

Exercise 1: The Product Application

1. Objectives

This exercise intended to help you get started with programming in C#, by a step-by-step guide to writing a simple console application. In addition, we exercise with writing code using a simple text editor. We compile and run the program through the Command Prompt Window (also known as Console). Follow the instructions and try to understand every step.

2. Where to begin?

To write a C# application, you need to install either Visual Studio (VS) or .NET Framework SDK on your computer. Assuming that you have Visual Studio installed on your computer, the next step is to learn the language basics by writing code and programming as much as you can.

A C# program starts with a class that has a method Main. A method is a block of cod containing instructions for the computer to do some work and carry out a task whenever it is called. Methods together solve a problem.

The Main method is a method that serves as the main entry to the program. This method is executed automatically when the program is started by the operating system. The Main method has a special construction that has to be that way in all applications. Remember that every application only can have one class containing the Main method.

We begin with a very small C# program, consisting of only one class, with the **Main** method containing only one line of code. Create a folder **ProductApp** somewhere on your hard disk. Then, create a text file using Windows Notepad, or any other text editor (not word processors such as MS Word), copy the following code and save the text file as **Program.cs** in the mentioned folder

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Testing my first C# program!");
    }
}
```

Open the Developer Command Prompt that follows the installation of Visual Studio (VS). In this Command Prompt, all the environment variables are correctly set by VS, and it knows where the C# compiler and the runtime tools are located on your computer. Before you can compile the **Program.cs**-file from the command prompt, you must change directory to the folder where you have saved your code file. Then, you can compile the file by using the C# compiler **csc.exe**, as shown in the figure below.

```

C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise>cd C:\DotNet\Module1\Exercise1\ProductApp

C:\DotNet\Module1\Exercise1\ProductApp>csc program.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20251.5 (910223b6)
Copyright (C) Microsoft Corporation. All rights reserved.

C:\DotNet\Module1\Exercise1\ProductApp>program.exe
Testing my first C# program!

C:\DotNet\Module1\Exercise1\ProductApp>
  
```

The screenshot shows a Windows command prompt window titled "Developer Command Prompt for VS 2019". It displays the output of the Visual Studio 2019 Developer Command Prompt v16.6.2, including the copyright notice for Microsoft Corporation. The user has navigated to the directory C:\DotNet\Module1\Exercise1\ProductApp and executed the command csc program.cs, which compiled the program successfully. The user then executed the command program.exe, which ran the program and displayed the output "Testing my first C# program!". Annotations in the image include a blue box around the directory path, a blue box around the csc command, and a green box with the text "Run the program" and "Output (fr Main)" pointing to the program.exe command and its output.

By writing **csc Program.cs**, the code is compiled and a runnable file, **Program.exe**, is generated by the compiler, provided that your code does not contain any error. To run the application , write **product.exe** (simply **program**) at the prompt and press Enter. The above figure shows the following steps:

1. Change directory to the folder where the code file is saved.
2. Compile the **Program.cs** to produce an exe-file.
3. Run the **Exe** file.

The Product class

Now, suppose that we would like to write a program that stores data about products. The data, which we are interested to store for a product, are:

- name (a text),
- price (a value that may have decimals), and
- quantity (a whole number, i.e. an integer) of that particular product.

These make input to the program. The above values are given when the program runs (run-time) which means that we don't know them at the time of writing code (compile-time). Therefore, we need to use variables.

We can think of a scenario where the user is to be prompted to receive these values, the application should count the total to pay for the product (depending on the value of the quantity), and display the amount for the user. These are the operations required to produce a useful program.

The data and operations that manipulate the data are very closely related. Together they represent a group of objects, i.e. any types of products, fruits, food items, electronics, etc., as long as they can be defined with the mentioned attributes and operations. **To represent a group of objects, we write a class.** Think of a class as a blank form. A copy of the blank form is filled with current values for each product. Our virtual form is very intelligent as it also has the capability of doing things by itself.

To solve this particular problem, we write a class in which we define the above data and write methods to receive input from the user, process the data (perform operations) and display the desired output. For each product, we can create an object of the class and save the products data. Each object will then have its own values (name, price and quantity) but will be able to do the same type operations on the data.

In most cases, there are three main operations to perform:

- Get user input, save in instance variables.
- Calculate or process data
- Display, or make available output

Create a new text file, and save it as **Product.cs** in the same directory as the **Program.cs**. Write the following code in this file and save it again.

```
public class Product
{
    //input variables to store values given by the user
    private string name; //name of the product as text
    private double price; // price as reel number (with or without decimals)
    private int count; // quantity of the product as whole number.

}
```

Variable names (**name**, **price**, **count** in above) are chosen by you with the following rules:

- Cannot contain spaces and symbols (+, -, etc.) other than “_” (underscore),
- Cannot begin with a number but can contain numbers.
- Cannot be a word that is reserved by C# (do, if, else, while, etc.).

The “;” at the end of each instruction (statement) is a part of the C# language rules (syntax). We need to have the data as **private** in this class. This is in accordance to one of the main principals of object-orientation. The double slashes, “//”, mark the beginning of a line comment. Comments are important to make notes, for ourselves and for others who review the code, but are not part of the code and are not compiled. You will learn a lot more about these issues in the coming modules.

The next step is to program instructions so the class performs its operations and accomplish its tasks. All operations (tasks) must be done inside what is known as methods. For each operation, write a separate method. Every method has a definition and a body. Let us begin with reading the user input.

Read Input: This task can be broken down to three tasks:

- Read name
- Read price
- Read count

Methods make a much larger topic, but we begin with a type of methods that is called **void**, without parameters. In all our programming, we write methods and therefore we will be learning to use other forms of methods as we move on.

A void method begins with the reserved word (keyword) **void**, followed by a name (with same rules as variable names), a pair of parentheses and then a body that is enclosed inside a pair of curly braces { }. Everything we write inside the braces belong to the method. All methods in a class have access to the variables of the class (variables outside the methods), called fields, attributes, or instance variables as well as other methods inside that class.

```
public void MethodName()
{
    //your code
}
```

The keyword "public" is used to make a method accessible for methods of other classes. Methods used internally in a class can be made **private**. Remember, instance variables should always be **private**. Methods can be either **private** or **public**.

3. The C# Console class

To interact with the user, C# provides a useful class called **System.Console**. This class has methods that we can use to interact with the user through the Command Prompt. A list of some useful methods of the **Console** class, which you can use in your own methods, are outlined in the table below.

Note: To skip writing "**System.**", add the following line at the top of the code file (before the class definition:

```
using System;
```

The above statement tells the compiler to import all classes, packaged in the **System** namespace, which comes with C#. A namespace is a logical grouping of classes.

Console.Title	Changes the title of the Command Prompt window	Console.Title = "My First C# Program";
Console.Write()	Writes a message to Command Prompt leaving the cursor	Console.Write("Hi there!");

	at the end of the message, on the same line	
<code>Console.WriteLine()</code>	Functions as <code>Console.Write</code> but moves the cursor to the beginning of the next line.	<code>Console.WriteLine("Hi there!");</code> <code>Console.WriteLine(); //black line</code>
<code>Console.ReadLine()</code>	Returns a line of text from the cursor positon to the end of the line.	<code>string name = Console.ReadLine();</code>
<code>Console. BackgroundColor</code>	Changes the background color of the Command Prompt window.	<code>Console.BackgroundColor= ConsoleColor.Blue;</code>
<code>Console. ForegroundColor</code>	Changes the text color of the Command Prompt window.	<code>Console.ForegroundColor= ConsoleColor.White;</code>
<code>Console.Clear()</code>	Clears the content of the Command Prompt Window. Call this after changing the background and/or foreground colors.	<code>Console.Title = "My first C# Program";</code> <code>Console.BackgroundColor = ConsoleColor.Blue;</code> <code>Console.ForegroundColor =</code> <code>ConsoleColor.White;</code> <code>Console.Clear();</code>

The Console class also has a method **Console.Read()**, and another one **Console.ReadKey()**. As these are not working with strings, they are more complicated to use. Using the **ReadLine** method is much easier and sufficient for most of the purposes.

Let us write the methods in several steps. We write code for each method, compile it, run and test it, and if the method functions as expected, we move to the next method repeating the same procedure.

4. Exercise 1A: Read name of the product

Prompt the user to give the name of the product; then save the input in the instance variable **name**.

```
using System;

class Product
{
    //input variables
    private string name; //To store name of the product as text
    private double price; //To store price as reel number (with or without decimals)
    private int count; //To store the quantity of the product as whole number.

    //Read the name of the product from the Prompt Window
    public void ReadName()
    {
        Console.WriteLine("Name of the product?"); //Writes the text to the Console window
        name = Console.ReadLine(); //returns a line of text from the Console window
        Console.WriteLine("The product's name is: " + name);
    }
}
```

The occurrences of the variable **name** are marked in red. You should notice that some of the texts in above code are placed inside quotation marks. These are constant strings. The variable **name** is not inside any quotation marks because it is a variable. In the first occurrence, it is declared as **string** telling the compiler that this variable will be storing a text, i.e. a sequence of characters. The modifier **private** tells the compiler that the variable is not accessible outside the class **Product**.

The variable **name** is given a value to store inside the **ReadName** method:

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

Remember that **name** is a string and so is the value returned by **Console.ReadLine()**. They are of the same type and therefore the assignment statement is correct, i.e. the leftside and the right side of the assignment operator "=" have the same type.

To test the above method, we need to modify the **Main** method in the **Program.cs** class. A new lesson to remember is that a class gets access to another class' methods by creating an object of the other class, using the reserved word "new", as in the code below. It is like taking one copy of the class and using it by filling data in its variables, and calling its methods to perform tasks.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.Title = "My first C# Program";
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.ForegroundColor = ConsoleColor.White;
        Console.Clear();

        Product product = new Product(); //Create an object of the Product class
        product.ReadName(); //call (activate) a method of the object

        Console.ReadLine(); //Put the Prompt Window in a input mode so it does not close
    }
}
```

Note: C# is case-sensitive; the words "**Product**" (with capital P) and "**product**" (with the lower-case p) are two different words. **Product** is the name of the class from which we create an object. We give the object the name **product** (with small p). We could choose any other word instead (following the naming rules explained earlier) such p, x, and productObj, but it is a part of a good programming style to use variable and method names that make their purpose and usage clear. Short names like p, x make your code less readable, and should be avoided.

Let us now compile and run the program. Go to the same directory as before and compile all cs files using the following command.

```
csc *.cs
```



```

C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise>cd C:\DotNet\Module1\Exercise1\ProductApp

C:\DotNet\Module1\Exercise1\ProductApp>dir
Volume in drive C is OSDisk
Volume Serial Number is 220C-C8DC

Directory of C:\DotNet\Module1\Exercise1\ProductApp

2020-07-02  01:41    <DIR>          .
2020-07-02  01:41    <DIR>          ..
2020-07-02  00:36                602 Product.cs
2020-07-02  00:36                437 Program.cs
                2 File(s)              1 039 bytes
                2 Dir(s)  33 134 338 048 bytes free

C:\DotNet\Module1\Exercise1\ProductApp>csc *.cs
Microsoft (R) Visual C# Compiler version 3.6.0-4.20251.5 (910223b6)
Copyright (C) Microsoft Corporation. All rights reserved.

Product.cs(10,18): warning CS0169: The field 'Product.price' is never used
Product.cs(11,15): warning CS0169: The field 'Product.count' is never used

C:\DotNet\Module1\Exercise1\ProductApp>dir
Volume in drive C is OSDisk
Volume Serial Number is 220C-C8DC

Directory of C:\DotNet\Module1\Exercise1\ProductApp

2020-07-02  01:42    <DIR>          .
2020-07-02  01:42    <DIR>          ..
2020-07-02  00:36                602 Product.cs
2020-07-02  00:36                437 Program.cs
2020-07-02  01:42                4 096 Program.exe
                3 File(s)              5 135 bytes
                2 Dir(s)  33 134 268 416 bytes free

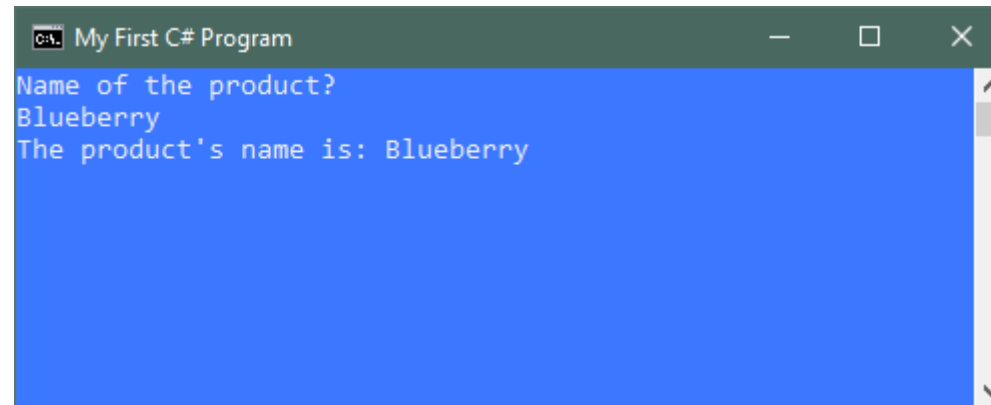
C:\DotNet\Module1\Exercise1\ProductApp>

```

before compilation

after compilation

The compiler gives no error but two warnings, which we don't have to worry about as we are going to use the variables mentioned in the warning texts. Now we can test the program by writing **program.exe** (or just **program**) and pressing Enter.



```
C:\> My First C# Program
Name of the product?
Blueberry
The product's name is: Blueberry
```

Here we go! Now let's move on to the next step and prompt the user for the price of the product.

5. Exercise 1B: Read the price of the product

The difference is now that we are expecting a number from the user, but the number is transferred as text (sequence of characters) from the keyboard to our application. The user must give the number in a valid format, e.g. 4, -1.0, 5E-10 (5 to the power of -10). The user is responsible for using the correct decimal sign as set in the operating system (Windows Regional Settings). If the input contains invalid characters, the program will crash when it tries to convert the text to a number.

Write a method to read the price from the user. Follow the same pattern as above. We name the method as **ReadPrice** (no spaces) and write the necessary instructions inside the method. Write the following code in your Product class. As you can see in the code, when the user feeds in the price, the **ReadLine** method transfers the line as a text containing only characters. The text has to represent a number. We have to convert the text to its corresponding number. We are going to save the price in a variable of a double type. The user can give a number with a fractional part or a whole number, but it will be treated as floating-point number.

```
//Ask the user to feed in the price of the product
public void ReadPrice()
{
    Console.WriteLine("Price of the product?"); //Prompt the user for price

    //User's input is a text (ex "34.5")
    string strPrice = Console.ReadLine();

    //convert the text to its corresponding number ex. "34.5" (4 chars) --> 34.5 (number)
    price = double.Parse(strPrice);

    //Confirm the price.
    //ToString converts the price to text: ex 34.5 (number) --> "34.5" (chars)
    Console.WriteLine("The product's price is: " + price.ToString());
}
```

The code lines that are marked in red are new in this method. The first line reads the user input and saves it in a temporary variable (**strPrice**). This variable is of the type **string** because **ReadLine** returns a **string**. The variable is called a "local" variable and it is not accessible outside this method. It cannot be declared as private or public as it is private inside its method.

The text, **strPrice**, is to be converted to a double (floating-point number) using the **double.Parse** method. This method expects a **string** that is made of characters that are digits and symbols that are used in a number (0 to 9, +, -, the decimal sign, etc.). If the text contains invalid characters, for instance a letter like x, a, o, l, etc., the method fails to do its job causing the program to terminate abnormally.

Note that when we need to show any number on the screen for the user using **Console.WriteLine** (or **Write**), we have to convert the number to its corresponding textual format. This is because **WriteLine** works with textual data (as the **ReadLine** method). However, **WriteLine** is very intelligent and it can often do the required conversion by itself when the code is being compiled. The code below will also work, but it is always good to work with correct data types.

```
Console.WriteLine("price" + price);
```

You cannot use "price" + price in other statements, as we do here; we are adding a string with a number (apples + pears). What happens in the above statement is that WriteLine method converts the variable price (double) to its corresponding textual representation.

Don't forget to save the file. Make this a habit to save your code very often.

To make use of the method, we need to call it from the **Main** method. Go to the **Program.cs** file and make the changes as shown below:

```
class Program
{
    static void Main(string[] args)
    {
        Console.Title = "First C# Program";
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.ForegroundColor = ConsoleColor.White;
        Console.Clear();

        Product product = new Product(); //Create an object of the Product class
        product.ReadName(); //call (activate) a method of the object

        Console.WriteLine(); //write a blank line on the screen
        product.ReadPrice();

        Console.WriteLine("Press Enter to exit!");
        Console.ReadLine(); //Put the Prompt Window in a input mode so it does not close
    }
}
```

Save this file too. Go to the Command Prompt window and compiles the files:

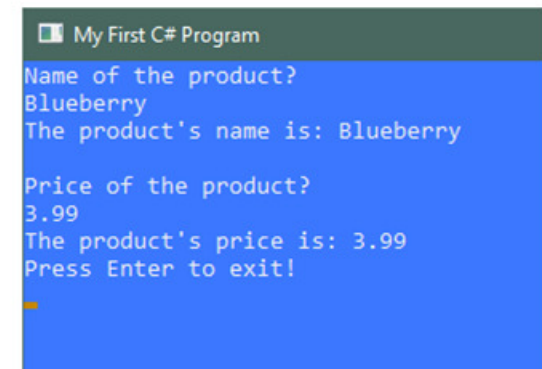
```
csc *.cs <Enter>
```

Run the program:

```
program <Enter>
```

You should then be able to run an example as demonstrated in the figure:

One more method left, **ReadCount** to let the user give the quantity of the product for which we now have a name and a price. To exercise with whole numbers, we have declared a variable, for it which is an integer type (int).



6. Exercise 1C: Read the quantity of the product

Proceed exactly as the previous exercise (reading the price), copy the code below to your Product class, call the method from the Main method in Program.cs, compile and run the program. Note that the difference is that we now expect a whole number from the user and use **int.Parse** to convert the input to an integer number: It should be pointed out that If the user gives a number with decimals, the number may be lose it decimal part or the program may terminate abnormally.

```
//Ask the user to feed in the quantity of the product as a whole number
public void ReadCount()
{
    Console.WriteLine("Quantity of this product (a whole number)?");

    //User's input is a text (ex "34")
    string strCount = Console.ReadLine();

    //Convert the text to its corresponding number ex. "34" (2 chars) --> 34 (number 34)
    count = int.Parse(strCount);

    //Confirm the input. ToString converts the count to text: ex 34 (number) --> "34" (chars)
    Console.WriteLine("The no of this product (units): " + count.ToString());
}
```

Call the above method from the Main method. Add these two lines (after the call to ReadPrice):

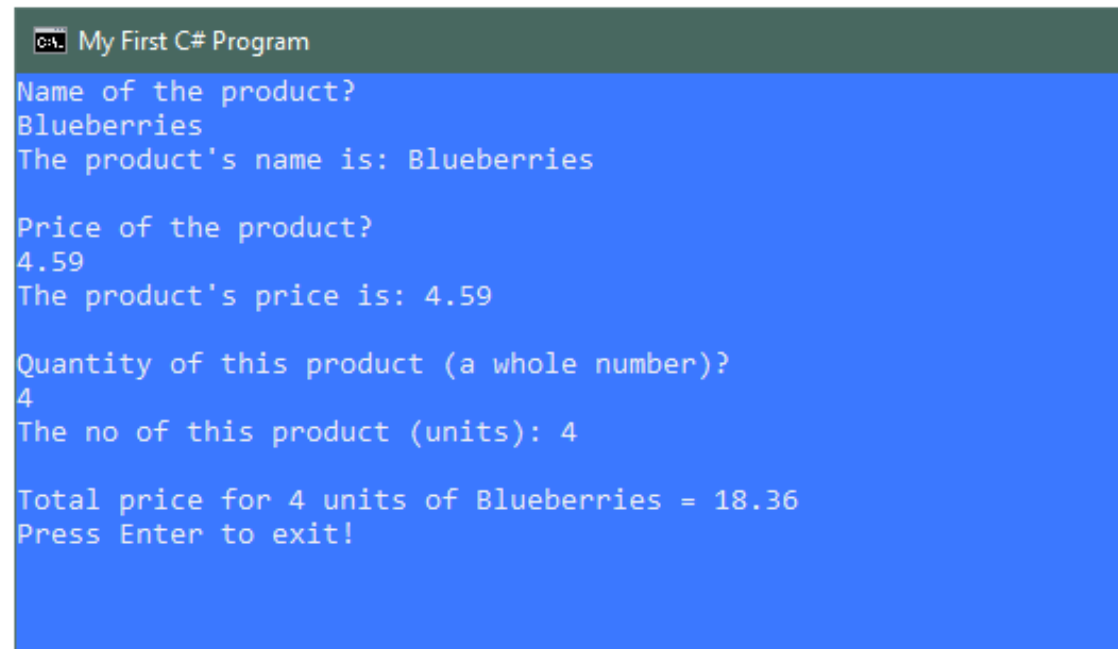
```
Console.WriteLine(); //write a blank line on the screen
product.ReadCount();
```

Save your code files. Compile and run as in the previous case. Now that we have all the input values, we can calculate the total price and display the result to the user. For this purpose, we write another method, as show in the code below:

7. Exercise 1D: Calculate the total price of the product

The total price is calculated by multiplying count and price. The following method carries out the calculation and displays the result. Call this method from the Main method, after the call to **ReadCount**

```
//Calculate the total price = count * price. Display the result to the user
public void CalculateTotalPrice()
{
    double totalPrice = count * price;
    //display the result to the user; put a space between two words
    Console.WriteLine("Total price for " + count.ToString() + " units of " + name + " = " +
        totalPrice);
}
```



```
C:\> My First C# Program
Name of the product?
Blueberries
The product's name is: Blueberries

Price of the product?
4.59
The product's price is: 4.59

Quantity of this product (a whole number)?
4
The no of this product (units): 4

Total price for 4 units of Blueberries = 18.36
Press Enter to exit!
```

The two code files are available for downloading in the module.

Remarks:

This exercise is intended to help you create a small application and learn to use methods for performing tasks. The same result could be achieved by much fewer lines using only the Main method, but that absolutely is not a good way of structuring and solving problems. It is important to do things in the right way without regard to the size of the application.

However, in the methods used in this exercise, we are mixing input, output and calculation, which is a simplification to avoid making the solution complicated. In the future modules, we will be learning to separate these concerns.

Remember that each class should have one responsibility, and handle one type of objects. In our exercise, we use a class to handle products that have a name, a price and a quantity. Each class should have a method for every task. Finally, all operations must be done inside methods. Only declaration of variables can be written outside the methods.