2. Encapsulation

In this chapter you will learn:

What encapsulation is
Why encapsulation is used
How to use private and protected attributes and methods
How to properly encapsulate a program

Encapsulation in object-oriented programming

As mentioned in Chapter 1, **encapsulation** is the idea of grouping data and subroutines to make a program easier to work on and understand. In object-oriented programming, encapsulation is achieved by using classes. A class should only contain the attributes and methods that it needs, and none of the logic of one class should depend on the internal processing in another class.

Imagine a company has a system that stores various information about different employees. If the system does not use encapsulation, any data can be used or altered in any part of the program. This is an issue for two reasons:

1. If any errors occur it will be much harder to identify the source of the error, because it could originate from anywhere. In a properly encapsulated program, any errors will originate either from the part of the program that isn't working correctly, or from an error in how different parts of the program interact with each other.
2. It means that some parts of the system will have access to attributes and methods that they shouldn't have access to. In the example of a company's employee information system, it would make sense for the employee to be able to update some of their own records (such as their name or bank details), but they shouldn't have access to change other information (such as their salary) or be able to see certain information about other employees (such as their addresses).

Private attributes and methods

Pseudocode

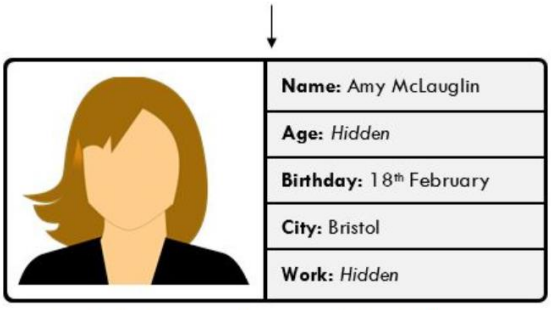
```
class Account

    private accountPassword //This attribute is private

    ...

    public function checkPassword(password) //This method is public

    ...

endclass
```

```
class Bank
        private accounts //This attribute is private
        ...
        public procedure withdraw(number) //This method is public
        ...
Endclass
```

Take the above program based on Task 1. Notice how **private** and **public** keywords are used here. If an attribute or method is private, it can only be accessed from within the class. If an attribute or method is public, it can be used by other classes. In Task 1, the attribute accountPassword in the Account class is made private so that the Bank class cannot see passwords which it shouldn't have access to. Instead, the Bank class must use the public method checkPassword to check whether a user has entered their password correctly. This is useful for security purposes (in a real-world system, the more a password is shared across the system, the more vulnerable it is to being stolen) and for encapsulating the program (the Bank class doesn't need to know what the password is, it only needs to be able to check that a password is correct, so it shouldn't have access to that data).

Pseudocode

```
class Account
        public profileImage
        public name
        private age
        public birthday
        public city
        private work
        ...
```



Private attributes can be hidden from other parts of the system.
There are many situations where you may want to hide some of the data being stored about a particular object.

When an attribute from another class is needed, instead of making that attribute public, you can create a public method that returns the value of the attribute. Similarly, to change the value of an attribute, you can create a public method to change its value rather than directly altering it. Methods that return the value of a private attribute are known as **accessors** (or 'getters'), and methods that alter the value of a private attribute are known as **mutators** (or 'setters'). These may at first appear unnecessary, but can be useful if you ever want to change the functionality of the class.

For example, imagine you have the following code:

Pseudocode

```
class Clock

        public currentTime //This attribute is public

        ...

endclass



class Display()

        clock = new Clock()

        ...

        public procedure showTime()

                print(clock.currentTime)//currentTime is public

                                        //so can be called in Display

        endprocedure

endclass
```

The Display class directly accesses the clock's currentTime attribute to display the time. This works fine, but if you wanted to make a change to how the clock's time is displayed (e.g. by making it a 12-hour clock instead of a 24-hour clock, or changing whether seconds or milliseconds are displayed) and the Display class referred to currentTime in multiple places, then formatting or other checks would need to be added in multiple places throughout the Display class, which could mean changing a lot of code. The program could be instead be written as:

Pseudocode

```
class Clock

        private currentTime //currentTime is made private

        ...

        public function getTime() //This public method gives access to currentTime

                return this.currentTime

        endfunction

endclass



class Display()

        clock = new Clock()

        ...

        public procedure showTime()

                print(clock.getTime()) //The public method is called in Display

        endprocedure

endclass
```

With this version of the program, the change could be made to the getTime accessor so that the Display class does not need to be updated. Accessors and mutators should not just be used to make a private attribute public, but to hide information from other classes or limit the ability of other classes to alter the attribute.

Python Note:

In Python, there are no 'private' or 'public' keywords; in fact, all methods and attributes are public in Python. However, if the name of an attribute or method begins with a double underscore, it cannot be accessed as easily. So, for the class:

```
class Class:
    def __init__(self, publicAttribute, privateAttribute):
        self.publicAttribute = publicAttribute
        self.__privateAttribute = privateAttribute
```

If you tried to access the attributes from outside of the class:

```
class = Class("Public", "Private")
print(class.publicAttribute)
print(class.__privateAttribute)
```

The program would print the value of class.publicAttribute ('Public') but throw an error when it tries to get the value of class.__privateAttribute. The '__' doesn't make __privateAttribute private, but instead changes the name of the attribute when it is called outside of the class. So:

```
print(class._Class__privateAttribute)
```

Would print the value of class.__privateAttribute ('Private'). While these values can still technically be accessed from outside of the class in which they are defined, you may treat any attributes or methods that begin with a double underscore as private.

Questions

Q1	Define the term encapsulation. (1 mark)
Q2	Explain the difference between a private attribute or method and a public attribute or method. (2 marks)
Q3	Explain one reason why an attribute may be made private. (1 mark)

Q4	Define the terms accessor and mutator. (2 marks)
Q5	Identify when accessors and mutators should be used. (2 marks)
Q6	Explain why you might make an attribute public instead of using accessors and mutators. (2 marks)