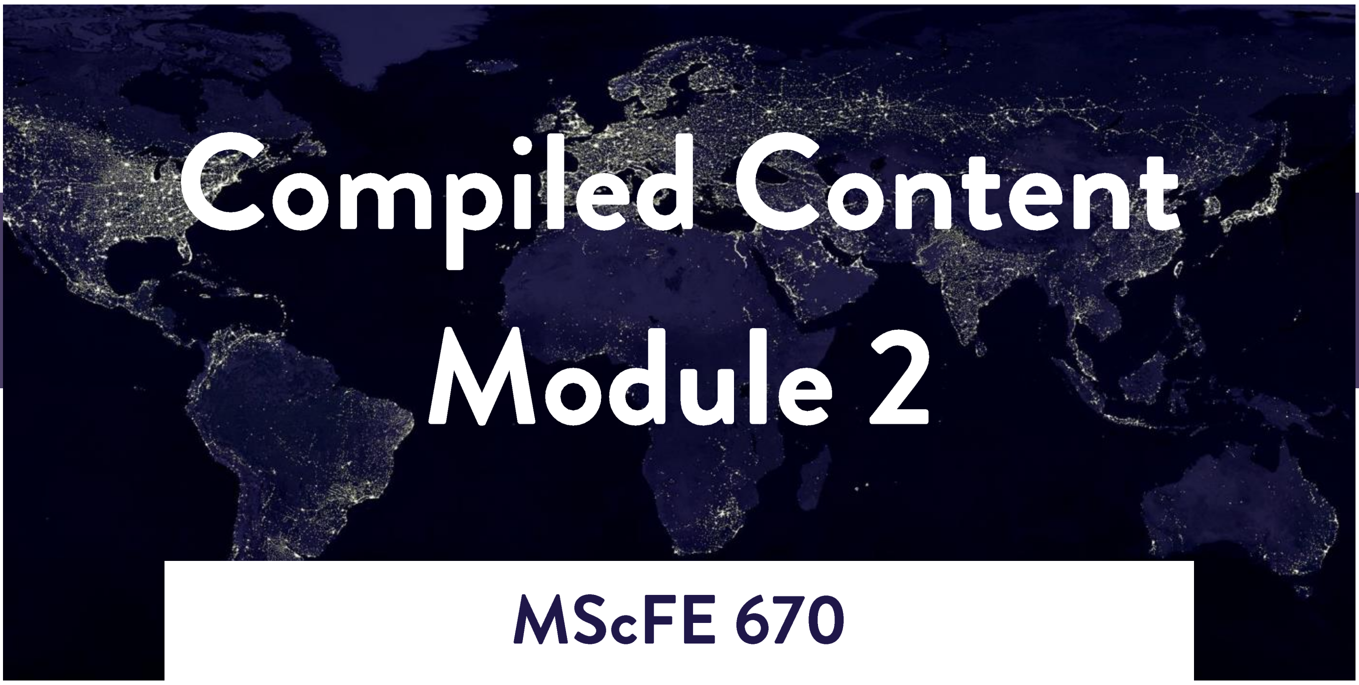
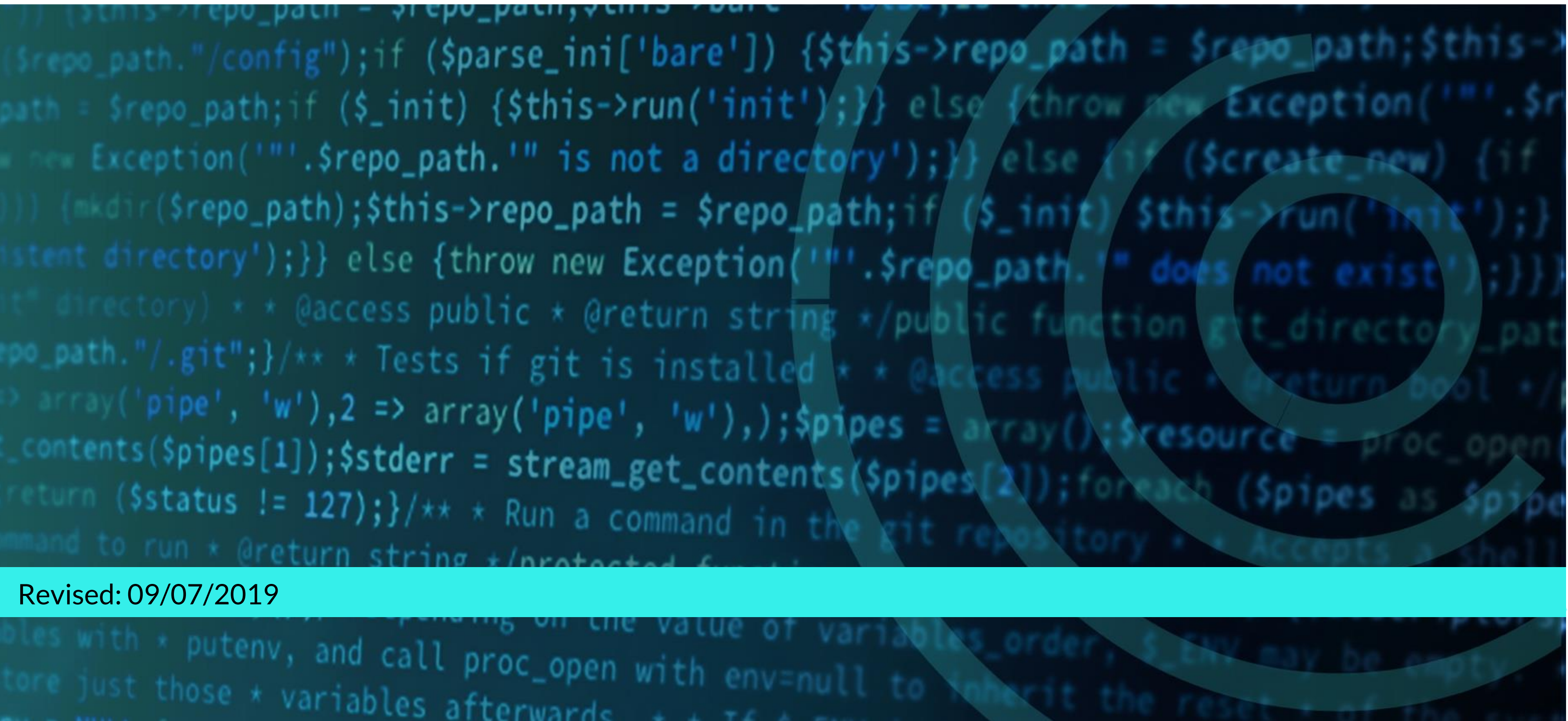
WORLDQUANT UNIVERSITY



Compiled Content Module 2

MScFE 670

Data Feeds and Technology



Revised: 09/07/2019

Table of Contents

Module 2: Introduction to C#

This module begins by introducing the fundamentals of C# (pronounced C-sharp) language such as setting up a C# work environment, C# data types, and C# data structures. The module continues by introducing principles of object-oriented programming as they relate to C# programming and ends by discussing practical examples of methods and objects implemented in C#.

Unit 1: C# Language Fundamentals

Introduction

C#, pronounced "C-sharp," is a general purpose, object-oriented programming (OOP) language developed and maintained by Microsoft. C# is used extensively in the creation of many enterprise systems because it integrates well with other software products in Microsoft's .NET ecosystem. If you'd like to learn more about the .NET ecosystem, check out the further reading section at the end of these notes. C# is also useful for financial programming applications because it integrates well with Microsoft Excel, a very popular spreadsheet application that sees wide usage in industry.

We will cover object-orientation in depth at a later stage, but for now it is simply important to know that this means that we model our programs using objects. Objects may contain data, as well as methods to operate on this data. Often this approach allows for much more intuitive modeling of complex systems, which would otherwise be challenging to represent using only primitive data types.

For example, if we are creating a program that works with information about various stocks, we might model each stock as an object that contains data regarding that stock's current price, full company name, and time series pricing. This stock object might also contain various methods such as "MeanPrice" and "PriceVariance" to calculate useful information from the raw, time series data.

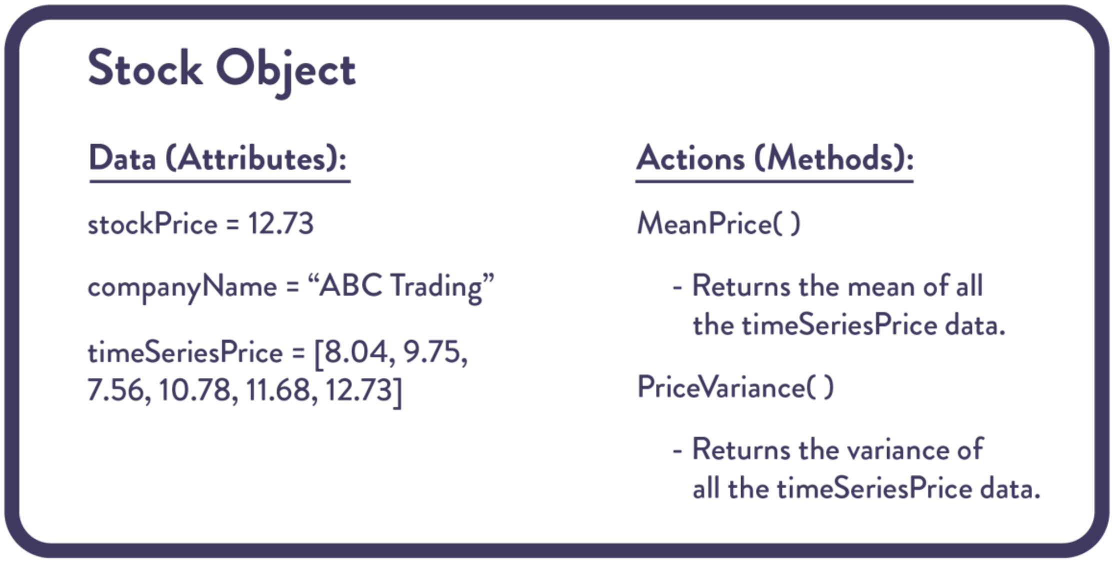

Stock Object
Data (Attributes):
stockPrice = 12.73
companyName = "ABC Trading"
timeSeriesPrice = [8.04, 9.75, 7.56, 10.78, 11.68, 12.73]
Actions (Methods):
MeanPrice( )
- Returns the mean of all the timeSeriesPrice data.
PriceVariance( )
- Returns the variance of all the timeSeriesPrice data.

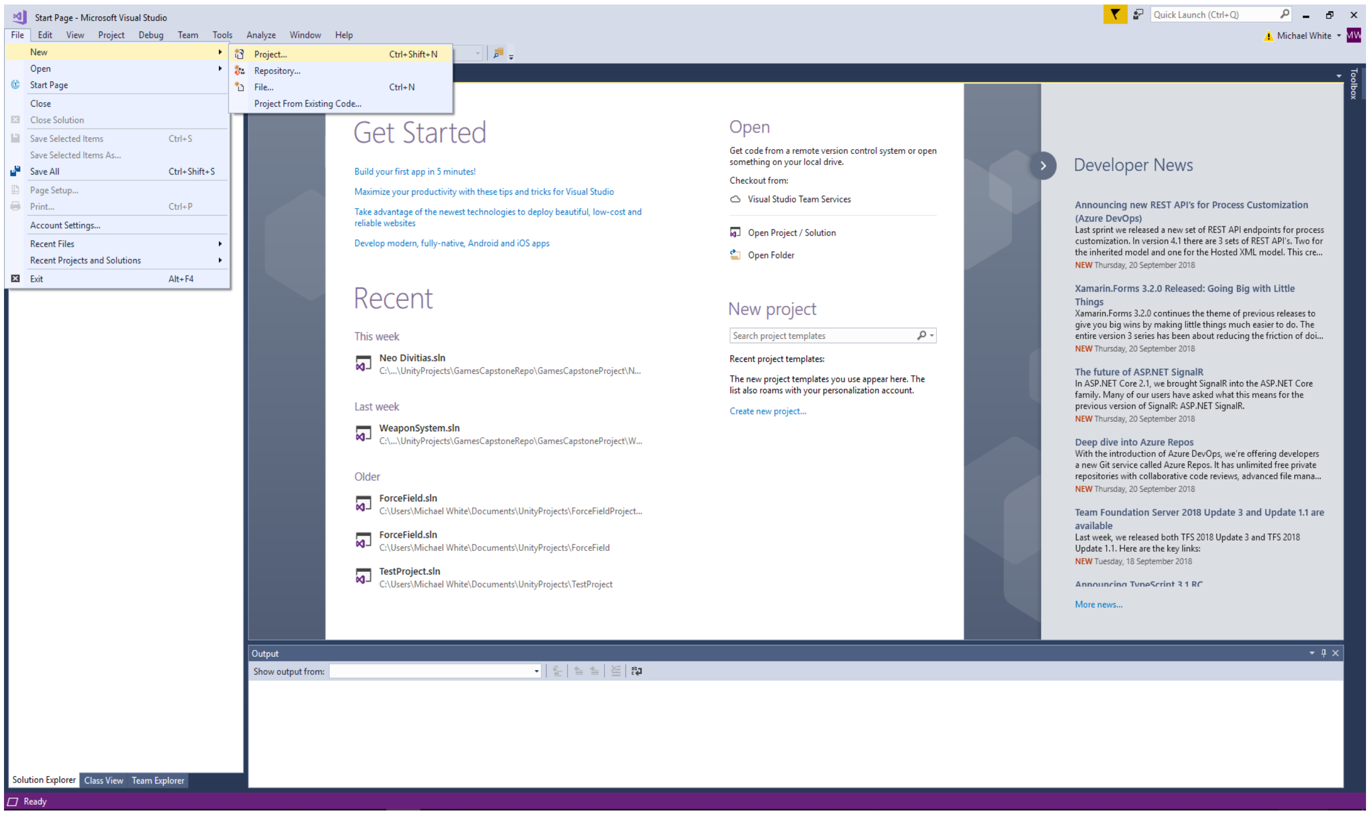
Figure 1: Diagram of the Stock object

Setting up your C# work environment

For the sake of this module, we will make use of C# version 7.3. The most popular work environment for C# development is Microsoft Visual Studio. We will make use of Visual Studio 2017 Community Edition, which is freely available from [https:// visualstudio.microsoft.com/ downloads/](https://visualstudio.microsoft.com/downloads/).

Hello, world

Let's dive right in and create a simple program using Visual Studio and C#, before we start to look at some of the theory behind the language. We start by launching Visual Studio, at which point the window depicted in Figure 2 should display.


Start Page - Microsoft Visual Studio
File Edit View Project Debug Team Tools Analyze Window Help
Quick Launch (Ctrl+Q)
Michael White
New
Open
Start Page
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Page Setup...
Print... Ctrl+P
Account Settings...
Recent Files
Recent Projects and Solutions
Exit Alt+F4
Project... Ctrl+Shift+N
Repository...
File... Ctrl+N
Project From Existing Code...
Get Started
Build your first app in 5 minutes!
Maximize your productivity with these tips and tricks for Visual Studio
Take advantage of the newest technologies to deploy beautiful, low-cost and reliable websites
Develop modern, fully-native, Android and iOS apps
Recent
This week
Neo Divitias.sln
C:\...\UnityProjects\GamesCapstoneRepo\GamesCapstoneProject\N...
Last week
WeaponSystem.sln
C:\...\UnityProjects\GamesCapstoneRepo\GamesCapstoneProject\W...
Older
ForceField.sln
C:\Users\Michael White\Documents\UnityProjects\ForceFieldProject...
ForceField.sln
C:\Users\Michael White\Documents\UnityProjects\ForceField
TestProject.sln
C:\Users\Michael White\Documents\UnityProjects\TestProject
Open
Checkout from:
Visual Studio Team Services
Open Project / Solution
Open Folder
New project
Search project templates
Recent project templates:
Create new project...
Developer News
More news...
Output
Show output from:
Solution Explorer
Class View
Team Explorer
Toolbox
Ready

Get code from a remote version control system or open something on your local drive.

The new project templates you use appear here. The list also roams with your personalization account.

Announcing new REST API's for Process Customization (Azure DevOps)
Last sprint we released a new set of REST API endpoints for process customization. In version 4.1 there are 3 sets of REST API's. Two for the inherited model and one for the Hosted XML model. This cre...
NEW Thursday, 20 September 2018

Xamarin.Forms 3.2.0 Released: Going Big with Little Things
Xamarin.Forms 3.2.0 continues the theme of previous releases to give you big wins by making little things much easier to do. The entire version 3 series has been about reducing the friction of doi...
NEW Thursday, 20 September 2018

The future of ASP.NET SignalR
In ASP.NET Core 2.1, we brought SignalR into the ASP.NET Core family. Many of our users have asked what this means for the previous version of SignalR: ASP.NET SignalR.
NEW Thursday, 20 September 2018

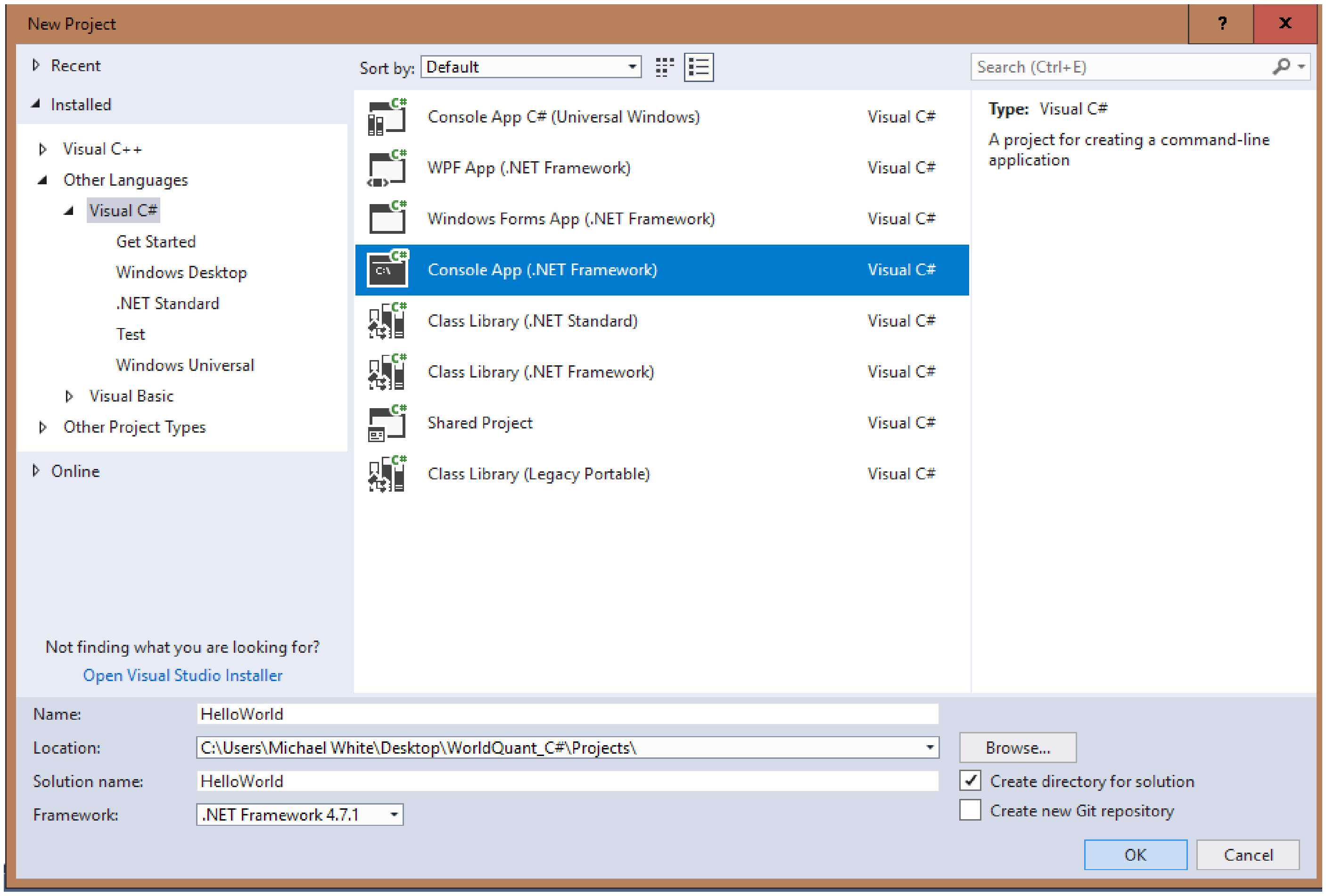
Deep dive into Azure Repos
With the introduction of Azure DevOps, we're offering developers a new Git service called Azure Repos. It has unlimited free private repositories with collaborative code reviews, advanced file mana...
NEW Thursday, 20 September 2018

Team Foundation Server 2018 Update 3 and Update 1.1 are available
Last week, we released both TFS 2018 Update 3 and TFS 2018 Update 1.1. Here are the key links:
NEW Tuesday, 18 September 2018

Announcing TypeScript 3.1 RC

Figure 2: Creating a new project - part 1

To create a new project, we click File -> New -> Project. At this stage we need to select the type of project to create. Search for "Console App (.NET Framework)" under the C# language templates, as shown in Figure 3. We now give our project a meaningful name, in this case "HelloWorld", and click "OK".

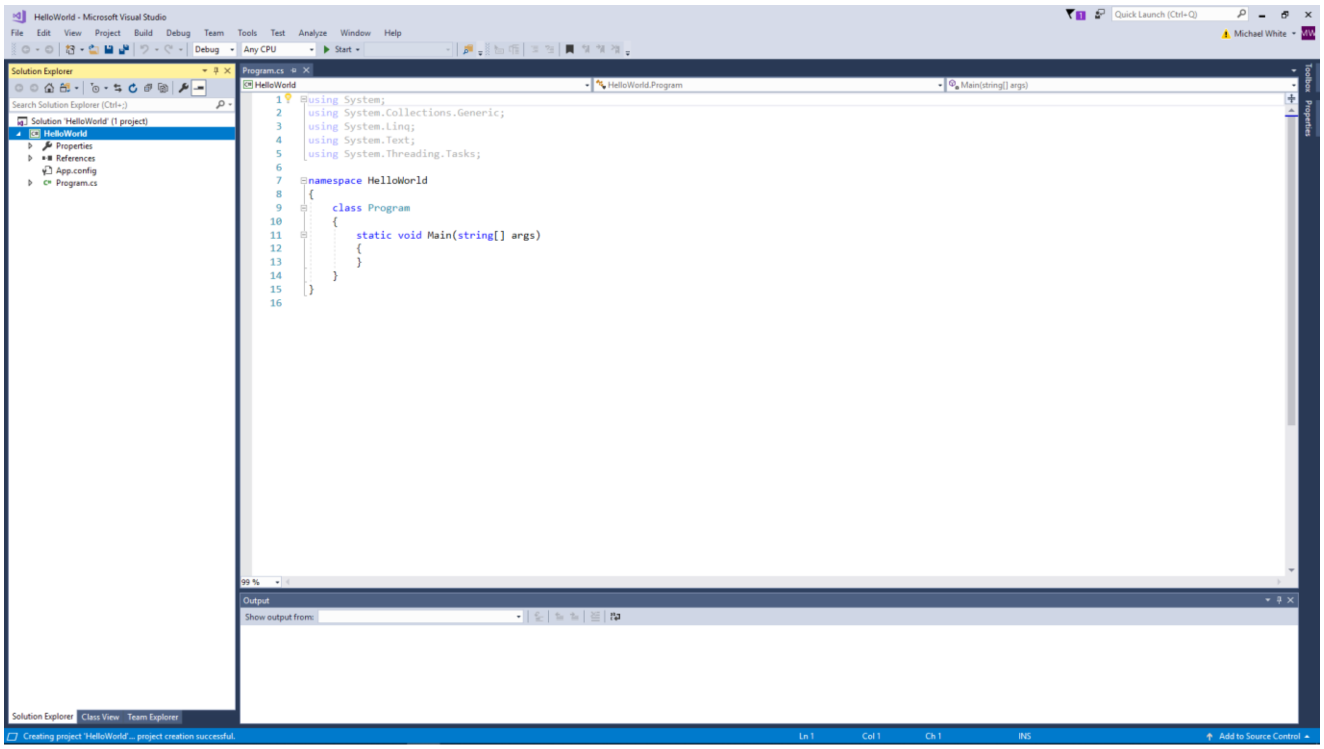

New Project
Recent
Installed
Visual C++
Other Languages
Visual C#
Get Started
Windows Desktop
.NET Standard
Test
Windows Universal
Visual Basic
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App C# (Universal Windows) Visual C#
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
Browse...
OK
Cancel

Type: Visual C#
A project for creating a command-line application

Name: HelloWorld
Location: C:\Users\Michael White\Desktop\WorldQuant_C#\Projects\
Solution name: HelloWorld
Framework: .NET Framework 4.7.1
[✓] Create directory for solution
[ ] Create new Git repository

Figure 3: Creating a new project - part 2

Visual Studio should now generate a new project, including a starting template within which we will add our code, similar to what is shown in Figure 4.



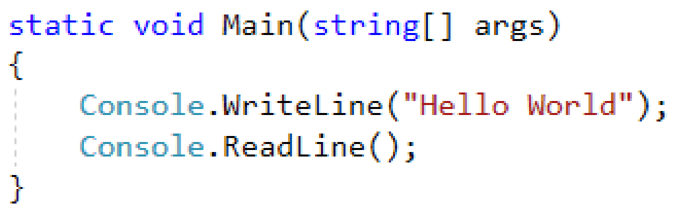
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

Creating project 'HelloWorld'... project creation successful.

Figure 4: The newly generated console application template

While this code may appear unfamiliar at this point, we will go over all the syntax in the upcoming sections. For now, it is just important to understand that in C# we place all our functional code inside of **classes**. These are blocks of code which form the basis of object-oriented programming and will be looked at in more detail later in this module. In order to execute a program, we need a **main class**, which is simply a class that contains a special, **main method**. This is the method that will be executed when we run our program. Essentially, it serves as an entry point to the program, and all program logic stems from the main method – either directly as part of the main method, or as part of another method which is called in the main method.

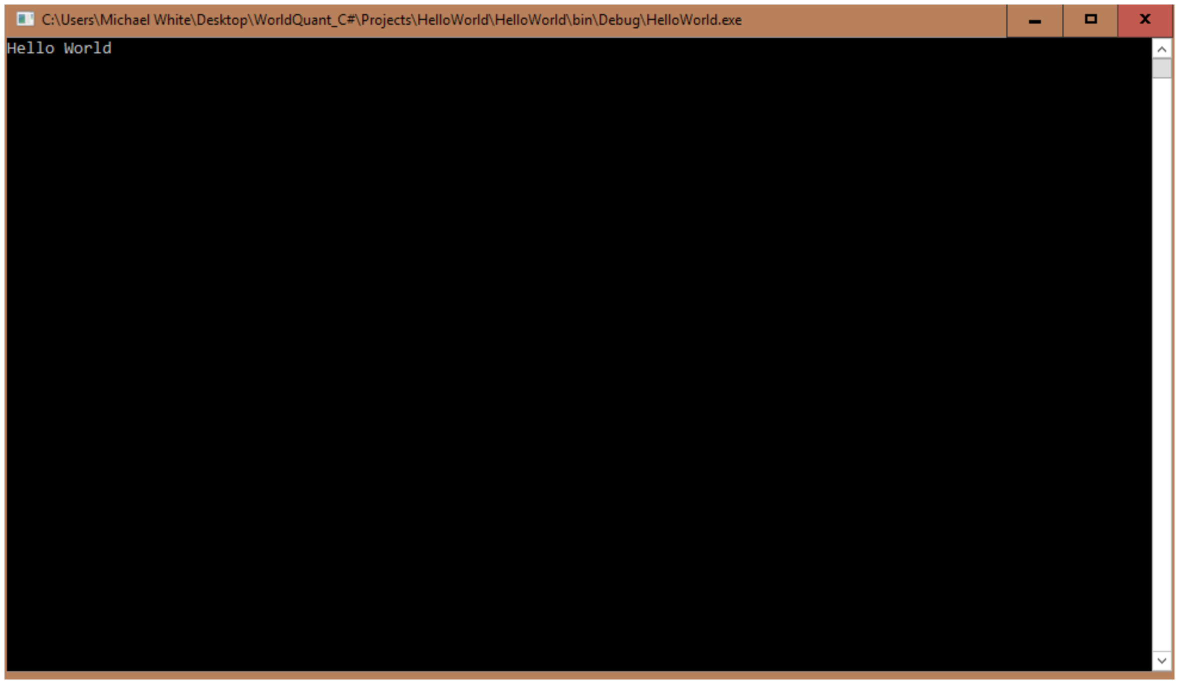
For the sake of this simple program, we will only add two lines, as depicted in Figure 5.



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

Figure 5: The complete code for our basic program

The first line runs the “Console.WriteLine” method in order to print out the phrase “Hello World”. Notice that in C#, we are required to end our lines of code with a semi colon (;). The second line runs the “Console.ReadLine” method, which simply waits for the user to press the enter key. We place this here to stop the console window from closing automatically at the end of our program’s execution. To run our program, we simply click the green triangle in the toolbar. At this point, the output shown in Figure 6 should become visible.


C:\Users\Michael White\Desktop\WorldQuant_C#\Projects\HelloWorld\HelloWorld\bin\Debug\HelloWorld.exe

```
Hello World
```

Figure 6: Output when the program is run

Data types and variables in C#

Unlike some other programming languages such as Python, C# is **explicitly typed**. This means that when we declare new variables, we have to specify the type of data which will be stored in that variable. For example, if we want to create a new integer variable with the identifier “x” and value “10“, rather than simply typing “x = 10; ”, we must type “int x = 10; ”.

It is possible to use the keyword “var” in place of a type identifier, which will cause C# to perform **type inference**. This means that the expression on the right side of the assignment operator, “= ", will be resolved, and C# will automatically use the type of this value to create the variable. For example, if we type "var x = ‘hello’;", C# will infer that the type is a string, whereas if we type “var x = 10; ", C# will infer that the type is an integer.

C# is also **statically typed** – in contrast to Python which is dynamically typed. This means that in C#, once a variable's type has been set, it cannot be changed. For example, if we declare a variable "x" to be of the string type, we cannot later set "x" to contain an integer value. Instead, we would have to use a completely different variable.

We will now briefly explore the various types afforded by the C# language. These fit into two broad categories:

1. Value types
2. Reference types.

Variables of **value types** directly contain their data, whereas variables of **reference types** store references to data that are stored elsewhere. This means that multiple reference type variables may reference the same underlying data. A change to one variable may therefore affect the other.

Value types are used for simple types, such as integers, characters, floating point numbers, and Boolean data. On the other hand, reference types are used for arrays and classes. Classes form the basis of the object-oriented programming model mentioned in the introduction to this section. **Classes** are complex types which can be defined by users. When these are **instantiated** – i.e. created and stored in variables – we refer to them as **objects**. It can thus be useful to think of classes as a sort of blueprint from which objects are constructed. The string built-in type is a class type.

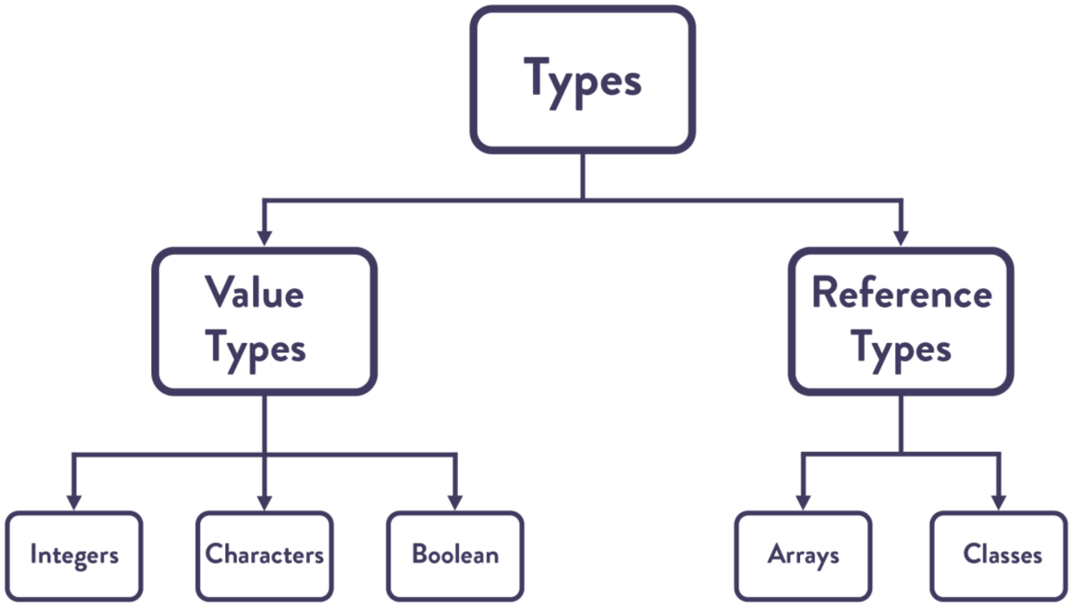


Figure 7: (Non-exhaustive) diagram of the C# type system

Value types

The following types of data are considered value types in C#. The type names used to represent them in code are placed in brackets.

- Integers [short, int, or long (in ascending order of the range of possible values which they can hold)]
- E.g. 17
- Floating point numbers [float or double (higher precision than float)]
- E.g. 1.32f
- High precision decimal numbers [decimal]
- E.g. 1.25
- Characters [char]
- E.g. 'C'
- Booleans [bool]
- E.g. true

Reference types

The following types of data are considered as reference types in C#. The type names used to represent them in code are placed in brackets.

- Strings [**string**]
- E.g. "Hello World"
- All class types [varies based on the particular class]
- Arrays [varies based on the type held within the array (This is covered in detail in the next section.)]
- E.g. { 1, 3, 5, 6, 8 }

Getting help

Many of the concepts that have been introduced in these notes (and indeed, many of those which will be covered later in this module) have very detailed guides to usage in the official C# documentation. This documentation can be accessed at https://docs.microsoft.com/en-us/dotnet/csharp/.

Further reading

What is .NET? - https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet

Summary

In this section, we went through the process of installing C# and Visual Studio on your own computer before taking an introductory look at the C# typing system and how this differs from that of a language like Python. We then covered the built-in data types afforded by C# and looked at how we can declare them in code.

In the next section, we will cover the built-in data structures of C# and see how we can use these to create powerful data-driven programs. We will also take a look at some of the other syntax of the C# language and see how we can use loops and conditional statements to generate more advanced functionality. Finally, we will explore how we can define our own functions.

Unit 2: Further Steps in C#

Introduction

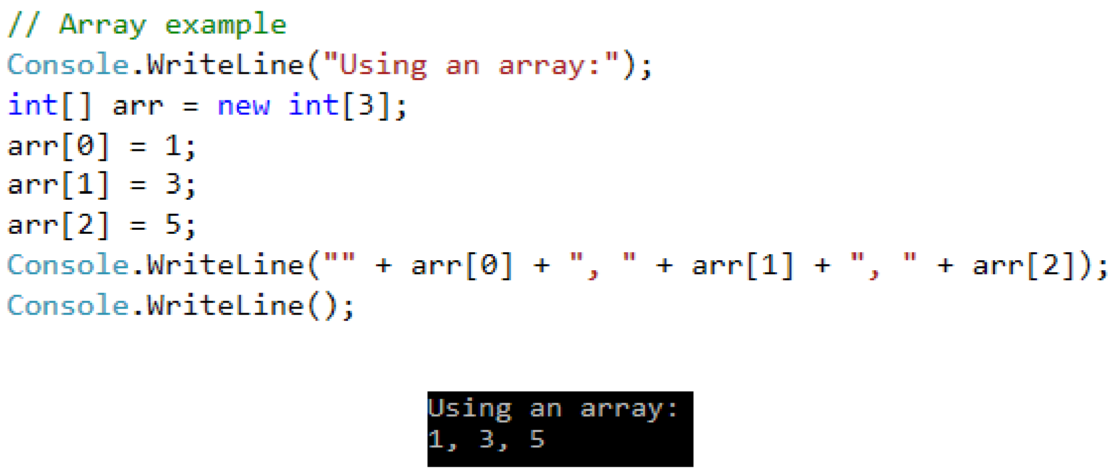
In order to work with large sets of data effectively in any programming language, it is necessary to organize this data into appropriate data structures. C# is no different and hence provides several useful data structures in its standard library. We will look at three of the most commonly used data structures in C#, namely:

1. Arrays
2. Lists
3. Dictionaries.

Arrays

Arrays are the most basic of data structures. For all intents and purposes, they are a group of values of a particular type stored in an indexed block of memory. In C#, arrays cannot be resized once created, which makes them less flexible than some other, more complex, data structures.

We declare an array variable using the name of the type of data that will be stored in the array, followed by square brackets. We initialize the array using the "new " keyword followed by the same type and square brackets, with the size of the array inside the brackets. We can access values in an array by typing its identifier followed by a set of square brackets with the index we want to access (see Figure 8).



```
// Array example
Console.WriteLine("Using an array:");
int[] arr = new int[3];
arr[0] = 1;
arr[1] = 3;
arr[2] = 5;
Console.WriteLine("" + arr[0] + ", " + arr[1] + ", " + arr[2]);
Console.WriteLine();
```

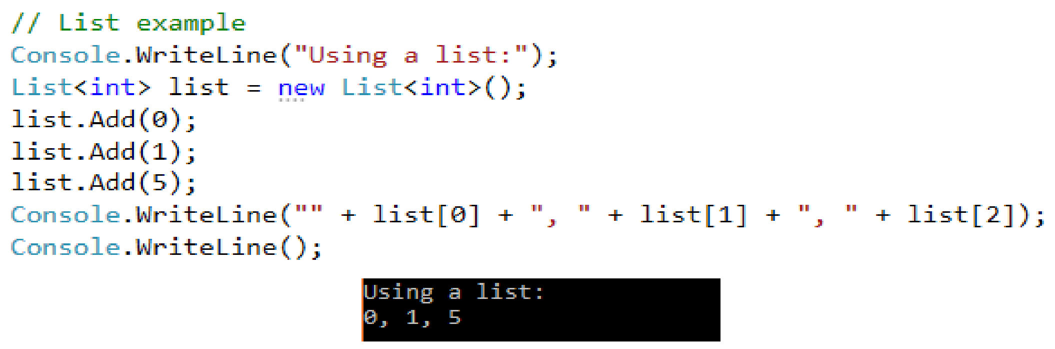
```
Using an array:
1, 3, 5
```

Figure 8: Example of array syntax usage

Lists

Sometimes the inflexibility of arrays makes them less suitable for a particular problem, such as when the size of a dataset is unknown at the start of processing. At these times, lists can often be useful. We can understand **lists** to be dynamically sized arrays that can grow or shrink as necessary. Like arrays, lists have a specified type upon creation and only values of this type can be added to the list.

We declare a list variable using the type name "List" followed by angled brackets containing the name of the type of data that will be stored in the list. We initialize the list using the "new" keyword, followed again by the List type name and the same angled brackets. Finally, we follow this with a set of rounded brackets. We append values to the end of the list using the "Add" method, and we access these values using the same, square-bracket, indexing syntax as we used for arrays.



```
// List example
Console.WriteLine("Using a list:");
List<int> list = new List<int>();
list.Add(0);
list.Add(1);
list.Add(5);
Console.WriteLine("" + list[0] + ", " + list[1] + ", " + list[2]);
Console.WriteLine();
```

```
Using a list:
0, 1, 5
```

Figure 9: Example of list syntax usage

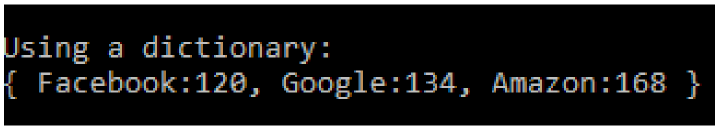
Dictionaries

There are times when it is helpful to reference our data not by numerical indices, but rather by some key, which can be of any data type. For example, if we have a collection of stocks, it might be useful to be able to reference each stock by the name of the company it relates to. In C# we can achieve this behavior using a dictionary. A **dictionary** is a data structure that stores data in key value pairs, allowing fast and intuitive retrieval of data. The C# dictionary operates similarly to Python's dictionary data structure. However, as with the other C# data structures we have looked at, the C# dictionary requires us to specify, at the time of its creation, the types of the keys and values which will be stored. These types do not necessarily have to match.

We declare a dictionary variable using the "Dictionary" type name followed by angled brackets containing the type names for both the key and value data we will be putting into the dictionary.

We initialize the dictionary using the “new” keyword much the same as we saw with the list example. We insert key value pairs into the dictionary using the “Add” method, and we access these values using the same, square-bracket, indexing syntax as we used for arrays.

```
// Dictionary example
Console.WriteLine("Using a dictionary:");
Dictionary<String, int> shares = new Dictionary<String, int>();
shares.Add("Facebook", 120);
shares.Add("Google", 134);
shares.Add("Amazon", 168);
Console.WriteLine("{ Facebook:" + shares["Facebook"] + ", Google:" +
                   shares["Google"] + ", Amazon:" + shares["Amazon"] + " }");
```



```
Using a dictionary:
{ Facebook:120, Google:134, Amazon:168 }
```

Figure 10: Example of dictionary syntax usage

Introducing repetition

There are two basic loop constructs in C# that we can use to create repetitive behavior:

1 For loops
2 While loops.

For both types of loop we must enclose their **bodies** – i.e. the code which will be run each iteration – in curly brackets – i.e. “{ " and “} "

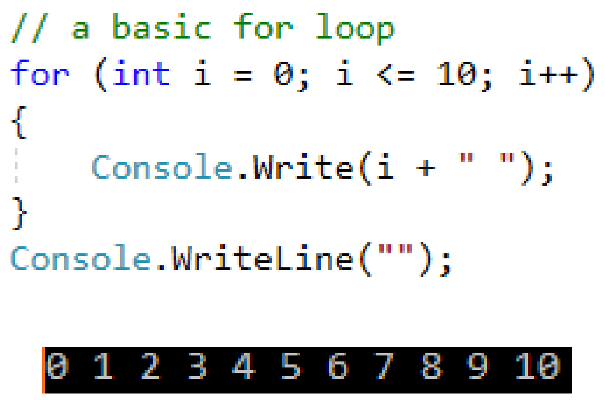
For loops

In cases where we know how many times we want to repeat a section of code, or we have a data structure and want to access every value, we generally use for loops. To declare this kind of loop, we use the “for“ keyword and then, within a set of rounded brackets, we initialize some variable to a starting value, provide a stopping condition, and finally declare the **step of the loop** – i.e. how much the loop variable changes every iteration. At the start of each iteration, the stopping condition is evaluated, and if it resolves to false, the loop will terminate. At the end of each iteration, the step is applied.

We can break down the example in Figure 11 as:

- **Initialization**: “int i = 0; ”

- **Stopping condition**: “i <= 10;“
- **Step**: “i++“ (This is simply shorthand syntax for adding 1 to the variable i)

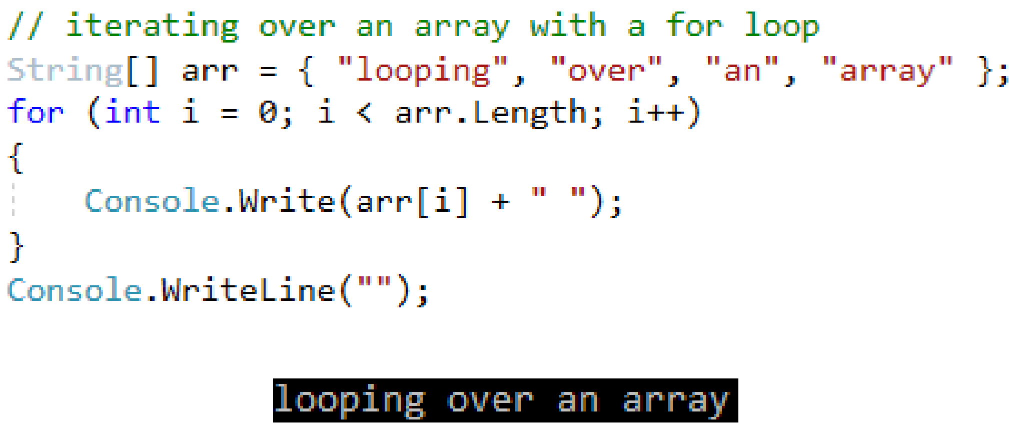


```
// a basic for loop
for (int i = 0; i <= 10; i++)
{
    Console.Write(i + " ");
}
Console.WriteLine("");
```

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

Figure 11: Example of basic for loop syntax in C#

We can iterate over an array by starting at index zero and increasing by one every iteration until we hit the length of the array (given by its “Length“ attribute, which we can access by typing “<array variable identifier>.Length“).



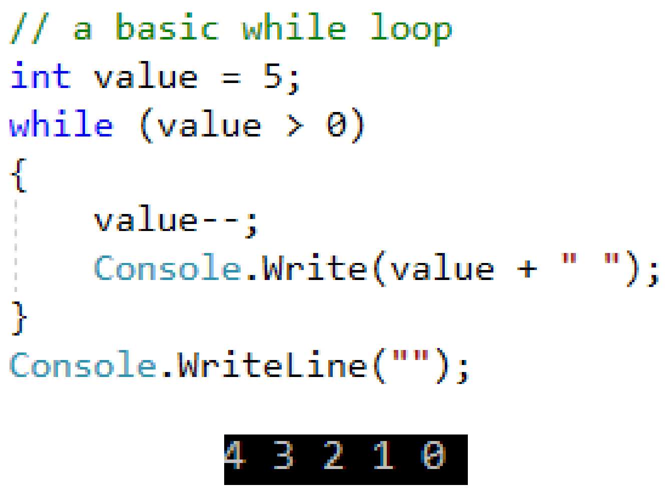
```
// iterating over an array with a for loop
String[] arr = { "looping", "over", "an", "array" };
for (int i = 0; i < arr.Length; i++)
{
    Console.Write(arr[i] + " ");
}
Console.WriteLine("");
```

```
looping over an array
```

Figure 12: Iterating over an array in C#

While loops

Sometimes we don’t want to iterate over a certain range of values. Rather, we might want to continue looping until a certain condition is no longer met. In these cases, we use a while loop. To declare this kind of loop, we use the “while” keyword then, in round brackets, provide some stopping condition that will be checked at the start of each iteration of the loop. If the stopping condition evaluates to false, the loop will terminate; otherwise it will proceed with the new iteration.



```
// a basic while loop
int value = 5;
while (value > 0)
{
    value--;
    Console.Write(value + " ");
}
Console.WriteLine("");
```

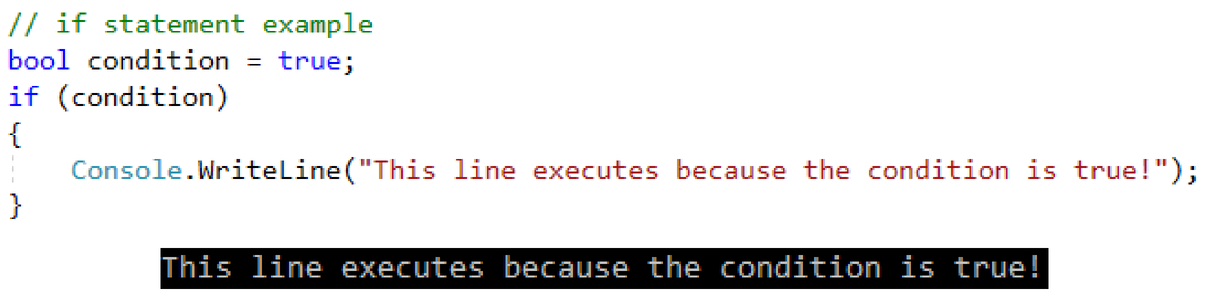
```
4 3 2 1 0
```

Figure 13: Example of basic while loop syntax

Making choices

We can introduce branching behavior into our programs using *if* and *else* statements. When using either of these statements, we must enclose their bodies in curly brackets, similar to how we constructed loops earlier.

When using an if statement, we must enclose some Boolean condition in rounded brackets before the body. This condition determines whether the code in the body will be executed. If the condition evaluates to true, the code is executed.

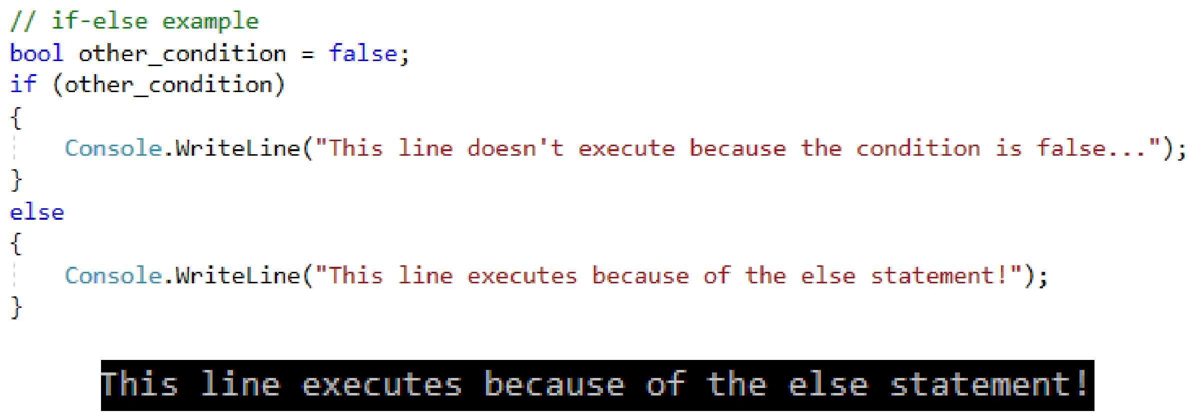


```
// if statement example
bool condition = true;
if (condition)
{
    Console.WriteLine("This line executes because the condition is true!");
}
```

```
This line executes because the condition is true!
```

Figure 14: Example of if statement syntax

Else statements must directly follow the closing bracket of an if statement's body. These do not have an additional condition to decide whether their body should be executed or not. Rather, we only execute their body if the preceding if statement's condition evaluated to false (and therefore its body was not executed).

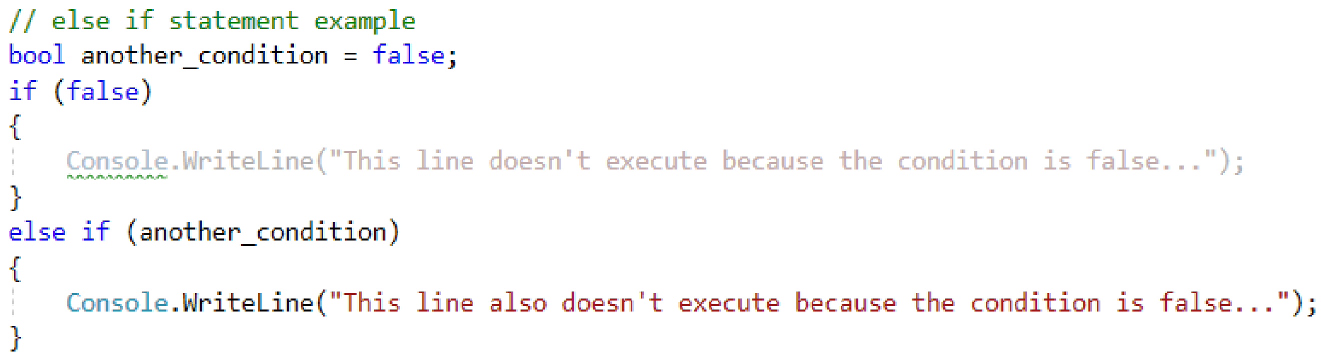


```
// if-else example
bool other_condition = false;
if (other_condition)
{
    Console.WriteLine("This line doesn't execute because the condition is false...");
}
else
{
    Console.WriteLine("This line executes because of the else statement!");
}
```

```
This line executes because of the else statement!
```

Figure 15: Example of the usage of if-else statements in C#

We can combine these statements into an *else if* statement. This acts like a normal else statement, except that it also has a condition that must be met in order for its body to be executed. Therefore, the preceding if statement's condition must have been false *and* the else if statement's condition must be true.



```
// else if statement example
bool another_condition = false;
if (false)
{
    Console.WriteLine("This line doesn't execute because the condition is false...");
}
else if (another_condition)
{
    Console.WriteLine("This line also doesn't execute because the condition is false...");
}
```

Figure 16: Example of the usage of else if statements in C#

Summary

In this section, we began by taking a look at the various built-in data structures provided by C# and how we can use these to meaningfully store sets of data. We then covered the syntax for adding repetition to our programs with for and while loops. Finally, we explored the use of if and else conditional statements to add branching behavior to our C# programs.

In the next section, we will cover the notion of object-oriented programming. We will start by exploring the theory behind this software development methodology to understand why it is useful to model programs in this way. We will then proceed to cover the notion of classes in detail before examining some of the core principles of object-oriented programming. We will finish off by learning to define our own methods in C#.

Unit 3: Object-oriented Programming in C#

Introduction

Object-oriented programming is a programming language paradigm that originated in academia in the late 1950s and early 1960s. By the 1990s, this paradigm had become dominant, and it is still largely dominant today. The core idea of object-oriented programming is to think of programs as a set of objects. Each **object** is simply an entity which may contain data and perform actions. To solve more complex problems, we then combine the work of many objects. The inherent advantage of this approach is that we can divide the code we need to solve any particular problem into many smaller subproblems, each of which is less complex to write and therefore less prone to error and easier to test for correctness.

To get an idea of object-orientation in practice, we can think of a kitchen (the overall program), which has a set of chefs (objects), who each have certain raw ingredients (data). Each chef is trained specifically to work with specific ingredients (and is unable to work with any other ingredients).

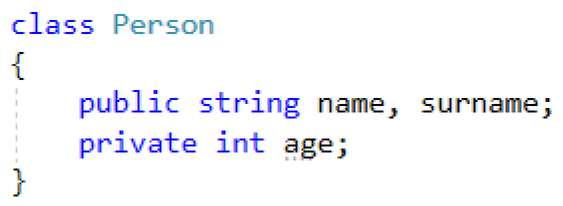
The goal of our hypothetical kitchen is to produce a cheese and tomato sandwich. So, we might have one chef who is trained to cut tomatoes and another chef who is trained to cut cheese. These chefs return the result of their work (the sliced tomatoes and cheese) to another chef who combines these ingredients with bread to form the final product (the sandwich). The important thing to note here is that while the first two chefs are trained specifically to prepare their respective raw ingredients, the processed ingredients which they provide might be used elsewhere (in this case, by the third chef).

By taking this approach, we divide up the work to make the process simpler for each individual object, thereby making the overall system easier to understand. We also make it easier to ensure our system works correctly. If we have trained a particular chef to do a particular job, and have ensured that they do this job correctly, then we can rest assured that any output they provide will be reliable anywhere it is needed.

Classes and objects

To create objects in practice, we need some way of defining what methods (made up of actions) and attributes (made up of data) will be associated with those objects. We also want to be able to create many objects of the same type. To allow for this behavior, we introduce **classes**, which are like blueprints from which we can create instances of objects. When we instantiate objects from classes, we get new objects which have their own versions of all the attributes defined in the class (which will be used if we call methods on that object which reference any attributes).

In C#, we organize all of our program code into classes. Each class is generally defined in a separate file. In fact, even in the very first example in this module, where we created a simple "Hello World" console application, we used a class (although this was generated for us by Visual Studio). Furthermore, any actions we want an object to be able to take must be organized into methods. We will look at how to define our own methods later. For now, let's look at a simple example of defining a class.



```
class Person
{
    public string name, surname;
    private int age;
}
```

Figure 17: Example of a simple class to store information about a person

We declare a class using the "class" keyword followed by the name of our class, "Person". We then declare some **attributes** (data which will be contained in instances of our class) just like we normally declare variables. Notice that the body of the class is enclosed in curly brackets.

This class definition tells us that each Person object has three attributes: their name, their surname, and their age. A Person object would not be able to perform any actions at this point because the Person class is made up of only attributes, with no methods. We will see how we can add methods to our classes later in this module, and how we can create objects from this class in the next set of notes.

Principles of OOP

We can talk about object-oriented programming in terms of four main principles, namely:

1. Inheritance
2. Encapsulation (and data hiding)
3. Abstraction
4. Polymorphism.

At present, we will examine each of these principles in theory. In the next set of notes, we will see how we can implement them in practice.

Inheritance

It is sometimes necessary to create two or more types of objects that are very closely related but with slightly different requirements in terms of the data they store or actions they can perform. In these situations, we introduce the idea of inheritance in the interest of maintaining both an easy to understand structure as well as avoiding having to copy code between different classes. **Inheritance** is a mechanism by which we can define classes to be "children" of other classes. A child inherits its parent's methods and attributes, but a child may also add extra attributes and methods or *override* existing ones, making them different to the way they are defined in the parent class.

Please note: sometimes we see child classes referred to as "subclasses" and parent classes referred to as "superclasses." These are different ways of talking about inheritance that mean the same thing.

The diagram below shows inheritance of a Student class from a Person class. It makes sense that Student would inherit from Person, because all students must be people, and therefore must have all the characteristics and abilities that people have.

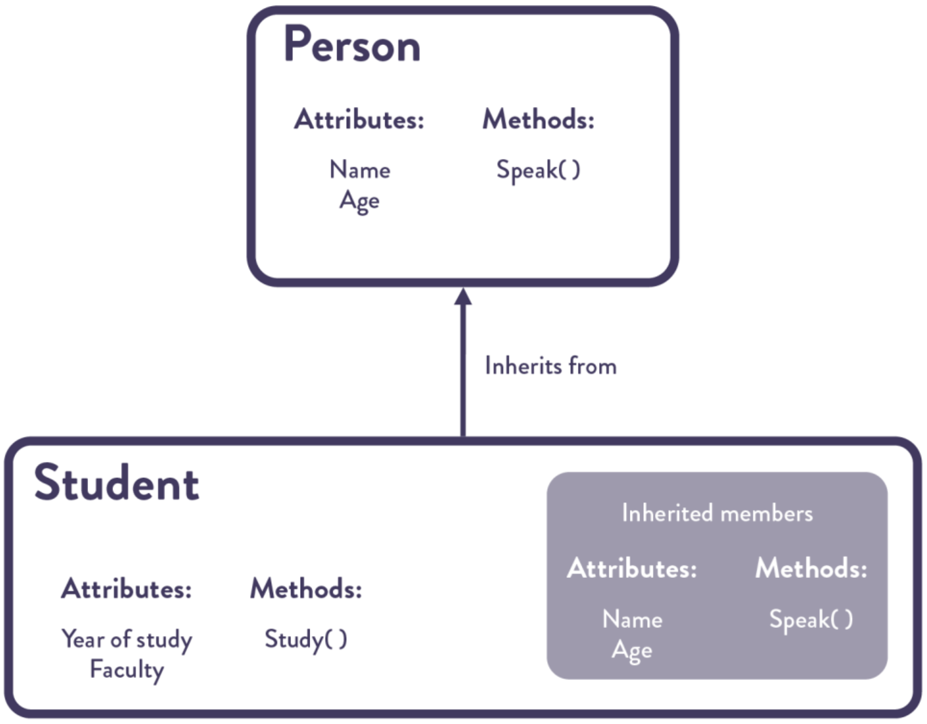


Figure 18: Diagram depicting inheritance

Encapsulation (and data hiding)

This principle relates to the way in which we design objects. Essentially, we want to firstly design objects in such a way that they have internal access to all the data necessary to perform their own actions, and, secondly, have this data protected from outside interference. This leads us to the idea of **data hiding**, meaning that an object may make its data inaccessible from outside that object. In C#, we can control this behavior using *access modifiers*, which we will look at in detail later.

The diagram below shows the way access modifiers allow us to hide members of a class from other classes:

- Members with the **private** access modifier are accessible only from within the class and cannot be accessed externally even in any subclasses.
- Members with the **protected** access modifier are accessible only from within the class and its subclasses, but not from unrelated classes.
- Members with the **public** access modifier are accessible anywhere.

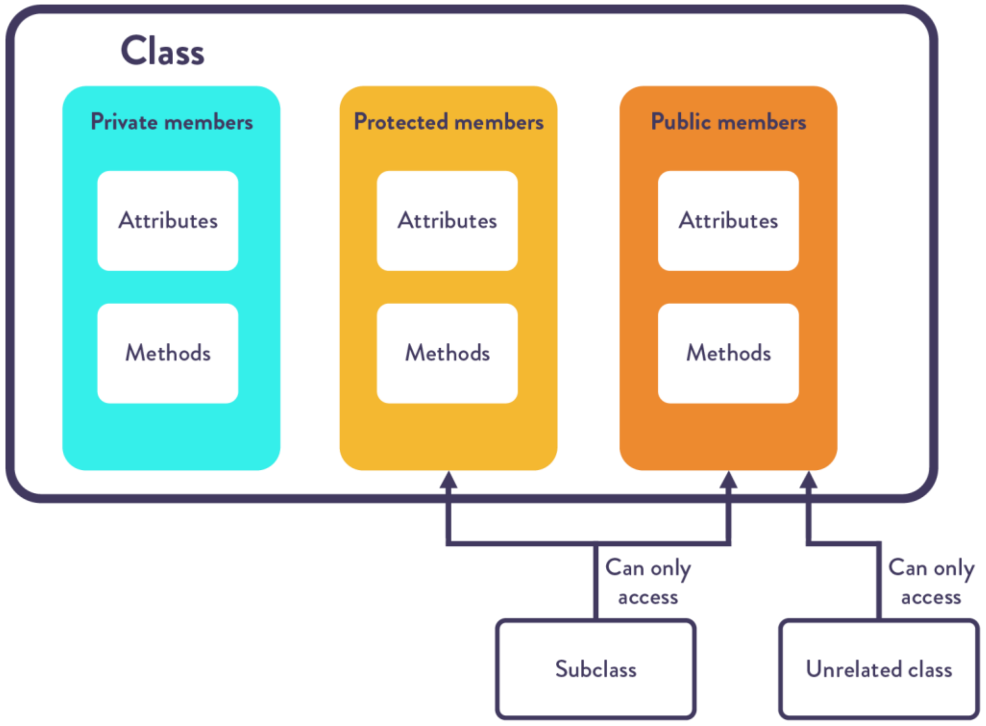


Figure 19: Diagram depicting the concept of data hiding

Abstraction

We want to make our systems easier to understand, and one way of doing this is to implement low level detail about how particular systems work in separate objects. Then, we can provide easy access to this functionality without requiring any underlying knowledge about how the system works. We only need to provide the necessary input to the system and know what output we expect from that system, without needing to understand how the underlying functionality of the system works.

Think of a production pipeline where the source material must be gathered by one expert and passed forward to a manufacturing expert to form the product, which must then be passed to a quality assurance expert to check for faults. No stage of the pipeline requires any knowledge of what the previous or future stages do, but, rather, each stage requires an understanding of what input they are supposed to receive and what output they are supposed to provide.

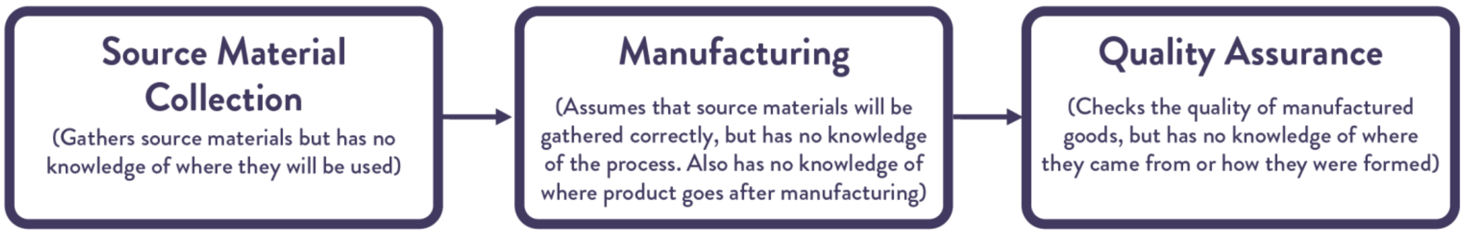

Source Material Collection
(Gathers source materials but has no knowledge of where they will be used)
Manufacturing
(Assumes that source materials will be gathered correctly, but has no knowledge of the process. Also has no knowledge of where product goes after manufacturing)
Quality Assurance
(Checks the quality of manufactured goods, but has no knowledge of where they came from or how they were formed)

Figure 20: Depiction of abstraction

Polymorphism

Polymorphism allows us to adapt the behavior of a method based on the context in which it is called. We can divide polymorphism into two behaviors in C#:

1 **Method overloading**. This is when we define several methods with the same identifier but different argument lists. C# will automatically detect which version of the function we are calling based on the arguments we provide. This is known as **static** or **compile-time polymorphism** because it occurs when we build our project, not when we run it.

2 **Method overriding**. This is when a method that exists in a parent class is redefined or overridden in a child class and C# needs to decide which method version it should run based on the run-time type of the object whose method is being called. This is known as **dynamic** or **run-time polymorphism**, because it occurs at run-time.

In the next set of notes, we will discuss polymorphism in more detail and see how it works in practice.

Defining our own methods

Methods are blocks of code that we create with the goal of completing a specific subtask. By using methods, we can break our code up into smaller, easier to understand parts which can then be combined to provide the functionality which is ultimately required.

Defining methods in C# is slightly more complex than in a language like Python, mainly because of C#'s strong typing system as well as its object-oriented nature. That being said, we can break method definitions into helpful divisions that will make them much easier to grasp. A method definition has five sections:

1. Access Modifier
2. Return type
3. Identifier
4. Arguments
5. Body.

Access Modifier

We mentioned access modifiers earlier when discussing the notion of data hiding in object-oriented programming. As explained then, access modifiers determine where methods and attributes owned by an object (collectively known as **members**) can be accessed from. The three most commonly used access modifiers are:

1. **Public**. The member is accessible from anywhere it is referenced.
2. **Private**. The member is only accessible by code in the same class.
3. **Protected**. The member is only accessible by code in the same class or a class that inherits from said class.

Return type

Often, though not always, our methods will compute a value of some type and return this to wherever the method was called. Because of the way C#'s typing system works, we must state in advance what the return type of any given method is. This uses the same type names as we have used in the past to declare variables. For example, if we want to compute and return an integer value in a method, then the return type would be "int".

In special cases where we have a method that does not return a value, we use the *void* type. This signifies that the method will not return any value. A common example of a void method is the *main method*, which acts as the starting point of any C# program. Because every other method execution stems from the main method, there would be nowhere for the return value to be sent at the end of its execution. Hence, we use the void type in this case.

Identifier

Similar to variables, every method needs a unique identifier by which it can be referenced.

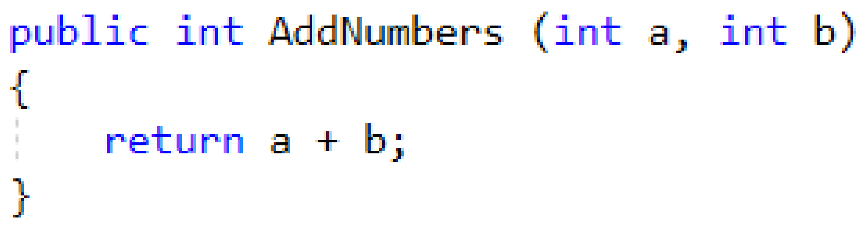
Arguments

Any information that needs to be passed into a method needs to be specified in the method declaration. Arguments are defined by a comma-separated list of standard variable declarations placed inside rounded brackets.

Body

The final section of a method definition is the actual body of the method. This is where we place the lines of code that will be executed when the method is called. In C#, we use curly brackets to indicate the start and end of a method's body.

Below, we have created a simple method that adds two integers "a" and "b" together, and returns the sum:



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

Figure 21: Example of a method

We can break this method down into its five constituent parts:

1. **Access Modifier.** In this case, the method is *public*, so it can be accessed from anywhere it is referenced.
2. **Return type.** The return type is *int*, since we want to return an *int* value.
3. **Identifier.** The identifier we use to call the method is "AddNumbers" because this is a good description of the class' purpose.

4. **Arguments.** We take in two *int* arguments, “a” and “b”.
5. **Body.** The body is everything enclosed between the curly brackets. In this case, this is just the one return statement, which returns the result of a + b.

Summary

In this section, we started by briefly looking at the history of object-oriented programming. We examined the defining characteristics of the paradigm and then learned how to create our own classes. Thereafter, we introduced four fundamental principles of object-oriented programming. Finally, we learned how to implement methods to provide objects with the ability to perform actions.

In the next section, we will explore some more object-oriented programming in practice. In particular, we will see how we can instantiate and use the classes we have defined before looking at how the four fundamental principles we explored earlier function in practice.

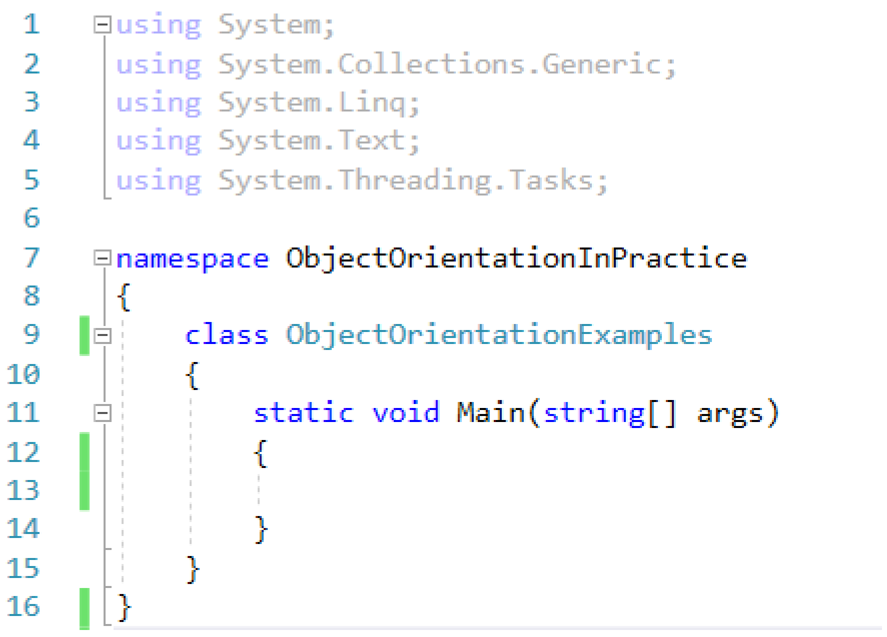
Unit 4: Programming in C#

Introduction

If we want to write code that is executable, we need some way to define where the "entry point" for the code is. That is, we need to define some place from which we will start executing code. In the case of C#, we define a method with the identifier "Main" to signify that it should be the starting point for execution. This method will always be static. Being **static** simply means that the method "belongs to the class" rather than belonging to an object. We call static methods by referencing the class rather than an instantiated object.

For more information on static members, see the official Microsoft Docs page, which provides a more in-depth explanation. A link to this page can be found in the further reading section below.

For now, we will create a main class that we will use to execute all the example code in this section. We will add code to this class as we move through this section.



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

namespace ObjectOrientationInPractice
{
    class ObjectOrientationExamples
    {
        static void Main(string[] args)
        {

        }
    }
}
```

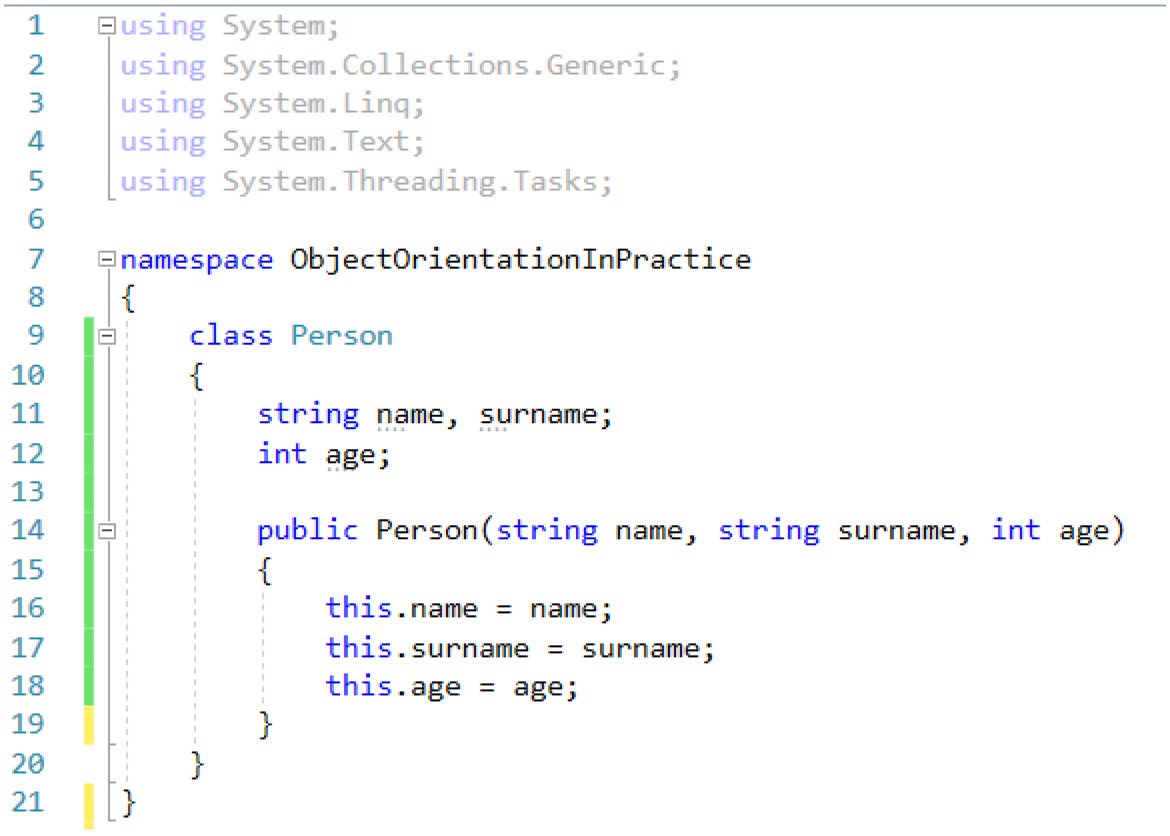
Figure 22: Skeleton main method example

You'll notice that the class we create is automatically wrapped in a "namespace" by Visual Studio. A simple way to think of a namespace is as a package of related code. Hence, we generally put all the classes for a program we are writing in the same namespace.

Constructors

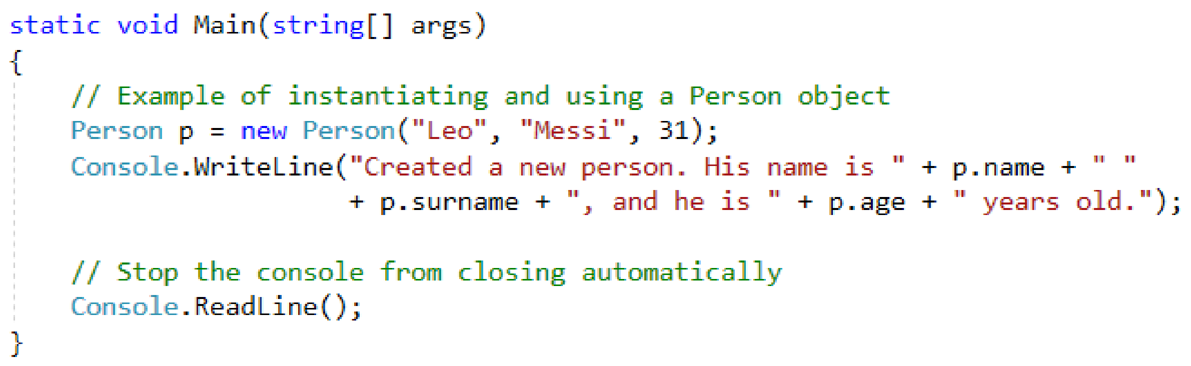
We saw in the previous section how to define our own classes, but we have not yet been able to use these classes to instantiate objects. For this purpose, we need constructors. **Constructors** are special methods defined in a class that return an instance of that class. They are defined slightly differently to other methods as they do not have an identifier. Rather, they simply have the type name of the class followed by the set of arguments.

Now, we will create a simple "Person" class and see how to instantiate Person objects. Notice the use of the "this" keyword to allow an object to reference itself in non-static methods.



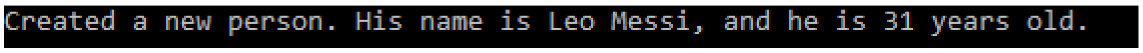
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ObjectOrientationInPractice
{
    class Person
    {
        string name, surname;
        int age;

        public Person(string name, string surname, int age)
        {
            this.name = name;
            this.surname = surname;
            this.age = age;
        }
    }
}
```

Figure 23: Example of a class definition with a constructor



```
static void Main(string[] args)
{
    // Example of instantiating and using a Person object
    Person p = new Person("Leo", "Messi", 31);
    Console.WriteLine("Created a new person. His name is " + p.name + " "
                        + p.surname + ", and he is " + p.age + " years old.");

    // Stop the console from closing automatically
    Console.ReadLine();
}
```

Figure 24: Example of instantiating an object with the "new" keyword



```
Created a new person. His name is Leo Messi, and he is 31 years old.
```

Figure 25: Output from our code when run

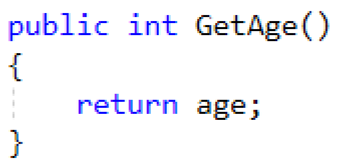
Data hiding in practice

Let's look at how data hiding and access modifiers work in practice. From inside the class, we will not notice any difference in behavior between access modifiers. From outside the class, however, access modifiers are key to determining whether members will be accessible or not.

For example, let's edit our Person class from earlier, changing the access modifier on the "age" variable to private. If we now try to run the program again, we will see an error when Visual Studio attempts to build the project:

error CS0122: 'Person.age' is inaccessible due to its protection level

Now, say we want to be able to access the person's age but not change it. We can define a public "GetAge" method that returns the person's age. This would make the person's age **immutable** (unable to be changed) but still accessible.



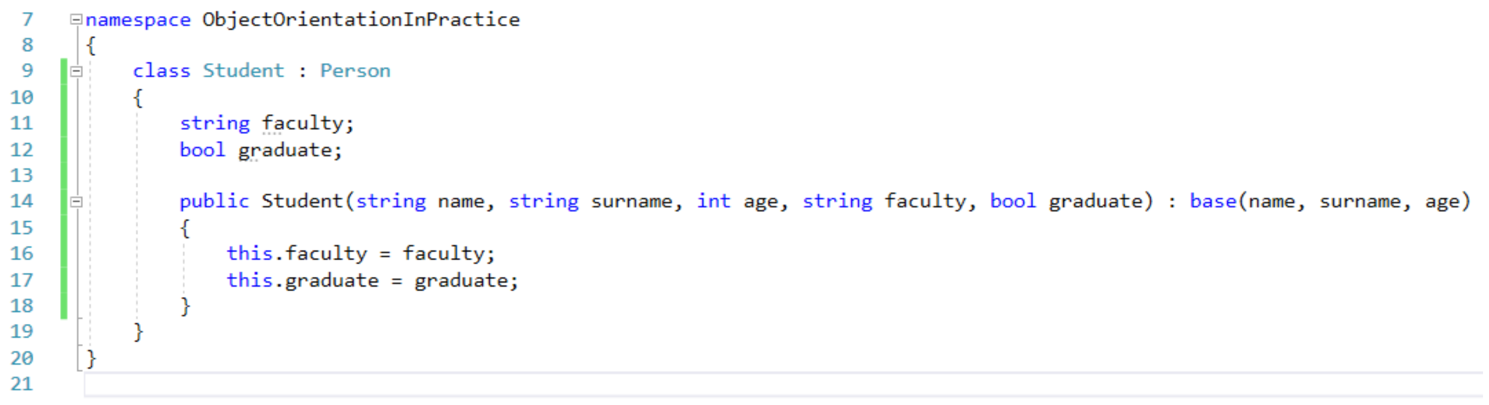
```
public int GetAge()
{
    return age;
}
```

Figure 26: Example of a 'getter'

Inheritance in practice

We will now demonstrate the usage of inheritance by creating a new class, “Student”, which inherits from our previous Person class. We can see why it makes sense for a student to inherit from a person, as all students under consideration must be people and should therefore have the properties and methods that people have. We will also add a few new properties to indicate the faculty of the student, and whether they are a graduate student or not.

To show that a class inherits from another, we put a colon after the class declaration followed by the name of the class that we want to inherit from. We may need to call the base class’s constructor from the subclass’s constructor (this will be necessary unless the base class has a no argument constructor, which will then be called implicitly). To do this, we put a colon after the argument list of the subclass constructor, followed by the keyword “*base”* and the argument list required for that constructor.



```
namespace ObjectOrientationInPractice
{
    class Student : Person
    {
        string faculty;
        bool graduate;

        public Student(string name, string surname, int age, string faculty, bool graduate) : base(name, surname, age)
        {
            this.faculty = faculty;
            this.graduate = graduate;
        }
    }
}
```

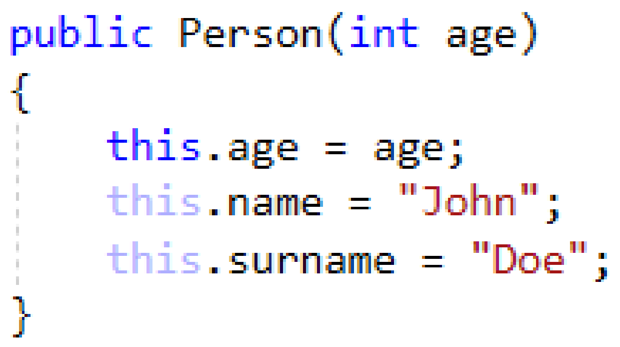
Figure 27: Example of inheritance

Polymorphism in practice

Method overloading

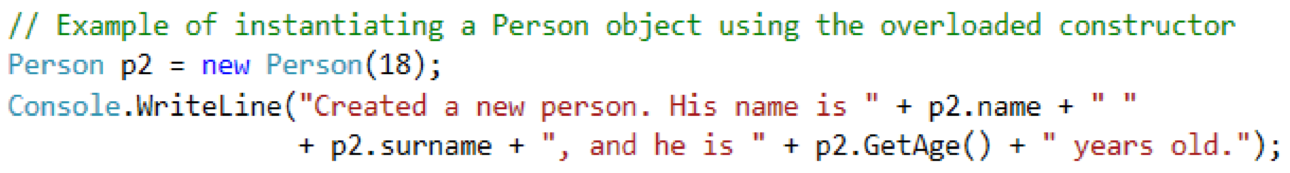
Recall from the previous notes that **method overloading** is when we define several methods with the same identifier but different argument lists. This can be done with any method, but perhaps the easiest way to demonstrate its usage, and one of the most common ways to use it, is to define multiple constructors.

See how we can define a constructor that takes only age as an argument and instantiates the name and surname to “John” and “Doe” respectively. C# will automatically deduce which version of the overloaded method to use based on the arguments we provide when calling that method.



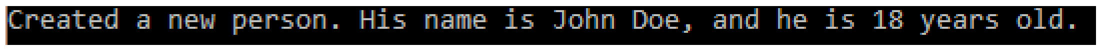
```
public Person(int age)
{
    this.age = age;
    this.name = "John";
    this.surname = "Doe";
}
```

Figure 28: Overloaded constructor



```
// Example of instantiating a Person object using the overloaded constructor
Person p2 = new Person(18);
Console.WriteLine("Created a new person. His name is " + p2.name + " "
                  + p2.surname + ", and he is " + p2.GetAge() + " years old.");
```

Figure 29: Instantiating a person using the overloaded constructor in the main method



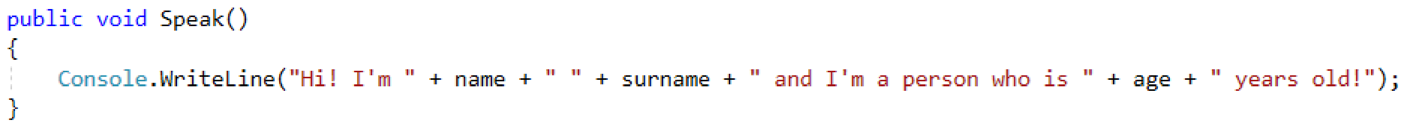
```
Created a new person. His name is John Doe, and he is 18 years old.
```

Figure 30: New output

Method overriding

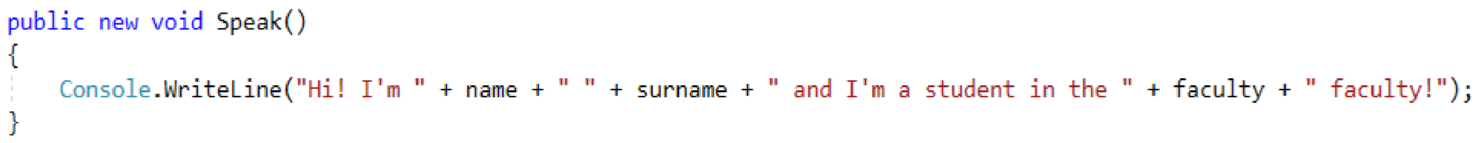
Recall from the previous notes that **method overriding** is when a method that exists in a base class is *redefined* or *overridden* in a subclass, and C# needs to decide which method version it should run based on the *run-time type* of the object whose method is being called. The run-time type of an object depends on the specific context in which that object is stored.

It is important to note that we can define a variable with the type of a base class but assign it a value which is an instance of a subclass. Similarly, we can define a collection with the base class type but store a mixture of objects instantiated from either the base class or a subclass. To demonstrate method overriding, we define a "Speak" method in the Person class that is overridden in the Student class.



```
public void Speak()
{
    Console.WriteLine("Hi! I'm " + name + " " + surname + " and I'm a person who is " + age + " years old!");
}
```

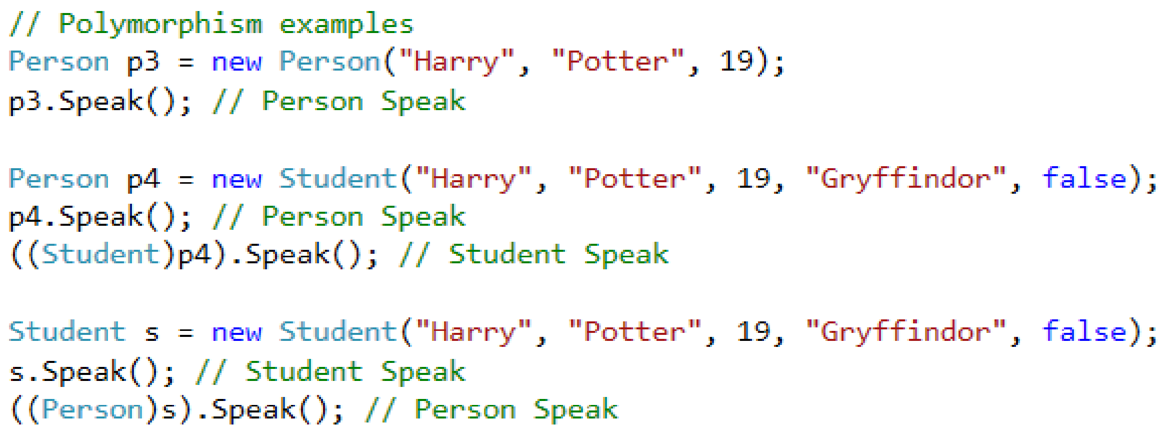
Figure 31: Person 'Speak' method



```
public new void Speak()
{
    Console.WriteLine("Hi! I'm " + name + " " + surname + " and I'm a student in the " + faculty + " faculty!");
}
```

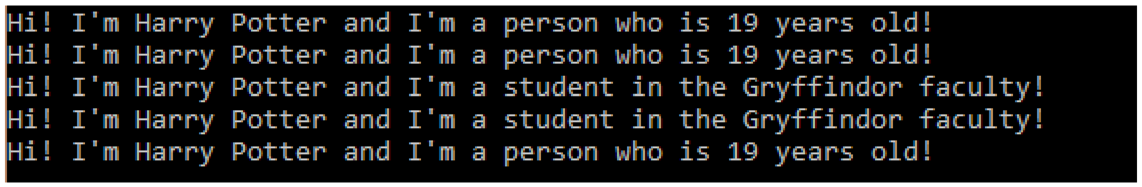
Figure 32: Student 'Speak' method

Now, let's see which version of the method is called in different contexts.



```
// Polymorphism examples
Person p3 = new Person("Harry", "Potter", 19);
p3.Speak(); // Person Speak

Person p4 = new Student("Harry", "Potter", 19, "Gryffindor", false);
p4.Speak(); // Person Speak
((Student)p4).Speak(); // Student Speak

Student s = new Student("Harry", "Potter", 19, "Gryffindor", false);
s.Speak(); // Student Speak
((Person)s).Speak(); // Person Speak
```

Figure 33: Overriding examples



```
Hi! I'm Harry Potter and I'm a person who is 19 years old!
Hi! I'm Harry Potter and I'm a person who is 19 years old!
Hi! I'm Harry Potter and I'm a student in the Gryffindor faculty!
Hi! I'm Harry Potter and I'm a student in the Gryffindor faculty!
Hi! I'm Harry Potter and I'm a person who is 19 years old!
```

Figure 34: Output

Summary

In this section, we started by learning to instantiate objects by using constructors. Thereafter, we learned to make executable programs by declaring a main method. We then looked at the practical use of data hiding, inheritance, and polymorphism.

This concludes the module, and you should now feel comfortable with the fundamentals of C# and be well equipped to explore more of the intricacies of the language. You should be able to create simple, object-oriented programs and define your own classes. If you are struggling with any aspect of the language, a key resource is the official documentation, which resides at https://docs.microsoft.com/en-us/dotnet/csharp/index.

Further reading

Microsoft Docs page on static members:

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members#static-members

Bibliography

Microsoft (2019) “Abstract and Sealed Classes and Class Members - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members

Microsoft (2019) “Abstract and Sealed Classes and Class Members - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/methods

Microsoft (2019) “Access Modifiers - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

Microsoft (2019) “Classes - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/classes

Microsoft (2019) “Collections and Data Structures | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/standard/collections/

Microsoft (2019) “Constructors - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/constructors

Microsoft (2019) “Inheritance - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/inheritance

Microsoft (2019) “Learn conditional logic with branch and loop statements | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/tutorials/intro-to-csharp/branches-and-loops-local

Microsoft (2019) “Methods - C# Programming Guide | Microsoft Docs” [Online] Available at:https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members

Microsoft (2019) “Polymorphism - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/classes-and-structs/polymorphism

Microsoft (2019) “Types - C# Programming Guide | Microsoft Docs” [Online] Available at: https://docs.microsoft.com/en-za/dotnet/csharp/programming-guide/types/

Collaborative Review Task

In this module, you are required to complete a collaborative review task, which is designed to test your ability to apply and analyze the knowledge you have learned during the week.

Overview

For this assignment, you will be required to generate a Visual Studio C# command line project (in the same way we have built programs previously in this module). You will need to use the C# syntax learned throughout the module along with your understanding of object-oriented programming principles to finish the task.

Programming task

You will be required to create a simple payroll program that can handle numerous different types of employees. A skeleton C# project is provided with a "PayrollRunner" class that should *not* be edited at all. This class provides a main method that you will be able to use to run your program once you have created all the required functionality.

Section 1: Create "Employee" class

To create a new class, right click on the project in the "Solution Explorer" section and select "New -> Class".

You will need to create various methods and attributes, but bear in mind that we want to adhere to the data hiding principle. All your attributes should therefore be private. The method signatures that you will need to provide *publicly* are given below. You should create attributes as needed but be sure to give them appropriate names.

Methods:

1. *Constructor*

   Employee (int employeeId, String fullName, float salary, bool taxDeducted)
2. *Overloaded Constructor – If taxDeducted is not specified, assume it will be true with this overload constructor method.*

   Employee (int employeeId, String fullName, float salary)

3 *getNetSalary – This method should return the Employee's salary minus 20% tax only if applicable.*

float getNetSalary( )

4 *printInformation – This method should not return anything but rather print out the Employee's information in the format:*

"<employeeId>, <fullName> earns <net salary> per month."

void printInformation( )

Section 2: Create 'WeeklyEmployee' subclass, inheriting from Employee.

1 *Constructor – Remember that this should call a base class constructor.*

WeeklyEmployee (int employeeId, String fullName, float salary, bool taxDeducted)

2 *Overloaded Constructor – Remember that this should call a base class constructor.*

WeeklyEmployee (int employeeId, String fullName, float salary)

3 *getNetSalary – This method should return the Employee's weekly salary minus 20% tax only if applicable. Remember that the salary given in the constructor is a monthly amount. This method should override the method in the base class.*

float getNetSalary( )

4 *printInformation – This method should not return anything but rather print out the Employee's information in the format:*

"<employeeId>, <fullName> earns <netWeeklySalary> per week."

void printInformation( )