET";
string csharpDotNet = csharp + dotnet;
Console.WriteLine(csharpDotNet); // C#.NET
```

```
string csharpDotNet4 = csharpDotNet + " " + 5;
Console.WriteLine(csharpDotNet4); // C#.NET 5
```

In the example above, two string variables are declared and assigned values, which are eventually converted to string types and printed in the last row.

- **Bitwise Operators**

These operators are applied to numeric binary representatives. In computer language, most data is represented in binary language, hence the binary numeral system. For example, in the binary numeral system, Hello World is represented as follows:

01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100

Bitwise operators and logical operators are almost alike. They both perform the same roles, the only difference is that they act on different data types. While logical operators return **True** or **False** for Boolean values, Bitwise operators are applied to numeral values. We can represent this in the table below:

a	**b**	**~a**	**a & b**	**a | b**	**a ^ b**
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

From the table above, you can see the similarity between bitwise operators and logical operators. You will also come across the operators bit shift right `>>` and bit shift left `<<`. You can use these operators with numeral values to move bits to whichever side you want them. Note that when you use them, any bits outside the number are eliminated, and replaced with a zero.

- **Comparison Operators**

These operators are for operand comparisons. You can understand the term from basic English. In C# programming, you will come across the following comparison operators:

> greater than

< less than

>= greater than or equal to

<= less than or equal to

== equality

!= difference

Note that when using comparison operators in C#, the operators act on Binary operands, but return Boolean values as the outcome. They are, however, lower in priority to arithmetic operators. Let's look at an example below:

```
int a = 12, y = 6;

Console.WriteLine("a > b : " + (a > b)); // True

Console.WriteLine("a < b : " + (a < b)); // False

Console.WriteLine("a >= b : " + (a >= b)); // True

Console.WriteLine("a <= b : " + (a <= b)); // False

Console.WriteLine("a == b : " + (a == b)); // False

Console.WriteLine("a != b : " + (a != b)); // True
```

From the example above, you can see different forms of comparison between the variables a and b, and the Boolean outcomes. Note that while the comparison operators are lower in priority to arithmetic operators, they are higher than assignment operators which we will look at shortly.

- **Assignment Operators**

These operators are used to assign values to variables, using the character `=`. Here is an example:

```
int a = 10;

string helloWorld = "Hello World.";

int b = a;
```

You can also use the assignment operator more than once in one expression, a process known as cascading. Let's see an example of this below:

```
int a, b, c;
```

a = b = c = 55;

Be careful not to confuse the assignment operator = and the comparison operator ==. This is one of the most common errors people make when writing code in C#.

Instead of writing voluminous code, you can reduce the work by using compound assignment operators. This is possible by shortening two operations using an operator. The syntax for compound operators is as follows:

operand1 operator = operand2;

The expression above can be expanded as follows:

operand1 = operand1 operator operand2;

In an example, this is as follows:

int a = 3;

int b = 5;

a *= b; // Same as a = a * b;

Console.WriteLine(a); // 15

You will come across the following compound operators in C#:

+=

-=

*=

/=

%=

- **Conditional Operators**

This operator applies an expression's Boolean values to select one or more expressions whose result is needed. It is a ternary operator because it is applied to three operands. In application, the syntax for this expression would be as follows:

operand1 ? operand2 : operand3

The compiler is instructed to calculate the value of the first argument. If the first argument is true, it returns the result. However, if the first argument is

false, the compiler calculates the value of the third argument, and sends back its result.

Type Conversion

Operators are written for arguments of the same data type. As you have seen so far, there are several data types in C#, and from time to time you will have to choose one that is most appropriate for your program. If you write a program whose variables are of different data types, you must convert them to one data type (type conversion). Writing an expression with different types can result in a compilation error.

While type conversion is a brilliant idea, there are only a few data types that can be converted. For ease of understanding, type conversions fall into the following three categories:

- **Implicit Type Conversion**

This is a hidden type conversion without the risk of data loss. An example is when converting a low range data type to a high range data type, like an **int** to a **long** . This type of conversion is implicit because you do not need to use an operator. The computer automatically converts types when you assign lower range values to larger range variables. This also happens if you write an expression with different data types within different ranges.

Let’s look at an example of this below:

```
int myInt = 7;

Console.WriteLine(myInt); // 7

long myLong = myInt;

Console.WriteLine(myLong); // 7

Console.WriteLine(myLong + myInt); // 14
```

In the example above, the compiler has converted the **int** value to a **long** value, printing the outcome to the console. In C# programming, you can perform the following implicit conversions:

byte to short , ushort , int , uint , long , ulong , float , double , decimal;

float to double;

int to long, float , double , decimal;

long to float , double , decimal;

uint to long, ulong , float , double , decimal;

ulong to float , double , decimal;

ushort to int , uint , long , ulong , float , double , decimal;

sbyte to short , int , long , float , double , decimal;

short to int , long , float , double , decimal;

Note that after conversion from a small range to a large range, the numerical value generally remains unchanged, hence no data loss at conversion. There are, however, a few exceptions to this rule, like conversion from an **int** to a **float** , or converting to a **64-bit double** from a **64-bit long** . This happens because of a difference in the number of bits used, and the possibility of rounding off the fraction parts of the data types.

- **Explicit Type Conversion**

This conversion type is recommended in situations where data loss is possible upon successful conversion. Let's say you need to convert data from an **integer** to a **floating point** ; you have to discard the fraction part of the data type, hence the need for an explicit conversion.

You might also experience data loss when converting from a large range to a lower range, for example, **long** to **int** . Let's look at an example of this below:

```
double myDouble = 7.1d;

Console.WriteLine(myDouble); // 7.1

long myLong = (long)myDouble;

Console.WriteLine(myLong); // 7

myDouble = 7e7d; // 7 * 10^7

Console.WriteLine(myDouble); // 50000000
```

Let's look at a different scenario below:

```
int myInt = (int)myDouble;

Console.WriteLine(myInt); // -2343576879

Console.WriteLine(int.MinValue); // -2343576879
```

In the example above, there is no change after conversion to **int** , because the value held in the variable **myDouble** is larger than the data range of **int** .

Given the difficulty in determining the variable value after conversion, it is advisable to use large data types to allow enough room for flexibility. At the same time, avoid, or be careful when converting to smaller data types.

- **String Conversion**

Where necessary, you can convert any data type to strings. String conversions happen automatically as long as you use the operator + for concatenation, alongside an argument that is not of the string type. If all these conditions are met, the compiler converts the argument and returns a string outcome. You can also achieve the same result using the method **ToString()** on the variables.

For example:

24.ToString() returns “24”

Using this information, let’s look at some dimensions of a rectangle below:

int x = 3;

int y = 6 ;

string SUMMARY: = “SUMMARY: = “ + (x + y);

Console.WriteLine(SUMMARY:);

Console.WriteLine(“Perimeter = “ + 2 * (x + y) + “. Area = “ + (x * y) + “.”);

From the arguments above, you will get the following output:

SUMMARY: = 9

Perimeter = 18

Area = 27

Note that if you need to change the operation priority, you must introduce brackets. Otherwise, arguments are executed from left to right by default.

Expressions

When you write a program, you are basically tasking it with calculating expressions. An expression loosely refers to a sequence of variables, literals,

and operators, which return a value. You must be careful when writing expressions because some of them usually contain embedded assignment operators. The outcome can either increase or reduce. Let’s look at an example to explain this:

```
int x = 19;

int y = ++x;

Console.WriteLine(x); // 20

Console.WriteLine(y); // 20
```

Make sure you consider the data type and the relationship between the operators when writing expressions. Ignoring this can lead to unpleasant outcomes. Many programmers assume that division by zero is impossible, because it is mathematically not doable. However, in C# programming, this concept will only be true with integer divisions. In other cases, fractional division by zero returns a **NaN (Infinity)** .

We mentioned this before, but we can highlight it again – the role of parentheses in programming. If you are in doubt about the priority of your operations, you can introduce parentheses for clarity.

Chapter 4:
Console Input and Output

The console in C# is a data input and output tool. It is the operating system window through which users interact with different programs in the operating system or other applications installed into the console. For this interaction, you will have text input from the standard input (keyboard) and the text display (computer screen). This is what we refer to as an input-output operation. Any console application through the operating system is connected through input and output devices. Generally, the default input and output devices are the keyboard and the screen, though you can also redirect either of the two to a unique file or device.

Modern user interfaces are more intuitive and convenient for users, making the console less desirable. However, there are many cases where you will still find the console coming in handy, especially for communication between users and the system. For example, if you need to write a simple computer program, you will have to use the console. This is because the console allows you to isolate the problem and focus on it, instead of the graphical representation of the solution to the user.

Another situation where the console will always come in handy is when you need to debug or test a subsection of code for a large program. You will simply isolate the section you need, address it, and solve the problem instead of having to go through the complexities of searching through the larger program.

Each operating system has a unique way of launching the console. Using Windows, for example, you can access the console in the following steps:

Start > All > Programs > Accessories > Command Prompt

The Windows console is also referred to as the **Command Prompt** or **Shell** . It is a command interpreter, a console-based program within the operating system through which you can access the system commands and any other programs you might need access to. There are two types of shells, depending on the kind of interface they provide your operating system. These are as follows:

- Command Line Interface (CLI) - This is the go-to console for

commands, for example, **bash** if you are using a Linux environment, or **cmd.exe** if you are using a Windows environment.

- Graphic User Interface (GUI) - This is the environment in which you can see the graphics, for example, Windows Explorer

In each of these instances, the core role of the shell is to run all the programs you run. However, you will also realize that you can do the same with most interpreters, especially those that are built in such a way that you can inspect and examine content within the files and directories.

Understanding the Console

For the purpose of this book, we will primarily use Windows as the programming environment. We will also look at some of the basic console commands built into the command prompt, which come in handy whenever you need to run a new program.

The command interpreter that runs in the Windows console was initially referred to as **MS-DOS Prompt** in the earlier Windows releases, though in modern versions it is known as **Command Prompt** . The following are some of the basic commands you can access using this interpreter:

dir - Shows all content in the current directory

cd <directory name> - Will change the current directory

mkdir <directory name> - Will create a new directory within the current directory

rmdir <directory name> - Will delete an existing directory

type <file name> - Will print content on the selected file

copy <src file> <destination file> - Will copy one file into the selected file

You will also come across **Standard I/O** , which refers to a system input-output mechanism that has been in use since the creation of the UNIX operating system module. It uses special peripheral input and output devices to process data. In this mode, you will notice a blinking cursor whenever the console is processing or waiting for the user to input a command for processing.

You will realize that most programming languages read and print information

from the console in the same manner. For this reason, the solutions returned are mostly under the standard input and standard output concept.

By default, every operating system defines its standard input-output mechanism for normal operation. When you start a program, it runs some code at initialization which allows it to automatically read your instructions from the standard input stream (**Console.In**) and print the outcome to the standard output stream (**Console.Out**). If the system detects an error, this is reported under (**Console.Error**).

We mentioned earlier that you input to the console through the keyboard. However, it is not the only input source. The console can receive input from many other sources, including barcode readers, microphones, and files. Below is an example of printing text to the console through a standard input-output operation:

Console.Out.WriteLine(“Welcome Home”);

When you execute the code above, you should have the following output

Welcome Home

This brings us to the **Console.Out** stream. With time, you will realize that there are different methods and properties used in **System.Console** not just to read and display text within the console, but also read and display the formatting correctly. The most important of these are as follows:

Console.Out

Console.In

Console.Error

Their role is to grant access to the basic streams you need when printing, reading on the console, and reporting errors identified. You can also replace these streams at runtime using the respective methods below:

Console.SetOut

Console.SetIn

Console.SetError

Console.WriteLine() vs Console.Write()

Earlier on we discussed different data types. It is easier to use different

console properties because, by default, they allow you to print any of the basic data types, primitive, numeric, and string. Let's explain this using an example below:

```
// Print String

Console.WriteLine ("Welcome Home");

// Print Int

Console.WriteLine (7);

// Print double

Console.WriteLine (2.635542798664338);
```

The outcome when you print this code will be as follows:

```
Welcome Home

7

2.635542798664338
```

Using **Console.WriteLine()** , you can print any kind of data type because, for each type you use, you will find a predetermined version for the method **WriteLine()** within the console class.

While **Write()** and **WriteLine()** might seem alike, **Write()** prints all the contents of the parentheses to the console but does not perform any execution. On the other hand, **WriteLine()** is a direct instruction to print what is on that line. Essentially, it will not only perform everything **Write()** does, it will also proceed to the next line. However, note that it will not print the new line, but instead, it inserts a moving command cursor where the new line should begin. Let's look at an explicit difference between these two:

```
Console.WriteLine("I am");

Console.Write("missing");

Console.Write("Home!");
```

From the code snippet above, you will have the following feedback at output:

```
I am

missing Home!
```

Now if you pay attention, you will realize that the console prints the output

on two separate lines yet the original code was written on three lines. This is because we use **WriteLine()** in the first line, which prints and jumps to the next line by default. The next two lines of the code use the **Write()** method that prints but does not jump to the next line, hence " **missing** " and " **Home!** " end up on the same line.

String Concatenation

By default, you cannot use operators over string objects in C#. The only instance where this is allowed is when using the **(+)** operation to concatenate two strings and return a new string as a result. Here is an example to explain this:

```
string students = "thirty five";
string text = "There are" + students + "in the class.";
Console.WriteLine(text);
```

When you execute the code, you get the following output:

```
There are thirty five students in the class.
```

The example above is a simple one. What if we encounter something slightly complex, with more than one type? So far, we have only worked with **WriteLine()** versions of one type. Assuming that you have to print different types at once using **WriteLine()** , you don't have to use different versions of the **WriteLine()** method. Note that in the next example, the number of students is of the integer type:

```
int students = "thirty five";
string text = "There are" + students + "in the class.";
Console.WriteLine(text);
```

We will still have the same output as shown below:

```
There are thirty five students in the class.
```

You can see the concatenation in the second line of the string " **There are** " and the integer type " **students** ". We are able to combine these two types because the result of concatenation between a string and any other type will always result in a string.

That being said, you have to be keen when performing string concatenation,

lest you run into errors. The order of execution is an important aspect of string concatenation that many programmers fail to get right. Let’s explain this with an example:

```
string x = "Four: " + 1 + 3;

Console.WriteLine(x);

// Four: 13

string y = "Four: " + (1 + 3);

Console.WriteLine(y);

// Four: 4
```

In both instances, we perform a concatenation of **Four**. In the first instance, we end up with a string result. A second concatenation is performed but the result, **Four: 13** is not right. The reason for this is because the console performs the operation from left to right, returning a string in each of the operations.

You can, however, avoid such a scenario by introducing parentheses in the operation. Parentheses help you change the order of execution, giving you the correct result. The reason for this is because, in C# programming, parentheses are operators with the highest priority. Therefore, the addition will take place, followed by the concatenation.

The lesson here, therefore, is that if you are performing a concatenation that involves sum numbers and strings, always introduce parentheses to bring order into the flow of operations. Without that, the console will execute code from left to right.

Console.ReadLine() vs. Console.Read()

By now you know that the console input is the most ideal when working with small applications in console communication because of its ease of implementation. Every programming language has a unique method of reading and writing to the console. In C#, the standard input stream is controlled by **Console.In** , from which you can read text data or any other type of data available after the console parses the text.

You will also realize that in most cases, **Console.In** is rarely used. Instead, you will use **Console.Read()** and **Console.ReadLine()** to read from the **Console** class. The easiest way to read from the class is **Console.ReadLine()** . Using this method, you key in the input and press **Enter** when ready in order for the console to read the string as per the instructions issued. Let's explain this with an example below:

```
class UsingReadLine
{
static void Main()
{
Console.Write("Enter car model: ");
string firstName = Console.ReadLine();
Console.Write("Enter car name: ");
string lastName = Console.ReadLine();
Console.WriteLine("You are viewing, {0} {1}!", carModel, carName);
}
}
// Output: Enter car model: Mazda
// Enter car name: Axela
// You are viewing, Mazda Axela!
```

As you can see, reading text from the console using the **Console.ReadLine()** method is simple. When using this method, the console blocks the program from performing any operation until you key in some text, and press **Enter** .

We switch things up a bit when using the **Console.Read()** method. In this method, the console will first read one character at a time, instead of the entire line as we did in **Console.ReadLine()** . Another difference between the two methods is that instead of returning the result as a character, **Console.Read()** returns the character's code. Therefore, you must convert the code back into the character to use it, using the **Convert.ToChar()** method.

As long as the **Console.ReadLine()** method is in use, the **Console.Read()** method is rarely used. This is because of its propensity for errors, and to avoid unnecessary complications.

Chapter 5:
Conditional Statements

Conditional statements are used in programming to instruct the console to perform different actions if a given condition is true. To learn how to use conditional statements, you must also recall how to use comparison operators. Comparison operators help us describe the appropriate conditions that make conditional statements relevant.

There are many comparison operators you can use in C# to compare integers, strings, characters, and other types. Here are the most commonly used comparison operators, which we will also use throughout this chapter:

== Equal to

!= Not equal to

> Greater than

>= greater than, or equal to

< Less than

<= Less than, or equal to

You use comparison operators to compare different expressions, for example, numbers and variables, or even a set of numerical operations. The result of the comparison will always be a Boolean true or false. Let’s explain this using an example below:

```
int weight = 800;
Console.WriteLine(weight >= 400); // True

char gender = 'f';
Console.WriteLine(gender <= 'm'); // False

double colorWaveLength = 2.360;
Console.WriteLine(colorWaveLength > 2.301); // True

int x = 7;
```

```
int y = 9;
bool condition = (y > x) && (x + y < x * y);
Console.WriteLine(condition); // True

Console.WriteLine('Y' == 'X' + 1); // True
```

From the code snippet above, we can compare characters and numbers. It is also evident that the **char** type in this example assumes the characteristics of a number, and for that reason, you can add, subtract, or compare it to other numbers. While this is possible, it is also advisable to use this approach sparingly because you can easily end up with complex code that is difficult to read or comprehend.

If we run the code snippets above, we should have the following results:

```
True
False
True
True
True
```

When programming in C#, you can compare object references or pointers, Booleans, numbers, and characters. Note that anytime you use comparisons, the iteration can affect two object references, two Boolean values, or two numbers. It is also possible to compare expressions that do not have the same types. For example, you can compare a floating-point number with an integer. Note, however, that you cannot always compare data types freely, for example, it is impossible to directly compare numbers with strings.

Role of Comparison Operators in Conditional Statements

In any comparison illustration, for example between characters and integers, what gets compared is their memory binary representation. Instead of comparing the types themselves, the console compares their values. Let’s say we want to compare two integer variables; what is compared is their 4-byte series representation. Let’s explain this with an example below:

```
Console.WriteLine("char 'x' == 'x'? " + ('x' == 'x')); // True
Console.WriteLine("char 'x' == 'y'? " + ('x' == 'y')); // False
Console.WriteLine("4 != 5? " + (4 != 5)); // True
Console.WriteLine("6.0 == 6L? " + (6.0 == 6L)); // True
Console.WriteLine("true == false? " + (true == false)); // False
```

We get the following result if we run the code above:

```
char 'x' == 'x'? True
char 'x' == 'y'? False
4 != 5? True
6.0 == 6L? True
true == false? False
```

When programming in C#, you will realize that not all reference data types contain their values. In some cases, the reference type will contain a memory address to which you can find the desired values. This is common with arrays, classes, and strings. In such cases, the reference data type has a null value, but instead, will act as a pointer to another object whose value you will infer. For such cases, instead of comparing the underlying value of the data type, you compare the address they point to.

It is also possible to have two references that point to the same or different objects. You can also have one of the references pointing nowhere, hence having a null value assigned. In the example below, we have two variables that point to the same object:

```
string str = "soda";
string anotherStr = str;
```

If we run the code, you will have both variables (**str** and **anotherStr**) pointing to the object with the value **soda** . If you want to find out whether two variables point to the same object, you use the comparison operator (**==**). Most of the time, the comparison operator will not actually compare the content of the objects, but instead, it verifies whether they point to the same memory location. In this way, it verifies whether they refer to the same object. Note that when using object type variables, you cannot use size comparisons (**>, <, >=, or <=**). Using this knowledge, let’s look at an

example below:

```
string str = "soda";
string anotherStr = str;
string thirdStr = "sod";
thirdStr = thirdStr + 'a';
Console.WriteLine("str = {0}", str);
Console.WriteLine("anotherStr = {0}", anotherStr);
Console.WriteLine("thirdStr = {0}", thirdStr);
Console.WriteLine(str == anotherStr); // True - same object
Console.WriteLine(str == thirdStr); // True - equal objects
Console.WriteLine((object)str == (object)anotherStr); // True
Console.WriteLine((object)str == (object)thirdStr); // False
```

If you run the code, you should get the following outcome:

```
Console.WriteLine(
str = soda
anotherStr = soda
thirdStr = soda
True
True
True
False
```

From the example above, the strings we used, which you can identify with the keyword **string** , have their values set as objects because they are reference type strings. You will also notice that while they are different objects, the objects **str** and **thirdStr** have equal values. They can, therefore, be found at different locations in the memory.

Take a look at the variables **str** and **anotherStr** . You will notice that they are equal, and are one, and are pretty much the same object. If we compare **str** and **thirdStr** , you end up with an equality. This happens because when

we introduce the comparison operator `==` , we compare the variables of the strings by their underlying value, and not their address. From the examples above, we can thus conclude that using the comparison operator `==` assumes different behaviors when comparing strings. However, when you use the comparison operator on other reference types like classes and arrays, the comparison is done by addresses.

In C#, we use logical operators in expressions to arrive at Boolean (logical) conclusions. The logical operators used in this case are **||, &&, ^, and !.** Let's have a look at how to introduce logical operators in conditional statements below.

You can only use the logical AND, and the logical OR on Boolean expressions. The result can only return true if all the operands are true. Let's see an example of this below:

```
bool result = (5 < 7) && (7 < 9);
```

The expression above will return true because the value in all the operands is true. As you delve into C#, you will come across the logical AND operator referred to as the short-circuit in some texts. This is true because of its nature. Using this operator, the compiler skips all the unnecessary computations. It first examines the first operand, and if the result is false, it does not proceed to the second operand. This is because as long as the first operand is false, by convention, the result of the entire operation cannot be true.

The operator will also result in true if at least one of the two operands is true. Let's see an example of this below:

```
bool result = (5 < 7) || (3 == 5);
```

This result will be true because the first operand is true. The compiler ignores the second operand because it already knows the first part is true.

The logical comparison operators `|` and **`&`** are similar in application to `||` and **`&&`** . The only difference is that in the application, these operators must be calculated in consequence, even though you know the outcome beforehand. For this reason, the operators are referred to as full-circuit logical operators, though they are rarely used.

If you compare an illustration where the two operands are in use, the second part is still executed even if the first one returns false. The same applies when the first operand is true. Take note, however, that as much as the Boolean

operators and bitwise operators are written in the same manner, their actions and results are not the same.

Another example of a full circuit operator is the exclusive OR (^). The operator result returns true if only one of the operands is true. However, if both operands are simultaneously true, the result is false. Let's explain this in an example below:

```
Console.WriteLine ( "Exclusive OR: "+ ((5 < 7) ^ (6 > 7)));
```

If we run the code above, we get the following result:

```
Exclusive OR: False
```

In the example above, each of the operands (**5 < 7**) and (**6 > 7**) is true, hence the compiler returns a false.

"if "and "if-else"

Having learned how to perform comparisons, the next step is to look at implementing logic in C# programming. The conditional statements if and if-else are useful in conditional control. When in use, the compiler first checks for a predetermined condition before executing the statement. The default conditional if statement follows the syntax below:

```
if (Boolean expression)
{
Body of the conditional statement;
}
```

In the syntax above, the Boolean expression can only be either a Boolean logical expression or a Boolean variable. Also, note that a Boolean expression can never be an integer. This rule, however, does not apply in other programming languages like C or C++.

The body of the conditional statement can contain more than one statement. The expression that comes after the **if** keyword will only return a **true** or a **false** . If the expression returns true, the compiler executes the rest of the body of the conditional statement. However, if the expression returns a **false** , it skips the other operators within the body of the conditional statement. Let's explain this with an example:

```
static void Main()
```

```
{
Console.WriteLine("Input two digits.");
Console.Write("Input first digit: ");
int firstDigit = int.Parse(Console.ReadLine());
Console.Write("Input second digit: ");
int secondDigit = int.Parse(Console.ReadLine());
int largerDigit = firstDigit;
if (secondDigit > firstDigit)
{
largerDigit = secondDigit;
}
Console.WriteLine("The larger digit is: {0}", largerDigit);
}
```

If we run this code using the numbers 7 and 9, we should get the following output:

```
Enter two digits.
Enter first digit: 7
Enter second digit: 9
The larger digit is: 9
```

Next, let's see how to handle conditional **if-else** statements. The syntax for an **if-else** statement is as shown below:

```
if (Boolean expression)
{
Body of the conditional statement;
}
else
{
Body of the else statement;
```

```
}
```

In this case, the compiler will calculate the Boolean expression, whose result must be a Boolean true or false. If the result returns true, the compiler executes the body of the conditional statement. However, if it returns false, the compiler ignores the **else** statement, thereby preventing its operators from executing.

On the other hand, if the Boolean expression returns false, the body of the statement in the **else** statement executes. If this happens, the compiler will neither execute the main body of the conditional statement or its operators. Let's explain this with an example:

```
static void Main()
{
int a = 7;
if (a > 9)
{
Console.WriteLine("a is greater than 9");
}
else
{
Console.WriteLine("a is not greater than 9");
}
}
```

From the statement above, a = 7. The compiler will execute the Boolean expression of the else structure, returning the following result:

```
a is not greater than 9
```

When using **if** and **if-else** statements, you can easily run into complications. To avoid that, use the following tips:

First, many beginners struggle with ambiguous code from time to time. To make your code clearer, write in code blocks, and enclose your code in curly brackets after **if** and **else** .

Secondly, you should also make an effort to keep your code neat and cleanly formatted. For this purpose, offset the code inwards by one tab after the **if** and **else** . This makes your code easy to read.

Chapter 6:
Loops

In C# programming, you will often come across code or code snippets that must be executed repeatedly. This is what we refer to as a loop. A loop is a programming convention whereby the compiler is instructed to execute a given code segment repeatedly. There are different types of loops. Usually, the code snippet is repeated in a loop either until a preset condition is met, or up to a preset number of times. You can also have an infinite loop, that which never ends. Considering the memory consumption, you will hardly ever use an infinite loop. However, in cases where an infinite loop is necessary, you must introduce a break operator in the body to prematurely terminate the infinite loop.

While Loops

The easiest and most common loop in C# is the **while loop** . The syntax for this type is as shown below:

```
while (condition)
{
loop body;
}
```

This loop describes an expression that will return a Boolean true or false as the result. The loop condition describes how long the body will be repeated. This is the body of code that is executed each time the code snipped is in a loop (when the predetermined input condition returns true).

A while loop will first evaluate the Boolean expression. If it returns true, the compiler loops the body of code. This process will repeat until a point where the conditional expression returns a false. At this point, the compiler terminates the loop, and the immediate line after the loop body is executed.

Note that the loop body will only execute if the Boolean expression returns a true. If it returns false, the while loop will not execute. At the same time, if the underlying condition that perpetuates the cycle does not break, the loop will persist indefinitely. Let’s look at a simple while loop example that prints

cycles in ascending order below:

```
// Start the count
int count = 63;
// Keep the loop running while the loop condition remains true
while (count <= 68)
{
// Display the cycle value
Console.WriteLine("Cycle : " + count);
// Increase the count
count++;
}
```

If we execute the code above, we should get the following outcome:

```
Cycle : 63
Cycle : 64
Cycle : 65
Cycle : 66
Cycle : 67
Cycle : 68
```

Now, let’s build on this example and see what else we can do with a while loop. The first step is to learn how to add numbers in a sequence as shown below:

```
Console.Write("x = ");
int x = int.Parse(Console.ReadLine());
int num = 1;
int sum = 1;
Console.Write("The sum 1");
while (num < x)
{
```

```
num++;
sum += num;
Console.Write(" + " + num);
}
Console.WriteLine(" = " + sum);
```

The compiler will begin with the variables **num** and **sum** at 1. In essence, we instruct the compiler to start with 1 and add 1 to each of the preceding numbers. Therefore, this is a simple loop where we add 1 to the previous number to get the next number.

In this part of the code **while (num < x)** , we instruct the compiler to keep adding 1 to the numbers, as long as it is within the range of 1 to x. This will persist until the final result when the **num** value is x. If we run the code above and enter the value of x as 6, we will have the following output:

```
while (num < x)
X = 6
The sum 1 + 2 + 3 + 4 + 5 + 6 = 21
```

As you can see, the program will keep running and adding 1 to the next value, until the value of x is 6.

Do-While Loops

The while loop is the basic foundation of loops. From there, we build on to the **do-while** loop. It works in the same way the while loop does, but with a twist. It must check after each loop execution to determine whether the condition is true or false. The syntax for a **do-while** loop is as shown below:

```
do
{
executable code;
} while (condition);
```

In the beginning, the compiler will execute the loop body. After that, it checks the condition of the loop. The loop repeats if the condition is true. However, if the condition is false, the loop terminates. If the condition of the loop never changes and is always true, this loop will run infinitely. It is an

important loop that you can use when you are certain the code sequence will be run repeatedly at least once at the beginning of the loop. Let's look at an example to explain this:

```
Console.Write("x = ");
int x = int.Parse(Console.ReadLine());
decimal factorial = 1;
do
{
factorial *= x;
n--;
} while (x > 0);
Console.WriteLine("x! = " + factorial);
```

In this code, we begin with a result of 1 and keep multiplying the result at each loop with x, and at the same time, reducing x by one value until the value of x is 0. This is a simple example of mathematical factorial expressions, in the format **x*(x-1)*...*1** . You will realize that this sequence will not yield when the value of x is equal to or less than 0. This is because the sequence must perform at least one multiplication. In the example above, we should have the following output:

```
x = 7
x! = 5040
```

Note that this will work flawlessly for small numbers. However, if you input larger numbers for the value of x, for example, x = 143, the decimal type overflows, and you end up with an exception as shown below:

```
Unhandled Exception: System.OverflowException: Value was either
too large or too small for a Decimal.
```

To avoid this problem, you will use the **BigInteger** data type. This data type allows you to use large integers. Provided your computer has sufficient memory, there is no limit on the size of numbers you can use with the **BigInteger** data type.

It might take a while to calculate the factorial using this data type, but you

will not run into **OverflowException** errors. Unfortunately, while this data type is powerful and allows you to do so much, it is heavy on memory consumption, thus slower than **long** and **int** data types.

For Loops

These loops build on the knowledge we have learned about the while and do-while loops. However, they are slightly more complex but are useful in that they reduce the amount of work you perform by writing less code than you would have with the while and do-while loops. The general structure of for loops is as shown below:

```
for (X; Y; Z)
{
N;
}
```

```
for (int i=0; i<10; i++)
{
/* loop body */
}
```

In the structure above, there are four parts, **N, X, Y,** and **Z** . The code initializes at **X** , the loop condition is prescribed at **Y** , while the body of the loop condition is outlined at **N** . At **Z** , we update the commands the loop variables will use. Based on these instructions, the syntax for this kind of loop is as shown below:

```
for (initialization; condition; update)
{
loop's body;
}
```

Explaining this further, the initialization applies to the counter (**int i=0**). We also have a Boolean condition where **i** must be less than **10 (i<10)** , and finally an instruction that updates the counter (**i++**). This counter is one of the distinct features between for loops and other loops. The number of iterations needed in a for loop is always predetermined before you execute

the first line of code.

Note that any elements used in for loops are not usually mandatory. Because of this reason, you can create an infinite loop by omitting them. The initialization stage should look like this:

```
for (int num = 0; ...; ...)
{
// You can use the num variables at this point
}
// You cannot use num variables at this point
```

The initialization code block runs once before the loop begins. At this stage, you declare the loop variables and set the initial values. Note that at this point, more than one variable can be declared.

Next, we declare the condition for the loop. The loop condition looks like this:

```
for (int num = 0; num < 10; ...)
{
// Body of the loop
}
```

Like we mentioned earlier, the compiler will evaluate the condition of this loop each time before it iterates as we saw in while loops. If the evaluation returns true, the compiler executes the code block. If it returns false, the compiler skips this code block, effectively ending the loop.

In the loop body, you have all the variables declared in the initialization block. For example, if we run the following code,

```
for (int i = 0; i <= 6; i++)
{
Console.Write(i + " ");
}
```

We will get the following output:

```
0 1 2 3 4 5 6
```

We can now introduce the **continue** operator. This operator terminates the loop by stopping the most recent loop iteration. Let's use an example to show how it works:

```
int x = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= x; i += 2)
{
if (i % 5 == 0)
{
continue;
}
sum += i;
}
Console.WriteLine("sum = " + sum);
```

In the example above, the code determines the sum of odd integers in the range of **1-x** , which are not divisible by 5 using the for loop. Using **x=13,** we are looking at the following odd integers: (1, 3, 5, 7, 9, 11, 13).

The code initializes with 1 (the first odd integer in our range **x=13**). At each iteration, the code confirms whether the value of **x** has exceeded **(i<=x)** . Since we are only looking at odd numbers, the update instruction is to increase the integers by 2.

Within the loop body, the compiler checks to determine whether the current odd integer is divisible by 5. If it is divisible, the operator **continue** is called into action, skipping the rest of the loop body. If the odd integer is divisible by 5, the code updates the sum with the current odd integer. Using **x=13** , we should have the following output:

```
13
sum = 44
```

Nested Loops

A nested loop is a set of programming instructions where loops are housed

within other loops. When using nested loops, the outermost loops are iterated fewer times than the innermost loops. The general syntax for nested loops is as follows:

```
for ( initialization, verification, update)
{
for ( initialization, verification, update)
{
executable code
}
...
}
```

Immediately after the first for loop is initialized, the body, which houses the nested loop, is executed. The compiler initializes its variables, checks the condition, and executes the code within that loop body. If the condition is true, it is updated and the loop continues until the condition returns false. If we have a false, the second code snippet of the for loop begins, performing the same process as above.

Basically, when implementing a nested loop, you are essentially implementing a for-loop internally and another for-loop externally.

Chapter 7:
Arrays

Arrays are useful when programming a sequence of elements from the same data type. An array is a collection of variables. In C#, the elements of an array are known as indices and are numbered as 0, 1, 2, …, N-1. The length of an array refers to the total number of elements found in the array.

An array can contain different elements. That being said, you will either have reference type elements or primitive type elements. In C# programming, all the elements that make up either of the types mentioned must be of the same type. This way, it is easier to identify a group with similar elements and work on the sequence as one unit.

While arrays can exist in different dimensions, for the purpose of this discussion, and most of the arrays you will come across in C#, we will use one-dimensional and two-dimensional arrays. One-dimensional arrays are known as vectors, while two-dimensional arrays are known as matrices.

Array Declaration

Arrays length is assigned when you instantiate the array, and it also defines the number of elements contained in the array. Note that the array length is fixed, and once it is set, you cannot change it. The syntax for declaring an array is as follows:

```
int[] myArray;
```

In the syntax above, the name of the array is captured in **myArray** . The array must be of the integer type, hence **int[]** . You should also note that by enclosing the integer numbers in the brackets, we are simply highlighting that we have declared a variable that is not a single element, but an array of elements.

If you declare an array type variable, we have a reference. A reference has no value, hence it will direct us to null. Reference points directly to null because they do not have any memory allocated in lieu of the elements.

Moving on from references, we create arrays in C# using the keyword **new** . In doing so, the console assigns memory to the array. An example of this is shown below:

```
int[] myArray = new int[8];
```

From the example above, we create an integer type array with 8 elements. What this means is that in the console's dynamic memory, we just created room for 8 integer numbers, but when they initialize, they will all have their values set at zero. The dynamic memory is the point at which array elements are stored, also known as the heap. Each time you create a new array and allocate memory to the **myArray** variable, its reference point is in the dynamic memory.

As you allocate memory to the array created, you must also indicate the number of elements that will be present in the array. This must be a non-negative integer number, enclosed in brackets, which describes the length of the array. After the keyword **new** , you indicate the type of elements used in the array, to which memory will be allocated.

Elements within an array are initially set at zero values. To use the elements, you must either set them to a default value or initialize them to a different value. Note the difference between C# and other programming languages in that all variables must always have a default value. This is not the case in other programming languages. If you try to access a variable in C# that does not have a default value, you get an error.

The default initial value for variables in C# is **0** when using numeral data types. If you are coding with non-primitive data types, the default initial value can be any relevant equivalent. For example, if you are using the **bool** type, the default initial value can be **true** , or **null** for a reference type. Let's look at an example of this below:

```
int[] myArray = { 1, 2, 3, 4, 5, 6, 7, 8 };
```

What we have done in the instruction above is to initialize the array elements immediately after the array is declared. Using the syntax above, instead of the keyword **new** , you will use curly brackets. Enclosed within the brackets, you will find a list of all the initial values used in the array. Let's use another example below:

```
string[] EPLTop10GameWeek14 =

{ "Liverpool", "Leicester", "Everton","ManUtd", "Chelsea",

"Arsenal", "Tottenham", "Southampton","Wolves", "ManCity" };
```

In the example above, we have created an array of ten elements of the string

type. The string type is the reference type, whose values are initialized in the dynamic memory. In the stack memory, we have allocated a variable **EPLTop10GameWeek14** , which directs us to a section in the dynamic memory where the array elements can be found.

By default, arrays are zero-based, meaning that we start counting from zero. The first array element is always 0, while the next, in an array of **n** elements, is derived as **N-1** .

Elements of an Array

In C#, you can directly access an array using its indicative index. Each element in an array can be identified using the array name and the index number corresponding to the element. In this manner, we can treat elements in the same manner that we treat variables. Let's look at an example below of how to access an element in an array:

myArray[index] = 73;

In the example above, the element is given a value of 73, which is its index position in the array. Each time you try to access an element, the .NET Framework automatically checks the elements to verify the validity of its index. This also helps you tell whether the element is outside the array range. If you try to access an element that is not within the range of the array (an invalid element), you get the following error:

System.IndexOutOfRangeException

This automated check is useful in that it will help you identify errors when coding with arrays. Unfortunately, such checks will always cause a performance lag, given that memory is committed to the check. However, what's a bit of memory lag compared to the magnitude of errors you would run into when trying to run code that references arrays which are impossible to reach?

Chapter 8:
Numeral Systems

Decimal, hexadecimal, and binary are the most commonly used numeral systems in computer programming. To get a better grip of C# programming, you must understand how these numeral systems work, and more importantly, how computers encode different numeral data. Numeral systems have been around for years, dating as far back as ancient times. There are many kinds of numeral systems that have been in use since then. In the modern world, the decimal system is widely used, especially since you can easily count the base numbers on your fingers.

In the realm of computer programming, numeral systems are simply a method through which we can represent finite digits. While many numeral systems have been used in the past, the Arabic numeral system gained more popularity over the years, with digits ranging from 0 to 9.

Each numeral system must have a base. The base is a unique number that is equal to the number of digits used in the system. For example, in the Arabic numeral system, we have 10 digits. This is a decimal system, and we can select any number to act as the base. The base in the Arabic numeral system must have an absolute value, which is neither 1 nor 0. The base can also be a complex number with a sign or a real number.

Non-Positional Numeral Systems

A positional numeral system is a numbering system where the position of digits is significant to the value of the number. What this means is that the digit's number does not have a distinct value. Therefore, since it is not strictly defined, the value depends on the position assigned to the digit. Let's look at an example to explain this below:

532, 25364

In the example above, the digit **2** has a value of 2 in 532, while it has a value of 20,000 in 25364.

For computing purposes, systems with bases of 2, 8, 10 and 16 are more prevalent than any other numeral system you will come across. Below, let's look at brief notation for different numbering systems as used in computer

programming:

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Away from positional numeral systems, we also have non-positional numeral systems. In this system, digits have permanent and consistent values. Such values are not dependent on the position in the number as we saw in the earlier example of 2 in **532** and **25364**. In this category, we have the Greek and Roman numeral systems.

While non-positional numeral systems are used in many instances, they all face an inefficiency challenge in large number notation. For this reason, they are often limited to a few use cases. Because of this inefficiency, they are prone to inaccuracies, especially when determining the values. Let’s take a closer look at each of these numeral systems:

The Roman numeral system sequence is as shown below:

Roman Number: I V X L C D M

Decimal Value: 1 5 10 50 100 500 1000

We already mentioned that the digit position holds no significance over the value of the number. Therefore, to determine the value, you consider the following rules:

First, you add the values if two consecutively represented digits are aligned in such a way that the first value is larger or equal to the second value. Here's an example:

MMC (2100) = 1000 + 1000 + 100

III (3) = 1 + 1 + 1

Second, if two values are aligned in increasing order of their values, you subtract them from right to left. Here are some examples:

MXL (1040) = 1000 – 10 + 50

IX (9) = -1 + 10

MXXIV (1024) = 1000 + 10 + 10 -1 + 5

The Greek numeral system, on the other hand, is a decimal system that groups digits in fives as shown below:

Greek Number: I Γ Δ H X M

Decimal Value: 1 5 10 100 1000 10000

Going by the examples above, the following are some numbering examples used in the Greek numeral system:

ΓΔ = 50 = 5 x 10

ΓH = 500 = 5 x 100

ΓX = 5000 = 5 x 1,000

ΓM = 50,000 = 5 x 10,000

The binary numeral system is the basis of computing. It is widely used because the cost of producing binary numeral devices is lower compared to other systems. Other than that, it is easy to implement for devices with two stable states. That being said, numbers expressed using binary systems are generally too long, which means that they consume a lot of memory for those with large bit numbers. This is, however, only a challenge for human application. We circumvent that problem by using systems programmed with large bases.

Number Conversion

Binary and decimal numbers are the most commonly used numeral systems in programming. From time to time, you will need to convert one to the other,

using a conversion system applicable to each of the systems. The binary system is made up of digits ordered to the power of 2. This means that each binary digit is raised to the power of the position it holds. Let's look at an example below:

Convert the binary number to decimal: 10101_2

$= (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

$= (16_{10}) + (0) + (4_{10}) + (0) + (0)$

$= 20_{10}$

From the example above, you can see the multiplication by 2, raised to the power of the position the digit is located. This is how we convert from binary to decimal.

You can also perform the conversion from decimal to binary numeral systems. To do this, you divide the number by two, leaving a remainder. This is how to arrive at the remainder and quotient. You continue the same routine until you end up with a zero quotient. You will notice that from the remainders, you either have a 1 or 0. Let's work on an example of this, using the number 158:

158/2 = 79 remainder 0

79/2 = 39 remainder 1

39/2 = 19 remainder 1

19/2 = 9 remainder 1

9/2 = 4 remainder 1

4/2 = 2 remainder 0

2/2 = 1 remainder 0

½ = 0 remainder 1

After the division, you take the remainders and represent them in reverse order as shown below:

10011110

Therefore, the conversion of 158_{10} to binary is 10011110_2.

Binary Numeral Operations

The next process you will learn is how to perform normal arithmetic operations using binary numbers. However, you will notice a few differences when computing binary numbers. Let's look at some examples below:

Addition

0 + 0 = 0

1 + 0 = 1

0 + 1 = 1

1 + 1 = 10

Subtraction

0 – 0 = 0

1 – 0 = 1

1 – 1 = 0

10 – 1 = 1

Multiplication

0 x 0 = 0

1 x 0 = 0

0 x 1 = 0

1 x 1 = 1

You can use the bitwise operators AND, and OR to check the values of different numbers. Let's say we want to confirm whether a number is odd or even. To do this, you try to confirm whether the lowest bit in that order is 1. The number is odd if the lowest order bit is 1, and even if it is 0. When programming in C#, you can represent the bitwise AND using the & sign as shown below:

```
int result = integer1 & integer2;
```

You can also use the bitwise operator OR to raise the value of a given digit to 1. This is represented in code as shown below:

```
int result = integer1 | integer2;
```

Next, you can also use the bitwise XOR operator. When used in your code, the compiler processes every binary digit independently. Assuming the operand contains a 0, the compiler will copy the value of the bit in the first operand. In each position with a value of 1 in the second operand, the value is reversed as was displayed in the first operand. This is represented in code as shown below:

int result = integer1 ^ integer2;

Hexadecimal numbers have a base of 16. This means that they use 16 digits to represent all the digits between 0 and 15. Note, however, that in the hexadecimal system, digits are represented from 0 – 9, and then from A – F. In an earlier table, we outlined the corresponding numeral values for the Latin numbers A – F (10 – 15).

Based on this understanding, you can also have hexadecimal numbers like D3 and 2F1F3. To convert numbers from hexadecimal to the decimal numeral system, you first multiply the value of the digit to the furthermost right by 16^0, the one after it by 16^1, and so on until the last one, then you calculate the €. Let's look at an example of this below:

D1E (16) = E*16 0 + 1*16 1 + D*16 2 = 14*1 + 1*16 + 13*256 = 3358 (10)

On the other hand, if you are to convert from hexadecimal to a decimal numeral system, you divide the decimal numeral by 16 then use the remainders in the reverse order as we saw earlier in decimal to binary.

Remember, however, that the remainders are expressed in hexadecimal numeral systems wherever applicable. For example, we have the following:

3358 / 16 = 209 + remainder 14 €

209 / 16 = 13 + remainder 1 (1)

13 / 16 = 0 + remainder 13 (D)

The outcome of the remainders, in reverse order, gives us $D1E_{16}$.

If this process seems tedious or too long, you can also perform the conversion by dividing the binary number into four-bit groups. In some cases, the digits will not be divisible by four, in which case, you do not add the leading zeros for the highest orders in the sequence. After that, you replace each of the groups with their corresponding digits. Let's look at an example of this below:

11100111110_2

For this value, we will first divide into groups of four as shown below:

0011 1001 1110

When we replace each of the half-bytes with the corresponding hexadecimal digits, you end up with $39E_{16}$.

Binary code is useful to programmers because it allows you to store data in the operating memory. The manner in which information is stored is always determined by the data type. It is, therefore, imperative that you learn how to present and process different types of data. Note that computing devices can only process data when it is presented according to the number of bytes, thereby forming a machine word.

When using numeral systems, integers can always be presented either with or without a sign. If you use an integer with a sign, you implicitly introduce a signed order. This becomes the highest order in the sequence, with positive numbers having 0 and 1 as the highest value for negative numbers. All the other orders are used for information purposes only, and for this reason, they can only represent the value of the number assigned to them. If you use an unsigned number, its value can be represented by any or all of the bits.

Chapter 9:
C# Methods

Methods refer to the simple parts of a computer program that solve problems and can use parameters to return a result. A method simply outlines all the data conversion processes involved in returning a result. This is also referred to as the logic of the program. Think of a method in the context of the steps you follow when solving a calculus problem. Once you understand the method, you can use that knowledge to code bigger, more complex programs that can solve complicated problems.

Let's say we want to write a program that calculates the area of a rectangle. That code will be as shown below:

```
static double GetRectangleArea(double width, double height)
{
double area = width * height;
return area;
}
```

The concept of methods in programming follows the rule of divide and conquer. Most problems can be solved by breaking them down into smaller sub-problems. It is easier to define, understand, and solve such smaller problems than to try to resolve complex problems as a whole.

Based on the same principle, computer programs are written to solve problems. First divide the problem into smaller tasks, and by finding solutions to the small tasks you can put them together into one solution for the entire program. These small solutions are referred to as subroutines, which in some programming languages, can allude to procedures or functions. In C#, however, we call them methods.

Granted, there are many reasons why you should consider using methods in programming. Let's have a look at some of the reasons why methods are an important aspect of programming that you must learn.

First, you have a better chance of writing code that is easily readable and properly structured using methods. This is good practice because it makes it easier for someone else to read and maintain your code. Once you write a

program, maintenance consumes a lot of time and resources. It gets even more complicated when adding new features and updates to the program. Once you write and release a program, other developers who have access to it will also invest their time and resources into maintenance to suit their needs.

Second, methods are a good way to prevent repetitive code. This is also related to code reuse. Code reuse refers to lines of code that are used several times in a program. If you have such code, you can highlight it in the method so that you don't have to rewrite it all the time. Code reuse will also help you avoid repeating code.

The challenge of repeating code is that it slows down the program and increases the risk of errors. Besides, when developers are fixing errors in code blocks, there's a good chance they will only fix some of the errors present in repeating code. Therefore, as long as some of the code still has errors, the program will still be buggy.

Declaring Methods

There are three kinds of actions you can perform to an existing method:

1. **Declaring a method -** This is a process where you call the method's identity so that the rest of the program can recognize it.
2. **Implementing a method -** This refers to creating a method. You enter code that eventually solves a given problem. Since the code already exists within the method, we can look at this as the logic of the method.
3. **Calling a method -** This is where you invoke the declared method from the section of the code where a problem is to be solved.

Before you learn how to declare a method, you should know where you can call methods in C#. Methods only exist when they are declared and enclosed within brackets of a class. On the same note, you cannot declare a method within the body of another method. Let's look at a simple example of declaring a method below:

```
public class HelloCSharp
```

```
{ // new class starts here
// declare the method here
static void Main(string[] args)
{
Console.WriteLine("Hello C#!");
}
} // new class ends here
```

To declare a method, you use the syntax below:

[static] <return_type> <method_name>([<param_list>])

From the syntax above, the type of result the method will return is identified by **<return_type>** . The name of the method is listed as **<method_name>** , while inside **<param_list>** , you list all the parameters that will be used in the method. Note that you can also have an empty list.

We can also use **Main()** to identify the elements present in the method declaration. An example of this is as shown below:

static void Main(string[] args)

In the example above, the method does not return any result, hence the returned value is **void** . The parameter in this example is the **string[] args** array. If you have an empty parameter list, all you have to do is key in **()** after the name of the method.

Method Implementation

Once you declare a method, the next step is to implement it. The implementation of a method will include the body of the code, which is executed when you call the method. Note that to represent the logic of the method, the code has to be within the body of the method in question. The body of the method refers to the code enclosed within the curly brackets as shown in the example below, which follows the declaration.

Static <return_type> <method_name>(<parameters_list>)

{

```
// The code in this segment is the body of the method
}
```

The method encapsulates the real programming work expected of the code. Therefore, if you are writing an algorithm to solve some problem, this is where it is included. Note that you can never declare a method within the body of another method.

Each time you declare a variable within the body of a method, the variable becomes a local variable with respect to the called method. The extent of a local variable starts at the line where you declare the variable and ends at the closing curly bracket of the method body. This extent is referred to as the variable scope or area of visibility of the variable. Assuming you try to declare another local variable after declaring a variable using a similar name, the compiler returns an error. Let's explain this with an example below:

```
static void Main()
{
int m = 82;
int m = 13;
}
```

You get an error when you try to run the code above. This is because the compiler cannot allow you to use the name **m** for two local variables. As a result, you will get the following error message:

A local variable named 'm' is already defined in this scope.

If we declare a local variable in a given block of code, it becomes a local variable for that block of code. Therefore, the area of visibility for that code will also start from the beginning of the variable declaration to the end, where you have the closing bracket.

Next, we look at what it means to invoke or call a method. This refers to executing the method code found in the method body. Invoking a method is quite easy, given that all you have to do is write the name of the method, the round brackets, and a semicolon as shown below:

```
<method_name>();
```

Immediately after you execute a method, it takes charge and initiates a

sequence of events that will run your program. However, if you call the method and call another method within the active method, the original method relinquishes control to the newly called method. It will only return control back once it terminates execution. After that, the original caller method continues from the line it paused before calling the new method.

Chapter 10:
Recursion

Recursion refers to an instance where an object can be defined by itself or contains itself. It is a technique where a method can call itself to solve problems. If this happens, the method is said to be recursive. One of the reasons why recursion is an important technique in programming is because it helps make code easily readable.

The simplest example of recursion is the Fibonacci numbers sequence. In this sequence, the next number is arrived at by adding the previous two numbers. Let's look at an example of the sequence below:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Recursive functions are useful in simplifying complex problems. However, you must also be attentive when using recursive functions in programming.

A direct recursive is an instance where a call to a method arises within the body of the same method. Assuming you have three methods, X, Y, and Z, if X calls Y, Y calls Z, and Z calls X, we have a mutually recursive or indirectly recursive function.

Before you start a recursive function, you must first make sure that you will get a definite result after a given number of recursive calls. As a precautionary measure, it is wise to have at least one scenario where you can arrive at a solution directly without invoking a recursive call. Such scenarios are referred to as the bottom of recursion.

In the Fibonacci numbers example we used earlier, the bottom of recursion is a situation where n is equal to or less than 2. You don't need to run a recursive function to arrive at this, because by default, the first two elements in the Fibonacci sequence must always be equal to 1. In an instance where the recursive method used does not have a bottom of recursion or base, we end up with an infinite, hence the outcome **StackOverflowException** .

Recursive tasks are generally complex in nature. For this reason, you must create smaller tasks (subtasks), solve them, and use the successful algorithm in a recursive function. By recursively using the algorithm, you arrive at solutions to the subtasks, and in the long run, you find a solution to the initial problem.

Recursive Calculations

To understand recursive calculations, we will use a simple factorial example. The factorial of x, written as (**x!**) should give us the product of all the integers between 1 and x inclusive. This can also be defined as **0! = 1** , and expressed as follows:

x! = 1.2.3...x

To create the definition above, it is easier to use the corresponding factorial definition as shown below:

x! = 1, for x = 0

x! = x.(x-1)!, for x > 0

In many cases, you have to analyze a problem and work out the values before you can identify the first integers. The simplest case is always found at the bottom of the recursion. In our case, that is **x = 0** , where the factorial value = 1. For other instances, you must first solve **x - 1** and then multiply the result by **x** . Since there are a definite number of integers between 0 and x, you will arrive at the bottom of the recursion after a definite number of steps. From here, you can then write the recursion method to compute factorials as shown below:

```
static decimal Factorial(int x)
{
        // The bottom of the recursion
        if (x == 0)
        {
        return 1;
        }
        // The method will call itself (a recursive call)
        else
        {
        return x * Factorial(x - 1);
        }
```

```
}
```

Using the method above, you can easily write an application to read integer values from the console, compute the factorials, and print the result. Let's look at an example of this below:

```
using System;
class RecursiveFactorial
{
        static void Main()
        {
                Console.Write("x = ");
                int x = int.Parse(Console.ReadLine());
                decimal factorial = Factorial(x);
                Console.WriteLine("{0}! = {1}", x, factorial);
        }
        static decimal Factorial(int x)
        {
                // The bottom of the recursion
                if (x == 0)
                {
                return 1;
                }
                // The method will call itself (a recursive call)
                else
                {
                return x * Factorial(x - 1);
                }
        }
}
```

Assuming that we are looking for the factorial of x = 6, we will have the following outcome:

x = 6

x! = 720

Recursion vs. Iteration

To explain the concept of recursion, factorial calculations are the most natural example. As simple as they are, there are many other instances where a recursion might not be the best solution. Naturally, a recurrent solution fits a recurrent problem definition. You will hardly encounter difficulties using that approach. However, iterative solutions might actually be a better option, and more efficient than recursive solutions. Here is an example of an iterative syntax:

```
static decimal Factorial(int x)
{
        decimal result = 1;
        for (int i = 1; i <= x; i++)
        {
                result = result * i;
}
        return result;
}
```

The lesson here is that before you rush to a recursive solution for your problems, you should also consider the possibility of an iterative solution, then choose the most viable of the two.

This brings us to the question, which is the better alternative between recursion and iteration? Granted, the first thing you have to consider is the nature of the algorithm. If you have a recursive algorithm, it makes sense to implement a recursive solution. Compared to an iterative solution, this would be more elegant and easy to read.

There are some instances where it might be difficult to define the algorithm,

or prove the difference between a recursive and iterative one. Recursion is generally recommended for shorter, simpler problems.

At the same time, it is worth noting that recursion is generally resource-intensive. The computer must set aside some new memory for each recursive call. If your program makes a lot of such calls, you will end up with a stack overflow as you run out of memory. There are also instances where a simple recursion solution might be more challenging to implement than a relevant iterative solution.

The bottom line is that regardless of its resource-intensive nature, recursion is still one of the most powerful techniques you will come across in C# programming. However, you must weigh your options before you use it. More often, using an unnecessary recursion solution where an iteration would be more efficient leads to maintenance problems. You might also end up with an inefficient program that is difficult to understand.

We mentioned the Fibonacci sequence earlier, as the simplest and most fitting example of recursion. We will build on that and show why recursion might not be the best idea. Let's look at an example below, where we use recursion to find the x^{th} Fibonacci number in the sequence:

```
static long Fib(int x)
{
        if (x <= 2)
        {
        return 1;
        }
        return Fib(x - 1) + Fib(x - 2);
}
```

This syntax is simple and easy to understand. At a glance, it looks like the best solution. Unfortunately, this is not the case. Let's say the x^{th} value is set at 141. The program will take a very long time to compute the value, and waiting would be pointless. Besides, implementing the solution would be equally inefficient, given that each recursive call creates two more calls, which further create two more and so on. Bearing this in mind, the Fibonacci tree will grow to exponential lengths, which is cumbersome.

You can overcome this problem through memoization. Memoization helps you optimize recursive methods by saving the numbers already calculated in

arrays. This way, it will only call the array if the calculation you are about to perform has not been done yet.

```
using System;
class RecursiveFibonacciMemoization
{
        static long[] numbers;
        static void Main()
        {
                Console.Write("x = ");
                int x = int.Parse(Console.ReadLine());
                numbers = new long[x + 2];
                numbers[1] = 1;
                numbers[2] = 1;
                long result = Fib(x);
                Console.WriteLine("fib({0}) = {1}", x, result);
        }
        static long Fib(int x)
        {
                if (0 == numbers[x])
                {
                numbers[x] = Fib(x - 1) + Fib(x - 2);
                }
                return numbers[x];
        }
}
```

By optimizing this new computation, you get the answer almost instantly because it performs a linear computation. Running the code above, you should have the following output:

n = 141

fib(141) = 131151201344081895336534324866

An iterative solution for calculating the next number in the Fibonacci sequence means you calculate the numbers consecutively. In this case, you use only the last two elements calculated in the sequence to determine the next element in the sequence. An example of this is as shown below:

```
using System;
class IterativeFibonacci
{
        static void Main()
        {
                Console.Write("n = ");
                int n = int.Parse(Console.ReadLine());
                long result = Fib(n);
                Console.WriteLine("fib({0}) = {1}", n, result);
        }
        static long Fib(int n)
        {
                long fn = 0;
                long fnMinus1 = 1;
                long fnMinus2 = 1;
                for (int i = 2; i < n; i++)
                {
        fn = fnMinus1 + fnMinus2;
        fnMinus2 = fnMinus1;
        fnMinus1 = fn;
        }
return fn;
```

```
        }
    }
```

This method will return the solution faster. It is elegant and shorter than a recursive solution. Other than that, it does not hog memory. From our discussion, it is safe to recommend you avoid recursion unless you are certain about what runs under the hood in your code. While it is a powerful programming tool in C#, it could be just as lethal when used in the wrong place.

In principle, you do not need to use recursion if the computational process is linear. Iteration is simpler, and performs efficient computations, for example when performing factorial computations. In such computations, you determine the elements in a sequence wherein each of the next elements in the sequence depend on the previous ones only, and not any other.

The main difference between a linear computation and an exponential computation process as used in a recursion solution is that the recursion is called only once in each step, and only in one direction. Below is an example of a linear computation syntax:

```
    void Recursion(parameters)
    {
            Perform computations;
            Recursion(some parameters);
            Perform computations;
    }
```

As you can see, if the body of the recursive method only features one recursion call, the iteration is obvious and you do not need to use recursion. However, in the event of a branched computation, for example when using N nested loops, you cannot replace the process with an iteration. Restrict the use of recursion to situations where it is the simplest, easiest to comprehend, and most efficient solution to your problems, and more importantly, where an obvious iterative solution does not exist.

Chapter 11:
Exception Handling

An exception refers to an instance where the normal operation of a computer program is interrupted. Most exceptions are often unexpected, so you should have a way to detect and react to them when they happen. In the unfortunate event of an exception, the normal flow of the program is interrupted. That being said, the state of the program is saved, and the control process is moved along to the exception handler.

Exceptions are raised by the program when an error arises, or if the program encounters an unusual situation. Let's say you are trying to open a file that was deleted from the hard drive. Since the file no longer exists, the path leading to it is inaccurate. In this case, the code instructions for opening the file will raise an exception and display the relevant error message.

From what we have explained so far, exception handling is an important part of object-oriented programming. By now you can already relate to the number of times your computer will raise exceptions when you try to process something out of the ordinary.

An exception in .NET refers to an object that identifies an unexpected event or an error that interrupts the normal flow of your program. When this happens, the method being executed sends a special object with unique information about the error, the program, or operation where the error was identified, and its state at the moment the error occurred. All the relevant information about the error can be found in the stack trace.

You have to understand the construct of a stack trace in order to use it effectively. You can read the following information from the stack trace:

1. Information on the call stack
2. Name of the exception class
3. A message with any additional information about the error

When you read the call stack dump, you will notice that each line has the following information, or something similar:

at <namespace>.<class>.<method> in <source file>.cs:line <line>

Let’s look at an example of an exception:

```
class ExceptionsDemo
{
        static void Main()
        {
                string fileName = "MissingTextFile.txt";
                ReadFile(fileName);
        }
        static void ReadFile(string fileName)
        {
                TextReader reader = new
          StreamReader(fileName);
                string line = reader.ReadLine();
                Console.WriteLine(line);
                reader.Close();
        }
}
```

In the example above, we are trying to open a text file that is missing. The method throws an exception in the first lines, **ReadFile()** . This code will still compile successfully. However, at runtime, we will receive an exception if we did not have the **MissingTextFile.txt** file. An error message is displayed on the console, complete with information on how and where the error was identified.

Building on this, exception handling refers to the process wherein exceptions are identified. The process runs through the common language runtime (CLR), wherein each exception is passed on to the kind of code that can handle the issues arising.

When an error is triggered, the local variables do not initialize, and instead, return **null** as the default value. In this case, the CLR will not process any of the lines of code that come after the method where the error happened. Thus, the program stays interrupted until the exception is processed by the relevant handlers.

Exception handling is an important part of object-oriented programming because errors can be processed centrally. This is a significant improvement from procedure-oriented programming where every function can return an error code.

An exception will generally arise when a problem is encountered such that it becomes impossible to successfully complete an operation. The method supporting the said operation can catch and handle the error, or move it along to the method calling the operation. This process makes it easier to run the program because error handling can be designated to different levels in the program call stack. In this way, you write programs in such a way that error management is efficient and flexible, and this also makes it easier to handle unusual situations when they arise.

Hierarchy of Exceptions

You will learn about two types of exceptions in the .NET Framework as shown below:

1. **ApplicationException** - These are raised by applications running on the system. They are defined by different app developers and are used by unique programs.
2. **SystemException** - These are raised by the runtime. They are defined within the .NET libraries and can only be used by the .NET Framework.

Each of the two types of exceptions has a different exception class hierarchy, with unique characteristics. When programming an app, it is always wise to inherit exceptions from the **ApplicationException** . It is good practice only to inherit the **SystemException** within the .NET Framework.

In the .NET Framework, every exception must have an exception class. The two types of exceptions mentioned above can directly inherit from the exception classes. You will also realize that for most of the exceptions you come across when running a program, **ApplicationException** and **SystemException** are the recognized base classes.

Within the exception class, you will find a copy of the call stack at the precise moment the exception was raised. You should also find in this class, a short message that describes the error raised by the method. In some cases, you might also find within the exception an internal exception, also known as a nested, wrapped, or inner exception. This is an important concept that saves time and resources. Instead of each exception appearing in isolation, you can have an exception chain that links all the relevant exceptions together.

Chapter 12:
Strings and Text Processing

Think about your computing activity for a moment. It almost always revolves around reading some text files, searching online for some keywords, keying in user data, and so on. All this data is stored in the form of text, which makes up strings. The strings are then processed in C#.

A string is simply a sequence of characters stored in memory in a specific address on your computer. In the Unicode table, all characters have a unique serial number in the .NET Framework. In C# programming, strings are handled through the class **System.String** . Strings allow you to work on computers for most of the tasks you need to process. Below is an example of a string in C#:

string greeting = "Good Morning!";

In the example above, we have created a string type variable known as **greeting** , with the context of **Good Morning** . This is simply a character array. Instead of declaring the class, you can also declare the variable **char[]** , and enter each character of the rest of the details. There are, however, a few problems with this. First, it is time and resource consuming to key in the characters one by one. Second, the fact that you must key in the characters one by one means that you do not know the text length beforehand. This means that you are unsure whether the text will fit into the preassigned space in your array. Finally, manual text processing is cumbersome.

It is worth noting that **System.String** is not always the best method. Therefore, it is wise to learn a few other ways of representing characters. Compared to other data types, the **string** class meets the threshold for object-oriented programming principles. Values used in this class are stored in the computer’s dynamic memory, while its variables maintain a reference to any objects stored in the dynamic memory.

An important feature that defines strings is that they are immutable. This means that the sequence of characters stored within the string cannot be changed directly. If you have to change the content of a variable once it has been assigned, it can only be saved to a different location within the dynamic

memory. From here, the variable can point to it.

You will also realize that strings and char arrays (**char[]**) share similarities, but they are not immutable. Just as you would expect with arrays, strings have identities through which you can tell how unique each string is. This way, you can access strings using their indices. To access the character of a given position in the string, you must use the indexer operator **[]** . Note, however, that using the indexer, you can only enjoy read properties for the characters, not read and write. Let's look at an example below:

```
string str = "wxyz";

char ch = str[1]; // ch == 'x'

str[1] = 'w'; // Compilation error!

ch = str[73]; // IndexOutOfRangeException
```

When using strings, you must be careful about how to use punctuation marks. For example, you identify the quoted character by assigning a forward slash before it. Here is an example:

```
string quote = "Her flight arrives \"on Tuesday\"";

// Her flight arrives "on Tuesday"
```

The quotation marks used in the example above are actually part of the text. You add them to the variable after the backslash. By doing this, you instruct the compiler that they are part of the content, and not used to mark the beginning or end of the string. This process where you use special characters within the code is referred to as escaping.

When declaring a string, you are essentially declaring the data type **string** . Note that at the point of declaration, you have not assigned any variables to the string, neither have you allocated memory on the computer for it. All you have done so far is to instruct the compiler that you are creating variable **str** , and when it is used, it should process it as a **string** type. Therefore, no variable has been created at the point of declaration yet, and for that reason, no memory allocated either. The value for the declared string so far is **null** . Note, however, that the value null does not mean the string contains an empty value. Null is a special value, and if you try to manipulate a string of this value, the compiler returns the following error **NullReferenceException** .

Once you declare a string, the next step is to create and initialize it. This

process is known as instantiation. At this point, a section of the dynamic memory is assigned to the string. You can instantiate a variable in any of the following methods:

1. Passing an operation value which will return a string

You can instantiate a string by passing the value of the operation or expression, thereby returning a string result. The result can be anything from a method that results in values of different variables depending on the instructions passed. Let's look at an example below:

```
string blog = "http://blog.gardens.me";

string info = "I have a blog at: " + blog;

// I have a blog at: http://blog.gardens.me
```

From the example above, we have created the **info** variable by concatenating a variable and a literal.

2. Assigning the value of a different string

By assigning the value of a different string, you are essentially redirecting the string variables or value to another string. In this case, you first instantiate the source variable. The assigned variable will then assume the value of the source. The string class in this case acts as a reference to a different address as defined in the first variable.

Since the variables are all directed to the same place, they both receive the same dynamic memory address, and for that reason, they hold the same value. In such an instance, altering the values of either of the variables only affects the altered variable, and not the redirected one. The reason for this is because of string immutability.

3. Assigning a string literal to it

This simply means assigning some text content to the string variable. This method of instantiation is recommended when you are already aware of the values expected to be stored in the variable. An example of this is as shown below:

```
string blog = "http://blog.gardens.me";
```

In the example above, you instantiated the variable **blog** with the string literal

values referring to the online blog address.

The next step is learning how to read strings and print them to the console. Earlier in the book, we talked about the **System.Console** class. We will build on that for this section. Below is simple syntax for reading from strings:

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

As we learned earlier, the **ReadLine()** method waits for you to key in some values and only responds with the variable name once you press Enter on the keyboard. The same class applies when printing data to the console. The syntax is as shown below:

```
Console.WriteLine("I have a blog at: " + blog);
```

Since we are using the **WriteLine()** method, the console will display the message (**I have a blog at**), and **blog** as the value of the name variable. You will also note that at the end of the name variable, the compiler adds the character for a new line. To avoid this and have the messages appearing on the same line, use **Write()** instead.

String Operations

Now that we already know how to create and print strings, our next step is how to perform different operations on them. We will look at some of the methods you use for string operations in C# below.

String Comparisons

While there are several parameters you can use to compare strings, most of the string class comparisons come down to the particular task you need to perform using that string. For example, let's say we want to compare two strings to determine whether or not their values are equal. To do this, we implement the **Equals()** method together with the == operator. For this operation, you will receive a Boolean result (**true** or **false**).

This method checks and compares each character in the strings under investigation. Let's look at an example below:

```
string grade1 = "A";
string grade2 = "a";
Console.WriteLine(grade1.Equals("A"));
```

```
Console.WriteLine(grade1.Equals(grade2));
Console.WriteLine(grade1 == "A");
Console.WriteLine(grade1 == grade2);
// Console output:
// True
// False
// True
// False
```

In the example above, the console has identified that we are comparing a small letter and a capital letter. Therefore, if we compare the grades “ **A”** and “ **a”** using the **Equals(0)** method, the compiler returns **false** .

Contrary to the simple example above, practically, the character casing rarely comes into focus when comparing two strings. You can circumvent the character casing and only focus on the content of the string using the **Equals()** method and introduce the **StringComparison.CurrentCultureIgnoreCase** parameter. Using this parameter in the code above, the compiler will return a true if we compare the grades “ **A”** and “ **a”** . The expression should look like this:

```
Console.WriteLine(grade1.Equals(grade2,
StringComparison.CurrentCultureIgnoreCase));
// True
```

Lexicographical Comparisons

Apart from equality, you can also compare different strings in alphabetical order. Usually, when comparing numbers, you can use the operators > and < . This, however, is not possible when comparing strings. Instead, to compare strings in alphabetical order, you use the **CompareTo()** method.

Using the **CompareTo()** method, you can compare the values in two strings to identify the appropriate alphabetical order. For this comparison, you must ensure that both of the strings under review have the same number of characters, which must also match. For the purpose of this example, the strings “gear” and “rear” cannot be compared because while they might have the same length, they do not have the same first character.

In such string comparisons, the compiler will return either a negative, positive, or zero value. A negative value means that the first string comes before the second string. A zero value means that the strings are equal, while a positive value means that the second string comes before the first. Let’s explain this with an example:

```
string grove = "gRove";

string gravy = "gravy";

Console.WriteLine(grove.CompareTo(gravy));

Console.WriteLine(gravy.CompareTo(grove));

Console.WriteLine(gravy.CompareTo(gravy));

// Console output:

// 1

// -1

// 0
```

In the example above, the **CompareTo()** method does not ignore character casing, so it identifies a mismatch between the **”R”** and **”r”** in the first and second strings. Because of this subtle difference, the compiler returns a positive 1.

If we swap the position of the first and second string, the compiler will return a negative 1 because the comparison now starts at **gravy** instead of **grove** . In the last comparison, the compiler returns a 0 because we perform a self-comparison.

We can also make an alphabetical comparison while ignoring the character casing. For this comparison, we use the **string.Compare(string strX, string strY, bool ignoreCase)** method. This method serves the same purpose as the **CompareTo()** method, but because of the **ignoreCase** , segment, it overlooks character casing. Let’s look at an example of this below:

```
string beta = "beta";

string grade1 = "gRadE";

string grade2 = "grade";

Console.WriteLine(string.Compare(beta, grade1, false));
```

```
Console.WriteLine(string.Compare(grade1, grde2, false));
Console.WriteLine(string.Compare(grade1, grade2, true));
Console.WriteLine(string.Compare(grade1, grade2,
StringComparison.CurrentCultureIgnoreCase));
// Console output:
// -1
// 1
// 0
// 0
```

In the example above, we use **Compare()** and by introducing the **StringComparison.CurrentCultureIgnoreCase** parameter, the compiler does not consider the difference between the character cases. It is important to mention at this juncture that when using the **Compare()** and **CompareTo()** methods, small letters alphabetically come before capital letters.

Keep this in mind because the rule might not always apply in all programming languages. For example, in the Unicode table, small letters come before capital letters. This simple rule can be confusing depending on how you code. Therefore, always remember that in C#, the alphabetical string comparison contravenes the Unicode table arrangement.

Operator Comparisons

The **!=** and **==** operators can be used to make comparisons using the **Equals()** method. Let's look at a simple example to explain this concept:

```
string str1 = "Good Morning!";
string str2 = str1;
Console.WriteLine(str1 == str2);
// Console output:
// True
```

In the example above, we are comparing two matching strings, **str1** and **str2**, hence the compiler returns **true**. This happens because while the second

string does not have any reservation in the dynamic memory, it references a reserved position for the variable in the first string. For this reason, the strings share a common address, and if we compare them for equality, the compiler returns true.

Let’s look at another example to expand on this concept:

```
string beg = "Beg";

string begin = "Begin";

string copy = beg + "in";

Console.WriteLine(copy == begin);

// True
```

In this example, let’s direct attention to the strings **begin** and **copy** . The first variable assumes the value **Begin** . The second variable, on the other hand, derives its value from combining a variable and a literal to return a value equal to the value of the first variable.

Up to this point, each of the variables is pointing towards a unique memory block. However, note that the content in each of the memory blocks is not the same. Using the operator `==` , the comparison returns a **true** result.

Chapter 13:
Defining Classes

In computer programming, one of your key objectives is to implement some ideas and solve problems. When tackling a big problem, it makes sense to create a simpler concept of it, which does not necessarily represent the entire problem but gives you a glimpse of the key facts useful towards achieving the end objective. Object-oriented programming languages are some of the most commonly used languages in the world at the moment. This is primarily because the language syntax is somewhat similar to the semantics of the normal human language.

Through this kind of programming, you have access to a lot of tools you can use to outline different object classes. Classes in this context refer to a definition or specification of a cluster of objects. By this definition, a class will represent identified patterns that describe different behavioral states of the cluster of objects.

Borrowing from this explanation, an object refers to a copy of the definition of a class, also referred to as an instance. If we create an object from the description of a class, say for example m, we can refer to it as an object of the type (m).

Let's say you have the class type **Mazda** . This class will describe some of the features you expect of cars in this category. Some of the objects you can find in this class include **Axela, Demio,** and **Verisa** .

A class essentially defines the objects and data types that describe it. The object in a class contains the data, while the data contains information about the state of the object. Other than the state, the class also defines the behavior of objects within it, by the actions you can expect of the objects. In object-oriented programming, we describe the behavior of objects in a class by declaring **methods** within the body of the class.

Elements of a Class

There are several elements that make up a class. The following are the key elements:

1. **Class declaration** - this is where we declare the name of the

class. An example of this is as shown below:

public class **Mazda**

2. **Body of the class -** The body of a class is outlined after the class declaration. The body must also be enclosed within the curly brackets. Below is an example:

```
public class Mazda
{
// The body of the class comes here
}
```

3. **Constructor -** You use the constructor to create a new object. Below is an example of a constructor:

```
public Mazda()
{
// Write some code here
}
```

4. **Fields -** This refers to the variables you declare within the class. The data represented by the variables tells you the state of the object. An example is as shown below:

```
// Define the field
private string name;
```

5. **Properties** - they are used to define the characteristics of a class. The values of such characteristics are stored in the fields of the objects. An example is shown below:

```
// Define the properties
private string Name { get; set; }
```

6. **Methods** - We discussed methods earlier in the book and realized they perform specific roles in any block of programming code. Objects, for example, derive their behavior

from methods depending on the class type. It is through methods that we are able to implement algorithms and data handling protocols effectively.

Implementing Classes and Objects

To use classes, you must first learn how to create objects. This you can do using **new** and a set of other constructors within the class. This way, you create an object from the class type. For each object you want to iterate, you must assign a variable to the object of the underlying class type. This is how the variable retains its reference to the object. If done correctly, you should be able to call methods and object properties and access other variables, too.

In recent examples, we defined the class **Mazda** that describes the car Mazda, probably in a dealership. We further added the method **Main()** to the class. In the following example, we will learn how to use the elements discussed above to create some more objects, add properties to the objects, and call methods on the said objects.

```
static void Main()
{
        string firstMazdaModel = null;
        Console.Write("Enter first Mazda model: ");
        firstMazdaModel = Console.ReadLine();
        // Here we use a constructor to identify a specific Mazda
  model
        Mazda firstMazda = new Mazda(firstMazdaModel);
        // Here we use a constructor to identify a different
  Mazda model
        Mazda secondMazda = new Mazda();
        Console.Write("Enter second Mazda name: ");
        string secondMazdaModel = Console.ReadLine();
        // Here we use the properties to name the Mazda model
        secondMazda.Model = secondMazdaModel;
        // Here we identify a Mazda model
        Mazda thirdMazda = new Mazda();
        Mazda[] Mazda = new Mazda[] { firstMazda,
  secondMazda, thirdMazda };
        foreach (Mazda Mazda in Mazda)
```

```
            {
            Mazda.Color();
            }
}
```

If we run this code, we have the following output:

```
Enter first Mazda Model: Axela
Enter second Mazda Model: Demio
Axela color is: Black!
Demio color is: White!
[Unnamed Mazda] is White!
```

In the example above, we used **Console.ReadLine()** to derive the name of the objects of the type **Mazda** , which the user keys in. After that, the second string is entered using the variable **firstMazdaModel** . This is the variable we will also use to create the first object from the class type **Mazda - firstMazda** , by assigning it to the constructor's parameter.

The second **Mazda** object is created without using a string for the name of the car in the constructor. Instead, we also use **Console.ReadLine()** to derive the name of the car, and then from there, we assign the value to the property **Model** .

The third object is derived from the class type **Mazda** , which we used for the default car model, whose value is **null** . In the **Color()** method, any Mazda model whose name we don't know (**name==null**) will be printed as **[Unnamed Mazda]** .

At this juncture, we successfully created an array from the type **Mazda** by initializing three of the new objects we created. Finally, we created a loop that runs through the array of objects of type **Mazda** .

It is important to note that each time you create an object in the .NET Framework, it is built from two parts. The first part, the significant data, holds all the object's data and can be found in the dynamic memory of the operating system. The second part, the reference part to the created object, is found in a different part of the operating system memory. The reference part stores the parameters and local variables associated with the object's methods.

If, for example, we create a class **Mazda** , some of the properties we can assign to it include color and model. From the same class, you can also create

a variable for the year of manufacture, which is pretty much a reference to the main object and is located within the dynamic memory. Note that the reference, in this case, is only a variable that you can use to access objects.

Using Namespaces in Classes

When programming in C#, one of the challenges you will encounter, especially when working with custom classes, is that you must save them using the file extension **.cs** . Using the **.cs** file, you can define different structures, classes, and data types. While it is not mandatory for the compiler to do this, it makes sense to store each class in a separate file that corresponds to its name. For example, you should store the class **Mazda** using the file **Mazda.cs** .

C# namespaces are generally a collection of classes that are connected logically, but without a strict illustration on how to store them in the file system. If you need to include a namespace in declared classes, you must use a directive. You don’t always have to use them, but where necessary, you must show them on the first line of your class file before you declare the class or any other data type.

After inserting the namespace, you declare the namespace of the classes used in the file. Remember that it is not mandatory to declare classes within the namespace, but it is good practice because it helps to organize your code better, and will also be useful if you have some classes in your code that have the same name.

Note that within a namespace, you can find other namespaces, classes, interfaces, structure, and other kinds of data. For example, the nested namespace **System** will also hold the namespace **Data** . This can be written in full as **System.Data** , but it is still nested within the **System** namespace.

In the .NET Framework, the full name of a class is written as follows: the namespace to which the class is declared, followed by the class name as shown below:

<namespace_name>.<class_name>

This way, it is easier to use data types from different namespaces without necessarily writing their full names.

Modifiers and Visibility

Modifiers are reserved words in C# programming, which can be used to add more information to the compiler, and any necessary code relevant to the modifier. There are four access modifiers used in C#:

1. Public
2. Private
3. Protected
4. Internal

These modifiers can only be used before the following class elements:

1. Class declaration
2. Fields
3. Properties
4. Methods

Through the access modifiers mentioned above, you can control visibility (access levels) to different class elements to which they are applied. The access levels used in the .NET Framework are as follows:

1. Public
2. Private
3. Protected
4. Internal
5. Protected internal

For the purpose of this discussion, we will only discuss internal, public, and private. Using the public access level before an element, you instruct the compiler that the element can be accessed from any class, regardless of its assembly, from the namespace in use. Using the public access level also means that there are no restrictions to it in terms of visibility.

The private access level is that with the most restrictions to elements and class visibility. Using this modifier, you instruct the compiler that the element to which it is applied cannot be accessed from any other class other than the class to which it is defined. This restriction applies even if the separate class trying to access it exists within the same namespace as the class to which the

element is defined. Note that the private access level is often set by default, such that it applies by default if you do not have any visibility modifier before an element in the class.

The internal access level limits access to class elements only to files located in the same assembly. This means that the elements can only be accessed by files that are used in the same project. Assemblies in the .NET Framework refer to collections of different resources and types that form a logical unit. Assemblies are found in **.dll** or **.exe** files, which are binary files. You must also have noticed this by now: all the files in the .NET Framework and C# exist within assemblies.

Let's explain class visibility using a simple illustration. We have two classes, **X** and **Y**. Class **X** can access all the elements of class **Y** if it can create an object from the class type **Y**. The same also applies if it can access the unique fields and methods used in class **Y**, as per the unique access levels applicable to the fields and methods under consideration.

Based on this illustration, as long as class **Y** is not visible to class **X**, the visibility levels of the fields and methods in class **Y** will not change a thing. Therefore, an outer class can only have public and internal access levels. Inner classes, however, can be defined using other access levels.

Chapter 14:
Working with Text Files

You work with text files all the time in programming. For this chapter, one of the most important aspects of text files you will learn about is a stream. It is an essential aspect of any input-output library you will come across in programming. Streams play an important role in programming because it is through them that your program learns how to read or write data to external sources like servers, other computers, or files.

Streams are essentially an order of byte sequences that are sent from an input device to an output device. This can also be from one application to another. Streams are written and read in sequence, such that the recipient receives them in the same manner they were sent. They are basically an abstract platform through which two applications or devices can communicate effectively.

In the computing world, streams are the primary means of communication. It is through them that we can create a communication network between two or more remote computers. Communication between computers, therefore, is simply reading and writing through a stream. Let’s look at an example of printing a file. Essentially, you sent some byte sequences to the stream connected to the printing port. Each time you read or write to or from a file, you basically open a stream to the file where the reading or writing takes place, then close the stream when you are done.

Streaming Basics

Most computing devices in use today implement streaming for data access. Streams allow communication between remote computers, programs, and files. Note that the sequence of bytes must be in order. Streams must be ordered and organized. Any attempt at distorting or influencing the order or organization of information flow through a stream makes the information unusable.

When we mention that streams support sequential data access, we must once again stress the importance of sequence. Any data in a stream can only be handled or manipulated in the order in which it is received from the stream. Therefore, order and sequence are mandatory in a stream.

Building on that, you realize that you cannot randomly access data in a stream. The access must be sequential, and strictly so. Depending on the situation for which you need data access, there are different types of streams you can use. We have streams that work with text files, streams that use binary files, and others that only work with strings. For any form of communication across a network to be successfully established, you must use the appropriate stream. Knowledge of the types of streams will help you select the right one for each application. Without that, you can forget about effective communication.

Like any other task you perform on a computer, you must always ensure you close the stream once they serve the purpose for which you opened them. Without this, there is always a risk of data loss, file damage, and so on.

There are a number of operations you can perform using streams. Let's have a look at them below:

1. **Opening (creation) -** This is the point at which you connect a data source to the stream or another stream. Let's say you are working on a file stream. You pass the name of the file and the file mode to the compiler. The file mode is the action that will take place on the file, for example, writing, reading, or both.
2. **Reading data -** At this point, you extract data from the stream. Reading takes place in sequential order from the present position of the stream. Note that reading is a blocking operation. A blocking operation is one whereby you are unable to do anything until something that precedes the blocking operation happens. For example, you cannot read until the data you are supposed to read arrives.
3. **Writing data -** This is where you send data to the stream in an appropriate manner. Writing also takes place in sequential order from the present position of the stream. Unlike reading data, writing may or may not be a blocking operation depending on circumstances before the data is sent. For example, there are instances where you can still write to data before the sent data arrives.
4. **Positioning -** In a stream, positioning, also referred to as

seeking, implies moving the current stream position. Positioning is always done relative to the current stream position. It is, however, not possible with all streams, so you can only do it with streams that support positioning. Network streams are an example of streams that do not support positioning. Most file streams, on the other hand, are fully compliant.

5. **Disconnecting (closing) -** Once the stream has served its purpose, you disconnect it. This frees up all resources that were in use. It is always advisable to disconnect the stream as soon as you are through, for resource efficiency. Note that once some resources are committed to the stream, they cannot be used by any other users, including any program on the computer that might require the resources, but run parallel to it.

Types of Streams

When programming in .NET, you will find all classes relevant to working with streams in the **System.IO** namespace. Some important features of streams that you must understand include the hierarchy, functionality, and organization. The input-output stream class is the highest order in the hierarchy of streams. This stream class outlines the core functions of all the other streams, and for that matter, you cannot instantiate it.

Next, we have buffered streams. These do not have any extra functionalities, but are useful when writing or reading data by creating a buffer. The buffer created is purely for performance enhancement. We also have streams that introduce additional functionality to reading and writing data. In this case, you can have streams that convert data into any manner of files to allow easy access. You also have streams that compress or decompress data, encryption and decryption streams, and so on. Such streams are often attached to a different stream, for example, a network or file stream, such that by activating their role in the stream, they make it easier for the attached stream to process the data, hence enhancing functionality.

In the **System.IO** namespace, we have the following main classes:

- Stream
- FileStream

- BufferedStream
- GZipStream
- NetworkStream
- MemoryStream

Note that whichever of these streams you use, you must always close the stream once you finish what you are working on.

Streams are widely divided into two, binary and text streams. Binary streams usually work with raw or binary data. For this reason, they lack specificity and are universal. Therefore, you can use them to read information from any kind of file. You will primarily read or write from binary streams using the following classes:

- **FileStream**

The **FileStream** class offers different methods of reading and writing to a binary file. Other than that, it also allows you to perform other activities like confirming the number of bytes available, skipping some bytes, and closing the stream.

- **BinaryWriter**

The **BinaryWriter** allows you to write different binary values and primitive data types to the stream using a unique encoding. It only uses one method, **Write()** , whose role we had seen in earlier chapters. Through the method **Write()** , you can record any data type to the stream.

- **BinaryReader**

Through the **BinaryReader** , you can read binary values and primitive data types created by the **BinaryWriter** to the stream. It allows you to read data from any integers, character arrays, floating points, and so on.

Text streams are almost similar to binary streams, but they only support text data. They only work with strings (**string**) and character sequences (**char**). Since they are only ideal for text files, text streams are not ideal when working with other binaries.

In the .NET Framework, the primary classes for working with text streams are **TextWriter** and **TextReader** . These classes are abstract, and for that reason, you cannot instantiate them. They basically describe the underlying

functionality for reading or writing to classes where they are used. The following methods are used in text streams:

- **ReadLine()** - This method will return a string after reading one line of text
- **ReadToEnd()** - This method returns a string, but will read the stream to the end
- **Write()** - We use this method to write a string to the stream
- **WriteLine()** - This method will only write one line of text to the stream

The classes **StreamWriter** and **StreamReader** naturally inherit the functionalities of the classes **TextWriter** and **TextReader**, thereby implementing their read and write roles on a file. Before you create an object of the type **StreamWriter** or **StreamReader**, you must first ensure you have a string complete with the file path. With those in place, you can implement any of the methods you are already aware of to read and write to the console.

Working with Text Streams

There are many ways of reading files in C#, but the simplest and easiest is to use the **System.IO.StreamReader** class. This is because it resembles reading from a console. Note that **StreamReader** can work with streams, but it is not a stream. It only gives us a comprehensively easy way to read from text files.

To read a file, create a **StreamReader** from the filename relevant to the path of the file. This is important because it will also reduce the chances of errors. An example of this is as shown below:

```
// We create a StreamReader for the relevant file
StreamReader reader = new StreamReader("testfile.txt");
// The compiler reads the file
// Once you are through, close the reader to free up resources
reader.Close();
```

It is always advisable to avoid using the full file path, and instead, work with relative paths. This reduces the amount of code and further reduces the risk of

errors. It is also easier to install and maintain the program.

Let’s look at an example to highlight the difference between full and relative paths below:

Example of a full path:

C:\Temp\Test\testfile.txt

Example of a relative path:

..\..\testfile.txt

If you have to use the full path, you must be careful to pass the full path to the file, without forgetting the backslash. The backslash helps to distinguish folders along the path to the file. An easier way of representing such files in C# is to use a quoted string or a double slash in front of the string literal. For example, our scenario above **C:\Temp\Test\testfile.txt can be represented as follows:**

```
string fileName = "C:\\Temp\\Test\\testfile.txt";
string theSamefileName = @"C:\Temp\Test\testfile.txt";
```

It is worth noting that while relative paths might seem difficult to use because you must consider the project directory signature, the project can change several times throughout its life, so it is always wise to avoid using full paths.

The other problem with using full paths when writing a program is because it ties down the project to the environment. This means that the program can only exist within the environment within which it was created. It also makes your program untransferable. Therefore, if you were to move the program to a different computer, you will have to verify all the file paths to ensure the program works correctly. Using a relative path, however, is effective regardless of the environment, and makes your program portable.

Chapter 15:
Data Structures

Lists and linear data structures are some of the most important forms of presentation you will use in programming. Learning about data structures is important because you will learn how to work with different aspects of classes in C#. When writing any program, you generally work with different kinds of data. You add and remove elements according to the requirements of your program. At the same time, you also have to find different methods of data storage relevant to the job at hand. This is where data structures come in.

Data structures are sets of data that are organized according to unique mathematical or logical laws. They are an important consideration in programming because your choice generally determines how efficient the program will be. Efficiency in this case is not only limited to resource and memory allocation at execution but also in terms of the amount of code you might have to write for the program. In programming, you will mostly write lots of code to build efficient solutions. It is only credible that you also learn how to code efficiently.

An abstract data type (ADT) defines different structures in terms of their properties and permitted operations, without necessarily focusing on their implementation. This way, ADTs can be implemented in different ways in a program, and perform at different efficiency levels. Below are the different data structures you will come across in programming:

1. Linear data structures
2. Tree-like data structures
3. Sets
4. Dictionaries
5. Other data structures, including graphs, bags, and multi-sets

For this discussion, we will focus on linear data structures. Knowledge of data structures is crucial for programming because it is almost impossible to efficiently write programs without them. Alongside data structures, another important aspect of programming is using algorithms.

Working with Lists

The most common data structures you will use in programming are lists. These are linear data structures that define all the series, sequences, and rows used in programming. A list is an ordered sequence of elements. Think of it as your to-do list. You can read each of the tasks on the list (elements), add or remove tasks as you see fit, or even reorganize their order of priorities.

A list must have a distinct length, and the elements on the list must be arranged in order. Elements on a list can be added at different positions and moved around accordingly. Every ADT used in programming must describe a specific interface. Below are some method definitions you can get from an ADT:

- **int Add(object)** - this method will add an element at the end of the list
- **void Clear()** - this method will delete all the elements on the list
- **void Insert(int, object)** - this method will add an element at a specific position on the list
- **void Remove(object)** - this method will remove an element from the list
- **void RemoveAt(int)** - this method will remove an element from the list, at a specific location
- **int IndexOf(object)** - this method will tell you the position of a called element

The methods mentioned above can be used in different instances especially when working with arrays. An array will generally perform most of the roles that the methods on the ADT list above can perform. However, the difference is that arrays practically are fixed in size, so you cannot add any new elements to an array.

That notwithstanding, you can still implement a list using an array by increasing its size automatically. Let’s look at an example to explain the operations on a to-do list. We will work with a list of tasks, add, insert, and remove a few tasks, then print the list to the console. We will also try to ascertain whether some tasks are available on the list:

class CustomArrayListTest

```
{
        static void Main()
        {
                CustomArrayList<string> todoList =
                new CustomArrayList<string>();
                todoList.Add("Cleaning");
                todoList.Add("Painting");
                todoList.Add("Repairs");
                todoList.Add("Mowing");
                todoList.Add("Cooking");
                todoList.Remove("Repairs");
                todoList.Insert(1, "Studying");
                todoList.Insert(0, "Jogging");
                todoList.Insert(6, "Relax");
                todoList.RemoveAt(0);
                todoList[3] = "I have to finish " + todoList[3];
                Console.WriteLine("Complete the following
  tasks today:");
                for (int i = 0; i < todoList.Count; i++)
                {
                Console.WriteLine(" - " + todoList[i]);
                }
                Console.WriteLine("Position of 'Cooking' =
  {0}",
                todoList.IndexOf("Cooking"));
                Console.WriteLine("Position of 'Mowing' =
  {0}",
                todoList.IndexOf("Mowing"));
```

```
                Console.WriteLine("Can I go Shopping? " +
                todoList.Contains("Shopping"));
        }
    }
```

If we run the code above, we should have the output below:

```
Complete the following tasks today:
- Cleaning
- Studying
- Painting
- I have to finish Mowing
- Cooking
- Relax
Position of 'Cooking' = 4
Position of 'Mowing' = -1
Can I go Shopping? False
```

The list above is a static list, and while it might be effective, it has one major flaw. The operation to remove and insert new items from within the array will involve rearranging the elements. This is doable for a small list. However, if you are dealing with a large list, this will definitely slow down your performance. For this reason, you can consider using a linked list. Let's explain how this works:

Linked lists are dynamic. When using such a list, the compiler will first check if the index you need is available. If it is not available, it raises an exception. After the exception, the compiler considers one of three possibilities. First, the list can remain empty after removing the exception. This removes everything on the list. Second, if the element to be removed is found at the beginning of the list, the new list can begin from the element right after the removed element. If the element we removed was the last one, the new list can start at **null** .

Finally, if the element to be removed is found at the end of the list or in the middle, the element before the removed one is redirected to start at the

element before it, or a **null** if it is the last one.

When you are done, always check to ensure that the tail of the list links to the last item on the list. If the element you removed was the tail element, ensure it links to the removed element's predecessor.

ArrayList Class

Once you learn how to work with different list implementations, you can look at classes used to pull data structure lists. The **ArrayList** is one such class, whose implementation is similar to the static list we discussed earlier. Using this list, you can add, remove, and search for elements. Below are some of the important members you will use in this class:

- **Add(object)** - You use this to add a new element to the list
- **Remove(object)** - You use this to remove an element from the list
- **Count** - This tells you the number of elements on the list
- **Clear()** - This will remove all the elements from your list
- **this [int]** - This is an indexer through which you can access any element according to its position

As we have already seen, one of the challenges you will experience with this array implementation is to resize the inner array, especially when you are adding or removing elements to it. This problem is easily solved by creating a buffer within the array. This allows you to add or remove items from the list without resizing the array at the point of addition or removal.

Since the **ArrayList** class is of the untyped category, you can use it to store any kind of element, including strings, numbers, or even other objects. Let's look at an example of this:

```
using System;
using System.Collections;
class ProgrArrayListExample
{
static void Main()
{
ArrayList list = new ArrayList();
```

```
list.Add("Welcome");
list.Add(7);
list.Add(2.46271);
list.Add(DateTime.Now);
for (int i = 0; i < list.Count; i++)
{
object value = list[i];
Console.WriteLine("Index={0}; Value={1}", i, value);
}
}
}
```

By running the code above, the compiler will give us the outcome below:

```
Index=0; Value=Welcome
Index=1; Value=7
Index=2; Value=2.46271
Index=3; Value=22.12.2020 19:12:16
```

In the example above, we created the **ArrayList** and then added different kinds of elements. In particular, the elements we added were of the types **string** , **int** , **double** , and **DateTime** .

One of the challenges that you will encounter using classes that use the **System.IList** system like the **ArrayList** class is that each time you add an object to that class, it is added as an object type. If you search for an element at a later date, the compiler returns it as an object. Given that all the elements on your list might not be of the same type, you need to find a solution to this problem. You also have to consider the fact that conversion from one type to the other might take a lot of time, and significantly reduce the performance of your program.

Generic classes are the easiest way to solve the problem above. These are unique classes that are meant to work with more than one type. When you create this type, you specify the types of objects that will be used in them. Let’s look at a simple example of this below:

GenericType<M> instance = new GenericType<M>();

By indicating the type of element in question, we create a generic class, which in the example above, is **GenericType** . The data type **M** as used in the example above can refer to any member of the class **System.Object** . In the recent example, we could use the data type **M** to represent **DateTime** . Here are some examples where this can be applied:

List<int> intList = new List<int>();

List<bool> boolList = new List<bool>();

List<double> realNumbersList = new List<double>();

Another important generic variant you can introduce into your work is the **List<T> class** . When using this class, you must also indicate the elements that will be used in the class. These are the elements that can be held, such that you can always replace the denoted element type with a real data type. By creating a list that only contains integer numbers, we implicitly cannot add other data types to the list. Therefore, if you try to add any other object type other than integer numbers to the **List<int>** , the compiler returns a compilation error. This is a useful application because the compiler automatically helps you avoid common mistakes when using data collections of different types.

The **List<T>** class comes in handy at different instances. Since it uses the inner array to keep elements, the array will simply resize if it is overfitted. Bearing that in mind, the following are possibilities when using this class:

It is incredibly fast when searching for elements using their index. The number of elements in your class does not matter; it will search through each of the elements with the same speed. It is also a faster way of adding a new element to the class. This makes sense because, by addition, you essentially increase the array's capacity. Increasing the capacity of an array is a really slow operation. However, by using the **List<T>** class, the insertion speed is not reliant on the number of elements in the class, hence it works faster.

Unfortunately, searching for an element in the class by its value is slow using this approach. The speed challenge comes in because the method performs as many comparisons as the number of elements in the class.

Another challenge you will encounter using this approach is that of adding or removing elements to the class. Each time you add or remove an element,

you must shift the other elements that are not affected directly by the change. This procedure is slow, especially if the elements being moved around are not found at either end of the array.

Considering the factors above, it is only advisable to use the **List<T>** class if you are certain you won't need to add or remove elements to the class frequently. On the same note, it also makes sense only when the new elements will be added at the end of the list, or if you need to access them by their index and not element values.

Conclusion

You have carefully read the entire book and are certain to have gained valuable knowledge in programming that will work for you in many instances. One thing you learn about programming is that knowledge is always transferrable. Therefore, what you learn in C# will come in handy in any other programming language that you might come across in the near future. The beauty of learning C# is that it is one of the fundamental programming languages used in computing. Therefore, even as technology advances from time to time, the knowledge learned will always be useful.

An important lesson you learn when writing code is about sustainability. Your code should stand the test of time, and be usable by any other programmer who comes across it. There's no better way to do that than to program in a language whose premise is sustainability. C# is the undertone of Microsoft. Looking at the mammoth of a company and player in different fields of programming Microsoft is, you can rest assured that C# is here to stay. Microsoft is at the forefront of future programming projects, including machine learning. This gives you confidence knowing that the foundation of C# programming will always be aligned with future advancements. This takes care of the fear that many programmers have today, whether their preferred programming language will still be relevant over the coming years.

Whether you move on to other programming languages or not, C# offers the ideal foundation for programming. With time, you will work alongside other programmers on small and large projects alike. You will also come to learn the value of shared knowledge, such that you can all work on a project that needs different skill sets. This way, as each programmer plays their role, there is no disconnect regardless of the programming languages you use.

Like math, you can only get better at programming by practicing every day. This is especially true when you are just learning the fundamentals. Spare a few minutes to code every day. This puts you in a better position and gives you a firm grasp of the core of C# programming. Try your hand at different challenges. Create solutions to simple problems around you. This gives you a simpler knowledge of how programming works, and more importantly, forms the foundation on which you will continually learn and become a better programmer.

From time to time, you will encounter challenges you cannot solve. Some of these might be simple, others difficult, but they should not discourage you. Learning is a collective effort when it comes to computer programming. There are many communities out there from which you can learn, share ideas, and find solutions to your current code challenges. It might take you a while, but as long as you are committed to the cause, you will enjoy the world of C# programming.

Where do you go from here? After reading this book and learning the basics of C# programming, the next step is to cast your net further. Delve deeper into C# programming. Tackle harder challenges. Look for code camps and hackathons to learn more about C#. You will be amazed at how much there is to do out there. Besides, as the world evolves, so does programming. You will come across programmers out there who have combined different languages to build some amazing projects.

With the foundation you learned from this book, you can also take a leap of faith and learn another programming language. There is so much out there, so you might be spoilt for choice. Note that there is nothing wrong with learning other programming languages. They all borrow from each other in one way or the other. To become an all-around programmer, you should be able to code proficiently in more than one language. That being said, you should at least work towards being an expert in two or three languages.

The C# platform is closely related to other paradigms that should interest you, like PHP, Java, Ruby, and so on. When choosing any of these, try to dwell on languages and technologies that are supported by your native platform. This makes it easier to learn the new interest, given that most of the concepts flow easily across platforms.

Having learned C#, it is also advisable to look at databases. Tables and relations are an important part of database programming. When building an application, you must be able to carefully work with tables and the relations between the content therein. You will also learn how to build queries, to select and update data in different databases. There are several databases you can learn, including MySQL and Oracle.

It would be pointless to talk about programming without touching on the obvious question lingering in your mind; the Internet, and websites. Of course, you can also learn different techniques and technologies in use for

building some of the most amazing websites. In a world as interconnected as ours is, everyone is learning how to work with websites, web apps, and mobile apps. Learning concepts like HTML, JavaScript, and CSS should be on your to-do list. Start by creating simple websites and graduate into dynamic websites with dynamic content. You can also learn how to create mobile applications.

Practicality is another point you should consider. As you learn C# programming, how practical are your skills? Put that knowledge to use. Look for, and challenge yourself to more serious projects. All the applications you see online today were once just an idea in someone's head. They probably started with simple illustrations, then built the programs. Programming using advanced software might be difficult, but you will get there someday. Start with simple programs and build your way up to the level of complexity that you desire.

One thing that I can guarantee you is that there is a whole world of opportunities out there that you should conquer. C# programming and programming, in general, will be your ladder to these opportunities. Get coding and grasp these opportunities. You should be at the forefront in programming, and positioning yourself for greater things that lie ahead.

Good luck in your programming journey.

References

Mills, H. D. (1988). Principles of Computer Programming : A Mathematical Approach. Wm. C. Brown.

Ullman, L. E., & Signer, A. (2006). C++ Programming. Peachpit Press.

Zheng, L., Dong, Y., Yang, F., & De, W. (2019). C++ Programming. Berlin De Gruyter.