Practical Task 5.1

(Pass Task)

Submission deadline: 10:00am Monday, August 30
Discussion deadline: 10:00pm Friday, September 17

General Instructions

This practical task introduces you to ***inheritance***, a mechanism in which one class acquires the features and behaviour of another class. The class whose features are ***inherited*** is known as a ***super class*** (or a ***base class*** or a ***parent class***). The class that ***inherits*** from the other class is known as a ***subclass*** (or a ***derived class***, or an ***extended class***, or a ***child class***). The subclass can add its own attributes and methods in addition to those of the superclass. Inheritance supports the concept of "***reusability***". That is, when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the attributes and methods of the existing class.

This task covers the basics; however, there is a lot more to test and explore. Its first part will set up a Zoo program without the use of inheritance. We will then move on to create it using inheritance and you should see the benefits of being able to control how the objects you create behave.

1. This part requires you to create a console application which will print the details of several animals in a zoo. This example does so without the use of inheritance.

   STEP 1: Start a new C# Console Application project and rename its main class `Program` to `ZooPark`. Along with the `ZooPark` class, you need to create an `Animal` class. The `ZooPark` class is where you will create animal objects and print out their details to the console. Add the `Animal` class to the project. The `Animal` class will be a blue print that contains the states and behaviours relevant to all animals.

   STEP 2: You will now need to add several variables and methods to the `Animal` class. Let's declare the attributes (states) that each animal will have, e.g. `name`, `age`, and etc. This should be declared in the `Animal` class as follows.

```
class Animal
{
    private String name;
    private String diet;
    private String location;
    private double weight;
    private int age;
    private String colour;
}
```

   Add a ***constructor*** to the `Animal` class that will allow you to create an instance of the `Animal` class with the details specified by the code below.

```
public Animal(String name, String diet, String location,
    double weight, int age, String colour)
{
    this.name = name;
    this.diet = diet;
    this.location = location;
    this.weight = weight;
    this.age = age;
    this.colour = colour;
}
```

   We now have enough set up to be able to create `Animal` objects in the `ZooPark` class. Therefore, let's navigate to the main `ZooPark` class and write the following code in its `Main` method.

```
Animal williamWolf = new Animal("William the Wolf", "Meat", "Dog Village", 50.6, 9, "Grey");
Animal tonyTiger = new Animal("Tony the Tiger", "Meat", "Cat Land", 110, 6, "Orange and White");
Animal edgarEagle = new Animal("Edgar the Eagle", "Fish", "Bird Mania", 20, 15, "Black");
```

STEP 3: You now will add methods to the `Animal` class that will allow each animal to carry out several common functions:

– `sleep()`: a method to allow the animal to lie down and take a nap
– `eat()`: a method to allow the animal to eat
– `makeNoise()`: a method to allow the animal to make a sound

To do this, return to the `Animal` class and enter the following code.

```
public void eat()
{
    //Code for the animal to eat
}

public void sleep()
{
    //Code for the animal to sleep
}

public void makeNoise()
{
    //Code for the animal to make a noise
}
```

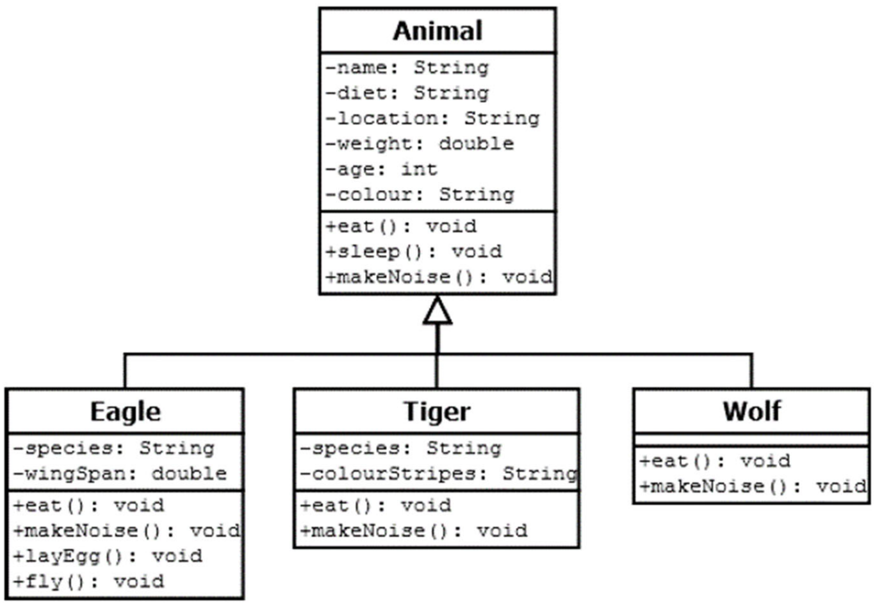
The `Animal` class now allows the animals to eat, sleep and make a noise. However, not all animals make the same noise, and they do not eat the same food. Therefore, we need more methods to be added.

```
public void makeLionNoise()
{
    //Code for the lions to roar
}

public void makeEagleNoise()
{
    //Code for the eagles to cry
}

public void makeWolfNoise()
{
    //Code for the wolves to howl
}
```

```
public void eatMeat()
{
    //Code for the animal to eat meat
}

public void eatBerries()
{
    //Code for the animal to eat berries
}
```

As you can see, the class is becoming quite cluttered and contains methods that will not be used by all animals. For example, wolves will not make an eagle noise or a lion noise but the `Animal` class contains methods to allow this.

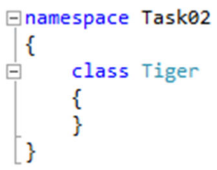
2. You will now use inheritance to create a ***base*** `Animal` class, which will give the animals everything they have in common (e.g., 'name' and 'make a noise'), and then create ***subclasses*** that inherit from the base class and contain more specific behaviours. This should enable you to see the benefits of being able to control how objects behave both generically and specifically. This is best described using the following UML diagram.



STEP 1: Start a new C# Console Application project and import both the `Animal` and the `ZooPark` classes. Do not change the names of the classes, but rename their code files to `AnimalWithInheritance.cs` and `ZooParkWithInheritance.cs`, respectively. Navigate to the `Animal` class and remove all methods except for `eat()`, `sleep()`, and `makeNoise()`. Obviously, you should not delete the constructor.

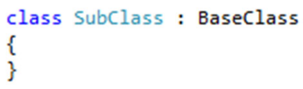
The `Animal` class is the ***base class*** which contains all of the generic states and behaviours that each animal will have. However, the zoo contains several different animals including Tigers, Eagles and Wolves. We will now have to create a class for each specific animal, but it will be able to inherit generic states and behaviours from the `Animal` base class.

Proceed with a `Tiger` class that inherits from the base `Animal` class. To do this, add a new class to your project and name it as `Tiger`. This will create a blank class as depicted below.



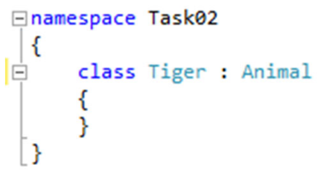
```
namespace Task02
{
    class Tiger
    {
    }
}
```

As we mentioned earlier, you should inherit the generic states and behaviours from the base class, i.e. the `Animal` class. This means that we can apply any generic attributes or functions, but in the subclass we can then define more specific attributes and functions (c.f. the above UML). The syntax is as follows.



```
class SubClass : BaseClass
{
}
```

Therefore, to allow the `Tiger` to inherit from the `Animal` class, change the class definition accordingly.

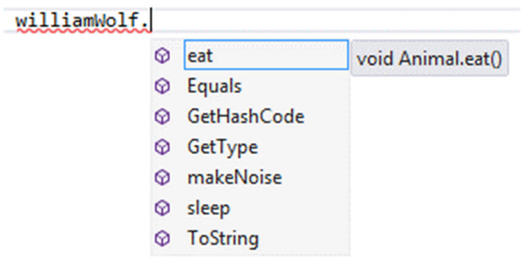


```
namespace Task02
{
    class Tiger : Animal
    {
    }
}
```

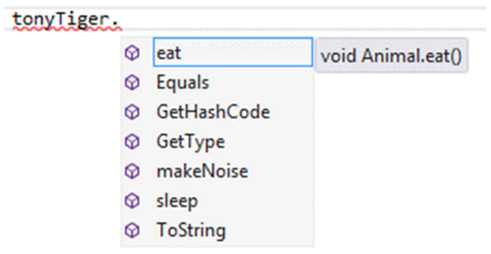
STEP 2: Let's navigate back to the `ZooPark` class where you will see the three `Animal` objects created. Comment out the `tonyTiger Animal` class object; this has been set up using the base class, but we now want it to be of the specific subclass `Tiger`. Now, create an instance of the `Tiger` class to see if it does in fact inherit from the `Animal` class.

```
Animal williamWolf = new Animal("William the Wolf", "Meat", "Dog Village", 50.6, 9, "Grey");
//Animal tonyTiger = new Animal("Tony the Tiger", "Meat", "Cat Land", 110, 6, "Orange and White");
Animal edgarEagle = new Animal("Edgar the Eagle", "Fish", "Bird Mania", 20, 15, "Black");

Tiger tonyTiger = new Tiger();
```

Note that you currently have no constructor in the `Tiger`, and therefore you do not need to specify values for any attributes. If you try to access the public methods and variables of the `Animal` object and the `Tiger` object, you can see that the `Tiger` has inherited all of the public methods and attributes.



```
williamWolf.
eat          void Animal.eat()
Equals
GetHashCode
GetType
makeNoise
sleep
ToString
```



```
tonyTiger.
eat          void Animal.eat()
Equals
GetHashCode
GetType
makeNoise
sleep
ToString
```

STEP 3: Now let's create a constructor for the `Tiger`. It will inherit the attributes from the base class and also set a species and a number of stripes. Return to the `Tiger` class and add a constructor as defined below.

```
class Tiger : Animal
{
    private String species;
    private String colourStripes;

    public Tiger(String name, String diet, String location,
        double weight, int age, String colour, String species, String colourStripes)
        : base(name, diet, location, weight, age, colour)
    {
        this.species = species;
        this.colourStripes = colourStripes;
    }
```

We pass in the values for `name`, `diet`, `location`, `weight`, `age`, `colour`, `species` and `numStripes`. The first six are immediately passed to the base class to set up the appropriate generic instance variables. The `species` and `numStripes` parameters are then used within the `Tiger` constructor to set up the specific instance variables related to the tiger.

STEP 4: Return to the `ZooPark` class and amend the creation of the `Tiger` object as follows.

```
Tiger tonyTiger = new Tiger("Tony the Tiger", "Meat", "Cat Land",
    110, 6, "Orange and White", "Siberian", "White");
```

STEP 5: Create classes for a `Wolf` and an `Eagle` that all inherit from the base `Animal` class as detailed by the above UML. Alter the creation of the two remaining `Animal` objects so that they are now an `Eagle` object and a `Wolf` object.

```
Tiger tonyTiger = new Tiger("Tony the Tiger", "Meat", "Cat Land",
    110, 6, "Orange and White", "Siberian", "White");
Wolf williamWolf = new Wolf("William the Wolf", "Meat", "Dog Village",
    50.6, 9, "Grey");
Eagle edgarEagle = new Eagle("Edgar the Eagle", "Fish", "Bird Mania",
    20, 15, "Black","Harpy", 98.5);
```

At this moment, you have subclasses for each animal and they all inherit from the base `Animal` class. We also have specified the `Animal` class' subclasses so that they all take the attributes from the base class, but both the `Tiger` and the `Eagle` have more specific states. The idea of ***inheritance*** is to implement the '***IS-A***' relationship. For example, `Tiger` '***IS-A***' `Animal` and `Wolf` '***IS-A***' `Animal`.

STEP 6: We now need to add some methods that are unique to each animal subclass. We will use the `Eagle` as an example. Therefore, open the `Eagle` class and add the following methods.

```
public void layEgg()
{
    //code to allow eagles to lay eggs
}

public void fly()
{
    //code to allow eagles to fly
}
```

If you try to access the public methods and variables of an `Eagle` object, you can see that the `Eagle` has inherited all of the public methods and attributes of `Animal`, but also has its own specific methods and attributes.

3. Different animals make different noises: Tigers roar, wolves howl and eagle's whistle. Each of the classes that inherit from the `Animal` will a have the `makeNoise()` method, but each of those methods will work in a different way and have different code.

When a subclass changes the behaviour of one of the methods that it ***inherits***, we say that it ***overrides*** the method. Therefore, when you have a subclass that inherits from a base class, it must inherit all of the base behaviours, but you can modify them in the subclass so they are not performed exactly the same way (i.e., they can be made specific).

STEP 1: Continue working on your project from the second part of this task. To allow the base methods to be overridden, the base methods must be marked as ***virtual***. The '***virtual***' keyword designates a method that is ***overridden*** in subclasses. Open the `Animal` class and label the `makeNoise()` method ***virtual***.

```
public virtual void makeNoise()
{
    Console.WriteLine("An animal makes a noise");
}
```

Since the base `makeNoise()` method is now ***virtual***, we can override this method in the `Tiger` class to allow the `Tiger` object to "ROAARRRR" when the `makeNoise()` method is called. To do this, create a `makeNoise()` method in the `Tiger` class and include the override keyword in the method declaration as shown below.

```
public override void makeNoise()
{
    Console.WriteLine("ROARRRRRRRRR");
}
```

STEP 2: Return to the `ZooPark` class and call the `makeNoise()` method on a `Tiger` object and an `Animal` object.

```
tonyTiger.makeNoise();

Animal baseAnimal = new Animal("Animal Name", "Animal Diet", "Animal Location",
    0.0, 0, "Animal Colour");
baseAnimal.makeNoise();

Console.ReadLine();
```

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

STEP 3: Add an ***override method*** for `makeNoise()` in the `Eagle` and `Wolf` subclasses to perform the correct animal noise, e.g. `Wolf` howls, `Eagle` whistles.

In addition, override the `eat()` method in the `Tiger` subclass to allow the tiger to eat the correct amount of meat. To do this, first amend the `eat()` method as ***virtual*** in the base `Animal` class.

```
public virtual void eat()
{
    Console.WriteLine("An animal eats");
}
```

Now, create an `eat()` method in the `Tiger` class and include the '***override***' keyword in the method declaration.

```
public override void eat()
{
    Console.WriteLine("I can eat 20lbs of meat");
}
```

Return to the `ZooPark` and call the `eat()` method on a `Tiger` object, `Wolf` object and an `Animal` object to view which method is executed.

```
tonyTiger.eat();
baseAnimal.eat();
williamWolf.eat();
```

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

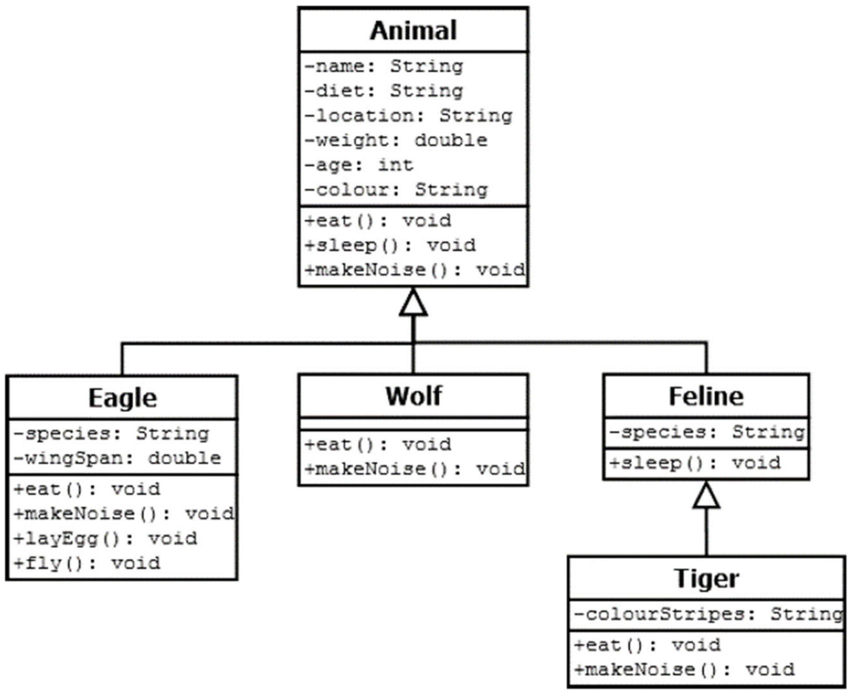
STEP 4: As we have not yet overridden the `eat()` method in the `Wolf` subclass, it looks to the class. Therefore, when we call the `eat()` method on the `Wolf` object it returns "An animal eats".

Override the `eat()` method in each subclass to allow each animal to eat the correct food and the correct quantity, e.g. `Wolf` eat 10lbs of meat, `Eagles` eat 1lb of fish. Test the `eat()` method for each of the objects. Compile and run the program.

STEP 5: It is time to be creative; so decide on a new method that the animal will perform. Add a new method in the `Animal` class that can be overridden and then override the method in the subclasses that can perform this action. You must also complete the other methods that are part of the subclasses by adding appropriate functionality and test them.

4. So the Zoo Park is starting to take shape, but could we make it better? At this moment, the `Tiger` inherits from the base class `Animal.` But if a lion joined our zoo, then since they are both big cats they will have some common methods and attributes.

STEP 1: Why do not we make a `Feline` class that inherits from the `Animal` and make the `Tiger` inherit from the `Feline` class instead of the `Animal` class? Then the `Feline` class will contain all of the common methods and attributes specific to cats and also all from the `Animal` base class. The following figure illustrates this through an updated UML.



Add the new Feline class that inherits from the base Animal class by writing the following code:

```
class Feline : Animal
{
    private String species;

    public Feline(String name, String diet, String location,
        double weight, int age, String colour, String species)
        : base(name, diet, location, weight, age, colour)
    {
        this.species = species;
    }
}
```

STEP 2: We will now need to alter the Tiger class as it needs to inherit from the Feline (rather than the Animal) class. Amend the code in the Tiger as shown below.

```
class Tiger : Feline
{
    private String colourStripes;

    public Tiger(String name, String diet, String location,
        double weight, int age, String colour, String species, String colourStripes)
        : base(name, diet, location, weight, age, colour, species)
    {
        this.colourStripes = colourStripes;
    }
```

Add the common Feline attributes and methods to the Feline class, as per the above updated UML.

STEP 3: Return to the ZooPark and call methods that are defined, respectively, in the Tiger class, the Feline class, and the base Animal class.

```
baseAnimal.sleep();
tonyTiger.sleep();
williamWolf.sleep();
edgarEagle.sleep();
tonyTiger.eat();
```

Save your work and run the program. The sleep() method is executed in the Feline class for the Tiger class object, but for all other objects the method in the base class is executed. The eat() method is executed in the Tiger class.

STEP 4: The Zoo has a new arrival and it is a lion. Now add a new Lion class that inherits from the Feline.

STEP 5: The Zoo has another new arrival and it is a Penguin. Now add a new Bird class that inherits from the base Animal class as there are now two birds that will share some common attributes and methods, e.g. wingspan and bird species.

You will also need a new Penguin class that inherits from the Bird class. You should also change the Eagle class to inherit from the Bird class instead of doing this directly from the Animal class. Think about methods and attributes that are unique to each type of the birds. For example, you could add a fly() method into the Bird class, and override the method in each subclass depending on if that bird can fly or not.

Remember to fully test your program using the ZooPark main class.

5. ***Overloading*** is what happens when you have two or more methods with the same name but different ***signatures***.

   Start a new C# Console Application project and name its main class as Overloading. Add the following two methods that have the same name but a different parameter list in the method's signature.

```
public static void methodToBeOverloaded(String name)
{
    Console.WriteLine("Name: " + name);
}

public static void methodToBeOverloaded(String name, int age)
{
    Console.WriteLine("Name: " + name + "\nAge: " + age);
}
```

We now have two methods that can be called by passing in one variable or two. Call each method using different parameters to print out the variables to the console.

At the compilation time, the compiler works out which one is to be called based on the compile time types of the arguments and the target of the method call. For example, if the method call contains a **string** and an **int**, then it calls the second method of the list that prints the values of the two variables. C# generally does not allow two methods in the same class to have the **same name** unless the number or the types of parameters differ.

6. In this last exercise, first, take a look at `ClassA` and `ClassB` given below.

```
public class ClassA
{
    private int i = 10;

    protected int Sum(int j)
    {
        return i + j;
    }

    public int Product(int j)
    {
        return i * j;
    }

    public virtual double Division(int j)
    {
        return i / j;
    }
}
```

```
public class ClassB : ClassA
    {
        public override double Division(int j)
        {
            return (double)i / j;
        }

        public void PrintResults(int j)
        {
            Console.WriteLine("i: {0}", i);
            Console.WriteLine("Sum(j): {0}", Sum(j));
            Console.WriteLine("Product(j): {0}", Product(j));
            Console.WriteLine("Division(j): {0}", Division(j));
        }
      }
```

Now, explore the following code snippet that is to be a part of the `Main` method.

```
ClassA a = new ClassA();
ClassB b = new ClassB();

Console.WriteLine("Sum by class A: {0}", a.Sum(3));
Console.WriteLine("Product by class A: {0}", a.Product(3));
Console.WriteLine("Division by class A: {0}", a.Division(3));

Console.WriteLine("Sum by class B: {0}", b.Sum(3));
Console.WriteLine("Product by class B: {0}", b.Product(3));
Console.WriteLine("Division by class B: {0}", b.Division(3));


b.PrintResults(3);
```

There are multiple compilation errors related to this code and the implementation of `ClassA` and `ClassB`. Your task is to find all existing issues and answer the questions listed below. Try first to figure out the errors from the pictures, then transfer the code to your IDE to run it to confirm your guess.

– Does b, the instance of `ClassB`, have an instance variable `i` accessible?
– Is the first call to `Console.WriteLine` in `PrintResults` in `ClassB` legal?
– Is the second call to `Console.WriteLine` in `PrintResults` in `ClassB` legal?

Fix the program code of the snippet and the classes to make the program compilable. Run the program to get output in the terminal. Explore the output and explain the difference in the behaviour of the two classes.

Further Notes

- Explore Section 5 of the SIT232 Workbook to learn the concept of inheritance, how it is applied in object-oriented programs, and when to use it. That section also explains the concept of reusability, how it is represented in object-oriented programs, and how to increase the likelihood your code will be reused. The Workbook is available in CloudDeakin → Learning Resources.
- Explore Section 3.4.2 of the Workbook to study the concept of method overloading; that is, the way of defining methods with the same name but a different set of arguments.
- The following links give some additional notes on inheritance, overriding and overloading in C#:
  - https://www.c-sharpcorner.com/UploadFile/8a67c0/method-overloading-and-method-overriding-in-C-Sharp/
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/versioning-with-the-override-and-new-keywords
  - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/knowing-when-to-use-override-and-new-keywords
  - https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/member-overloading

Submission Instructions and Marking Process

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

- Make sure 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 programs thoroughly before submission. Think about potential errors where your programs might fail.
- **Submit** the expected code files as a solution to the task via OnTrack submission system. You must **record a short video explaining your 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. Note that the video recording must be made in the **camera on mode**; that is, the tutor must see both the presenter and the shared screen.
- Once your solution is accepted by the tutor, you will be invited to **continue its discussion and answer relevant theoretical questions** through the Intelligent Discussion Service in OnTrack. Your tutor will record several audio questions. When you click on these, OnTrack will record your response live. You must answer straight away in your own words. As this is a live response, you should ensure you understand the solution to the task you submitted.

  Answer all additional questions that your tutor may ask you. Questions will cover the lecture notes; so attending (or watching) the lectures should help you with this **compulsory discussion part**. You should start the discussion as soon as possible as if your answers are wrong, you may have to pass another round, 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 grading your achievements 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 the quality of your solutions.