Practical Task 2.1

(Pass Task)

Submission deadline: 10:00am Monday, April 6
Discussion deadline: 10:00am Saturday, April 11

General Instructions

This practical task introduces you to ***classes*** and ***objects***; this is a fundamental feature of the object-oriented paradigm. You should consider your problems as you solve them on paper and aim to identify what objects you will need to program the solution efficiently. This will involve looking at

- the ***states***, i.e. ***instance variables***, which will allow you to classify and identify each instance of that object, and
- the ***behaviours***, i.e. the ***methods***, which will allow you to apply functions to the object.

Object orientation allows code reuse, modularity and portability of your programs. This should make testing and debugging much easier.

The tasks in the first part of this practical provide step-by-step instructions on creating an object in C#. This is followed by a number of tasks for you to complete by applying the knowledge gained in the first few tasks.

1. You go into a mobile phone shop to buy a mobile phone – what options do you have?
   - Do you want to "Pay As You Go (PAYG)" or "Pay Monthly"?
   - What is the type of phone that you wish to purchase?

   Let's say you decide on PAYG - how is the phone then unique to you?

   - You are allocated a phone number.

   How is the account activated?

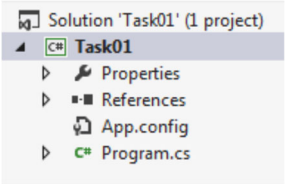
   - The account balance is set to zero.

   What functions can be carried out from an account perspective? You can

   - top up account with credit;
   - make a call, which reduces credit;
   - send a text, which reduces credit too;
   - get balance.

   There are other functions that you could apply, but let's start coding this part of the solution.

   STEP 1: First, create a new C# Console Application project. Name the main class as `MobileProgram` instead of the default class name `Program`. Add a new class and name it as `Mobile`. In Visual Studio this can be done from the Solution Explorer on the right hand side of the screen, where you need to right click on the project name and select Add → Class as shown below.



STEP 2: Add the following code into the `Mobile` class.

```
class Mobile {
    private String accType, device, number;
    private double balance;

    public Mobile(String accType, String device, String number)
    {
        this.accType = accType;
        this.device = device;
        this.number = number;
        this.balance = 0.0;
    }
}
```

The object constructor

This constructor sets up how the object will initially be created. For the `Mobile` object it accepts:

- The account type (either PayAsYouGo or Monthly);
- The name of the chosen device (e.g. iPhone S6);
- A mobile number as a String.

It then sets each of the values into variables.

As you may have noticed the variables are set to ***private*** and are therefore only accessible between the opening curly brace of class `Mobile` and the closing brace of this class. We normally consider these variables to be ***instance variables*** as they will help us to define the instance of an object (in this case a mobile) to enable us to make it ***unique***.

We have also introduced the use of ***this***. The '**this**' keyword refers to the ***current instance*** of the class (i.e. in this case the `Mobile` object and more specifically the instance variables).

This is the `Mobile` class file – or the ***object blueprint***. To test it, we will set up a new class. But as mentioned, we then cannot access the instance variables. To do this we will set up ***accessor*** and ***mutator*** methods to allow us to access and manipulate the instance variables. Making the instance variables ***private***, or ***hiding*** them, and keeping them together in a class with related methods is called ***encapsulation***.

Accessor methods allow us to access the information that is held in an instance variable and ***mutator methods*** allow us to change that information. We normally have an accessor and/or mutator method for **each** private instance variable.

STEP 3: Let's set up our accessor and mutator methods. Take a new line after the closing curly brace of the `Mobile` constructor (but before the closing curly brace of the `Mobile` class definition) and type the following code:

```
//Accessor Methods
public String getAccType()
{
    return this.accType;
}

public String getDevice()
{
    return this.device;
}

public String getNumber()
{
    return this.number;
}

public String getBalance()
{
    //returns the balance as a currency through the
    //ToString method and the parameter "C"
    return this.balance.ToString("C");
}
```

Note that `ToString("C")` returns a string that is in currency format, e.g. prints out the double value to two decimal place along with a '$' sign.

STEP 4: Now, navigate back to the `MobileProgram` class, which contains our main method. Add the following code inside the braces of the `Main` method:

```
Mobile jimMobile = new Mobile("Monthly", "Samsung Galaxy S6", "07712223344");

Console.WriteLine("Account Type: " + jimMobile.getAccType() + "\nMobile Number: "
    + jimMobile.getNumber() + "\nDevice: " + jimMobile.getDevice() + "\nBalance: "
    + jimMobile.getBalance());
Console.ReadLine();
```

Compile and run the program. Check whether the produced output is as expected.

STEP 5: Go back to the `Mobile` class and add the following mutator methods.

```
//Mutator Methods
public void setAccType(String accType)
{
    this.accType = accType;
}

public void setDevice(String device)
{
    this.device = device;
}

public void setNumber(String number)
{
    this.number = number;
}

public void setBalance(double balance)
{
    this.balance = balance;
}
```

Again, let's test these methods to ensure that they do as we expect, i.e. change the values currently stored in the instance variables. To test them, return to the `MobileProgram` and alter the `Main` method by adding the extra code as follows:

```
jimMobile.setAccType("PAYG");
jimMobile.setDevice("iPhone 6S");
jimMobile.setNumber("07713334466");
jimMobile.setBalance(15.50);

Console.WriteLine();
Console.WriteLine("Account Type: " + jimMobile.getAccType() + "\nMobile Number: "
   + jimMobile.getNumber() + "\nDevice: " + jimMobile.getDevice() + "\nBalance: "
   + jimMobile.getBalance());

Console.ReadLine();
```

Compile the program and run it to check if the produced output is as expected.

STEP 6: Now, let's add some more functions, that we noted at the start of this practical, such as:

- `AddCredit()` to top up account with credit;
- `MakeCall()` that reduces credit;
- `SendText()` that also reduces credit.

Select the `Mobile` class and add the following code:

```
//Methods
public void addCredit(double amount)
{
    this.balance += amount;
    Console.WriteLine("Credit added successfully. New balance " + getBalance());
}

public void makeCall(int minutes)
{
    double cost = minutes * CALL_COST;
    this.balance -= cost;
    Console.WriteLine("Call made. New balance " + getBalance());
}

public void sendText(int numTexts)
{
    double cost = numTexts * TEXT_COST;
    this.balance -= cost;
    Console.WriteLine("Text Sent. New balance " + getBalance());
}
```

You will notice that we use two variables here CALL_COST and TEXT_COST these will need to be declared and initialised at the top of the class with the instance variables. These variables will not change during the operation of the program. Therefore we must set them to be ***constants*** using the *'**const**'* keyword. To do this, type the following code:

```
//Instance Variables
private String accType, device, number;
private double balance;

//VARIABLES
private const double CALL_COST = 0.245;
private const double TEXT_COST = 0.078;
```

Navigate to the `MobileProgram` tab and add the following code before the `Console.ReadLine` statement to test the functions we have just implemented.

```
Console.WriteLine();
jimMobile.addCredit(10.0);
jimMobile.makeCall(5);
jimMobile.sendText(2);

Console.ReadLine();
```

Compile the program, run and test it again.

2. In this part of the practical, you will add more functionality to the `Mobile` class and also introduce a few more tests to ensure that what you are coding in the class actually executes as you think it should.

   In the `MobileProgram` class, add lines of code to create another instance of the `Mobile` object. You should think about this as a new Mobile account. Note that you will only be editing the `MobileProgram` class; the `Mobile` class does not need to be amended for this part of the practical.

   For the new `Mobile` user add credit, make a call (remember to add minutes) and finally send one or more text messages from this account.

3. Create a new C# Console Application project and add a class called `Employee`. This class should have two instance variables: a name (of type String) and a salary (of type double). Implement the constructor with two parameters that will set up the initial values of the instance variables:
   - `public Employee (string employeeName, double currentSalary)`

   You must implement the following methods:
   - `getName` – this is an accessor method that should return the name of the employee
   - `getSalary` – this is an accessor method that should return the salary of the employee in currency format
   - `raiseSalary` – this method should raise the employee's salary by a certain percentage

Test this class through the main `EmployeeProgram` class, which should replace (rename) the default class name `Program`.

Compile and run the program. If it performs as expected, then extend the object's functionality. To do this, navigate to the `Employee` class and add a new method `Tax()`, which calculates how much tax is deducted from the `Employee's` annual pay, according to the following criteria:

$0 – $18,200	0%	Nil
$18,201 – $37,000	19%	19c for each $1 over $18,200
$37,001 – $90,000	32.5%	$3,572 plus 32.5% of amounts over $37,000
$90,001 – $180,000	37%	$20,797 plus 37% of amounts over $90,000
$180,000 and over	45%	$54,096 plus 45% of amounts over $180,000

Use ***if…else*** statements. Make sure that the final program works correctly and meets the specification by testing it with different salaries in the `EmployeeProgram` class.

4. Create a new C# Console Application project and add a class called `Car` with the following parameters, which you will set in the constructor:
   - fuel efficiency which is measured in miles per gallon (mpg);
   - fuel in the tank (in litres) with an initial value of 0;
   - total miles driven with an initial value of 0.

   Set up a constant to hold the current cost of fuel in dollars per litre. For the purpose of this program, the cost per litre of petrol is 1.385$.

   You must implement the following methods in the Car class:
   - `getFuel` – this accessor method should return the amount of fuel in the tank
   - `getTotalMiles` – this accessor method should return the total miles the car has driven
   - `setTotalMiles` – this mutator method should update the total miles the car has driven
   - `printFuelCost` – a method that returns the cost of fuel in dollars and cents.
   - `addFuel` – this method should add fuel to the tank and calculate the cost of the fill
   - `calcCost` – this method will take the amount of fuel, in litres, and multiply it by the cost of fuel (in dollars per litre)
   - `convertToLitres` - this method should accept a parameter for the number of gallons, convert that value to litres and return the value in litres. This can be done by multiplying the value by 4.546.
   - `drive` – this method should simulate driving the car for a certain distance in miles. It will accept the number of miles driven and update the total miles stored for the car. Then you will need to calculate the fuel used by dividing the distance driven by mpg. This will return the number of gallons used, which you then need to convert into litres. The method should display a message to state the total cost of the journey, in dollars and cents.

   For the purposes of this class, you can assume that the `drive()` method is never called with a distance that consumes more than the available fuel.

   Change the default `Program` main class name to `CarProgram`. Write appropriate code in the `CarProgram` class to test all of the methods that you have implemented. Compile, run and check the program for potential errors.

Further Notes

- Study the way to program and use classes and objects in C# by reading Sections 2.3-2.6 of SIT232 Workbook available in CloudDeakin → Resources.
- The following links will give you more insights on classes/objects and relevant topics:
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/object-oriented-programming
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/members

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/objects

  Because this information is quite detailed and addresses some advanced concepts in OOP, you should mainly focus on the parts that have been mentioned in the first part of this practical. You may skim through the rest of the material to outline OOP as a field of study and get a rough understanding of other notions.

- The outline of how to debug your C# program code is given in Section 2.10 of SIT232 Workbook. More details are available online. For example, explore
  - https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-debugger?view=vs-2019
  - https://docs.microsoft.com/en-us/dotnet/core/tutorials/with-visual-studio-code

  for Microsoft Visual Studio and Visual Studio Code, respectively.

- If you seriously struggle with the remaining concepts used in this practical task, we recommend you to start reading the entire Sections 1 and 2 of SIT232 Workbook.

- In this unit, we will use Microsoft Visual Studio 2017 to develop C# programs. Find the instructions to install the community version of Microsoft Visual Studio 2017 available on the SIT232 unit web-page in CloudDeakin in Resources → Software → Visual Studio Community 2017. You however are free to use another IDE, e.g. Visual Studio Code, if you prefer that.

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your programs implement the required functionality. They must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- For this task, during your discussion, you must demonstrate how you run your program in debug mode to detect and fix potential errors related to objects and their use. Note that debugging is an essential skill necessary to implement errorless code: The sooner you start debugging your programs the sooner you will become proficient with programming and understanding the theory part. We do not expect you to demonstrate this skill for every program you write, so task 4 of this practical should serve as a good example.

  Cloud students must include debugging process complemented with their verbal explanation into the video.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail one of the deadlines, you fail the task and this reduces the chance to pass the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.