Unit D: Object orientated programming part 2

The previous Object Orientated Programming chapter focused on the theoretical ideas involved. This chapter will introduce how to actually implement OOP in your programming and draw your attention to various technical issues you should be aware of.

1. Defining class identity

As previously discussed, a **class** is analogous to creating a new variable data type. The individual variables created of that type are known as **objects**. This means prior to creating objects, we must define the class that it will belong to.

The class is the blueprint to the object. Think of the class as architectural drawings, and the objects as all the homes built from those drawings.

Let's take a look at how to create a simple class to represent a Person.

Part of the rationale for object orientated programming is to increase code reuse through modularity. To facilitate that, best practice is for each class to be in its own unique programming file. This is a requirement in Java, and a recommendation in Python.

Create a class with Python

With Python, to have a class in a separate file means using an import statement to load it into the main file. The "from name" portion of the import statement you can see below must match the name of the Python file exactly (case-sensitive) without the ".py" extension. The following code creates a Person class, so it will require two files: **main.py** and **person.py**.

```python
# main.py
from person import Person    # from the person.py file, import the Person class

p = Person("Groot")          # Create a Person object named `p`
print(p.get_name())          # prints 'Groot'
```

```python
# person.py
class Person:
    def __init__(self, name):
        print("Person constructor is running...")
        self.name = name

    def get_name(self):
        return self.name
```

Create a class with Java

As indicated, each class in Java needs its own unique file. The name of the file must perfectly match the name of the class it contains (case-sensitive). The following code creates a Person class, so it will require two files: **Main.java** and **Person.java**.

```
// Main.java
class Main {
    public static void main(String[] args) {
        Person p = new Person("Groot");      // Create a Person object named `p`
        System.out.println( p.getName() );
    }
}
```

```
// Person.java
public class Person {
   private String name;

   Person(String name) {
      System.out.println("Person constructor is executing...");
      this.name = name;
   }

   public String getName() {
      return this.name;
   }
}
```

2. Class basics

2.1 Instance variables

Consider these terms equivalent: Properties, attributes, and variables

One of the advantages of object orientated programming is that the variables associated with an object, and the functions that use and manipulate those variables, can be bound together and used across your program as a single entity.

The variables stored within an individual object are known as instance variables. This is because there is a separate copy made of these variables for every instance of the class that is created.

As with procedural programming, these variables have a data type associated with them. These may be the implicit types int, long, double, String, or they could be another class.

The previous example that created a Person class had one instance variable, a String called **name**.

In Python the variable was declared when it was initialised by the constructor method described in the section that follows.

In Java the variable was declared by the **`private String name`** line shown here...

```java
public class Person {
   private String name;  // Declare an instance variable of String called `name`

   Person(String name) {
      // ... code continues
```

Best practice note: As instance variables are descriptive of an object, they should generally have noun based names.

2.2 Constructors & instantiation

This process of actually "creating" an object based on its class definition is known as instantiation. It is instantiation that will allocate the memory required to a new object based on the class definition and will execute the code necessary at its construction or birth. This is what occurs when your main, or some other function, is creating an object variable based on the class.

The constructor is the initialisation code you create for an object. This code will run automatically as part of instantiation. The constructor is simply a method (function). So that it can be run automatically it must be named a special way.

In Python, the constructor function must be named `__init__()`. In the previous **Person** example, this was the code fragment that represented the constructor.

```
    def __init__(self, name):
        print("Person constructor is running...")
        self.name = name
```

This function takes two parameters, but you don't have to manually provide the **self** parameter. That is a programmatic reference to itself, the object that the function belongs to. You must include the **self** parameter for all Python methods that are part of a class. Other than the **self** parameter, there is only one other parameter for this method, the string called **name**. This parameter was provided via initialisation in this code fragment.

```
p = Person("Groot")
```

The string, **`"Groot"`**, was provided to the name parameter of the `__init__()` constructor function.

In Java, the constructor function name must match the name of the class it represents. In the previous Person example, this was the code fragment that represented the constructor in Java.

```
   Person(String name) {
      System.out.println("Person constructor is executing...");
      this.name = name;
   }
```

This function takes one parameter, the string called name. This parameter was provided via initialisation in this code fragment.

```
Person p = new Person("Groot");
```

The string, **`"Groot"`**, was provided to the name parameter of the **`Person()`** constructor function.

Why exactly is the "Person" required twice in this line?

- The use of Person on the left, **`Person p`**, is declaring an object variable called **p** of type **Person**.
- The use of Person on the right, the **`new Person("Groot")`**, is invoking the constructor function which returns an instance of **Person** for assignment into the **p** variable.

If you really wanted to, you could split it over two lines to represent the two different actions like this (but no one does it this way)....

```
Person p;                   // Declare a new object variable called `p`.
p = new Person("Groot");    // Call the constructor and put the result into `p`.
```

Finally, in Java, the keyword **`this`** is a reference to the current object. You can use it from within the object to refer to itself such as one of its instance variables or methods.

3. Class methods

Note: Consider these terms equivalent: Behaviours, methods, and functions

3.1 Accessors and mutators

Best practice with object oriented programming involves creating *accessor* and *mutator* methods for other code to use your attributes. These may also be referred to as *getters and setters*. This involves creating methods by which code external to your class can get the value of an attribute, or set the value of an attribute.

The following example illustrates this. A string, **`name`**, exists within the Person class but rather than reading and writing this attribute directly, a **`get_name()`** and **`set_name()`** method have been written and it is those methods that the main code is using to access the value of **`name`**, or to mutate it's value.

```python
# Python
class Person:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, name):
        self.name = name

p = Person("Julien")
print(p.get_name())     # prints 'Julien'
p.set_name("Amanda")    # set name to 'Amanda'
print(p.get_name())
```

```java
// Java
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    void setName(String name) {
        this.name = name;
    }
    public static void main(String[] args) {
        Person p = new Person("Julien");
        System.out.println(p.getName());    // Output is Julien
        p.setName("Amanda");                // Set name to Amanda
        System.out.println(p.getName());    // Output is Amanda
    }
}
```

On first glance that may seem like a lot of extra code for no obvious benefit. Why not just set the value for the **`name`** string directly, and print from it directly? Creating these accessor and mutator methods are actually essential for the purpose of preventing unintended side effects. By using accessor and mutator methods, it allows separation of responsibilities between the code inside the Person class and external code that uses Person as instantiated objects.

For the author of external code using Person objects, accessors and mutators mean you don't need to know or care how the **`Person`** class is storing the name information. This complexity is irrelevant to you. You simply call the **`.get_name()`** or **`.set_name()`** methods and trust it to reliably perform the intended task as advertised by its documentation.

For the author of the Person class, the **`name`** string may have been intended to represent just the given or family name, so if external code wrote a person's full name to that attribute it could cause an error in some other method that depends on the data being structured a particular way. Encapsulation through accessors and mutators, also provides flexibility for the author of the **`Person`** class to change how the internals of the class work at a later point. Provided the methods behave the same, no external code should break as a result.

3.2 Helper methods

Any function contained within your class that is for the internal use of your class, without being used by code external to the class, can be referred to as a helper method.

Helper methods can be used by your class for its internal logic.

3.3 Access modifiers

We've seen the merit of creating accessor and mutator methods, but these won't be much help if programmers using our objects don't follow best practice, and attempt to read or write the object attributes directly. Access modifiers allow programmers to enforce this encapsulated approach. You are able to dictate which code outside your class can access or mutate individual attributes and methods within your class.

There are three or four access modifiers depending on your language.

- **private**: Setting an attribute or method to private indicates it should only be accessible by the current object.
- **protected**: Setting an attribute or method to protected indicates that it may be accessed by the current object, and any objects that inherit it (more on that later).
- **public**: Setting an attribute or method to public indicates it may be accessible to any code within your program.

- **default**: Java also has a fourth modifier, known as default when no modifier is provided. This allows access for any object within the same Java package (as denoted by the `package` statement at the top of your Java file). Python does not have an equivalent of this.

As a rule of thumb for beginner programmers looking to establish good habits, it is recommended to start with the following and then adjust as your needs require:

- All attributes (or properties) in a class should be private. Create accessor and mutator methods for any attribute you want external code to have access to.
- Methods (or functions) should be public if you want external code to have access to them. If the method is an internal helper function, set it to private.

Java and Python take different approaches to access modifiers so these will now be discussed separately.

Access modifiers in Python

The access for an individual attribute or method in Python is determined by the presence of leading underscores in its name.

- To designate an attribute or method as **private**, use two underscores to start the name.
- To designate an attribute or method as **protected**, use a single underscore to start the name.
- Any other attribute or method will be considered **public**.

Taking our previous example with a slight modification...

```
class Person:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name
```

The double underscore indicates the variable **`self.__name`** is private.

Be aware that Python access modifiers are really only enforced through convention. You can, if you really wanted to, write code outside the class that can read/write anything that is private. In the previous example, the following code would break...

```
p = Person("Max")
print(p.name)   # Causes an error
```

But the manner in which Python makes the **`__name`** variable private is actually just by invisibly renaming it for you. So, if you know how Python does this, you can bypass it as the following demonstrates...

```
p = Person("Max")
print(p._Person__name)   # Will work!!!
```

You can use access modifiers on any attribute or method in a class.

Access modifiers in Java

Like Python, you can use access modifiers on any attribute or method in a class. The access for an individual attribute or method in Python is determined by the use of a keyword as part of the attribute or method signature. The keywords are **`private`**, **`protected`**, and **`public`**. The absence of any keyword will set the access modifier to default.

Take note of the use of the keywords here...

```java
// Person.java
class Person {
    private String name;                    // Notice keyword `private`

    Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Person p = new Person("Julien");
        System.out.println(p.getName());    // Output is Julien
        p.name = "Amanda";                  // Compiler error as `name` is private
        p.setName("Amanda");                // This will work
        System.out.println(p.getName());    // Output is Amanda
    }
}
```

Unlike Python, the access modifiers of Java are strictly enforced.

3.4 Best practices

TODO:

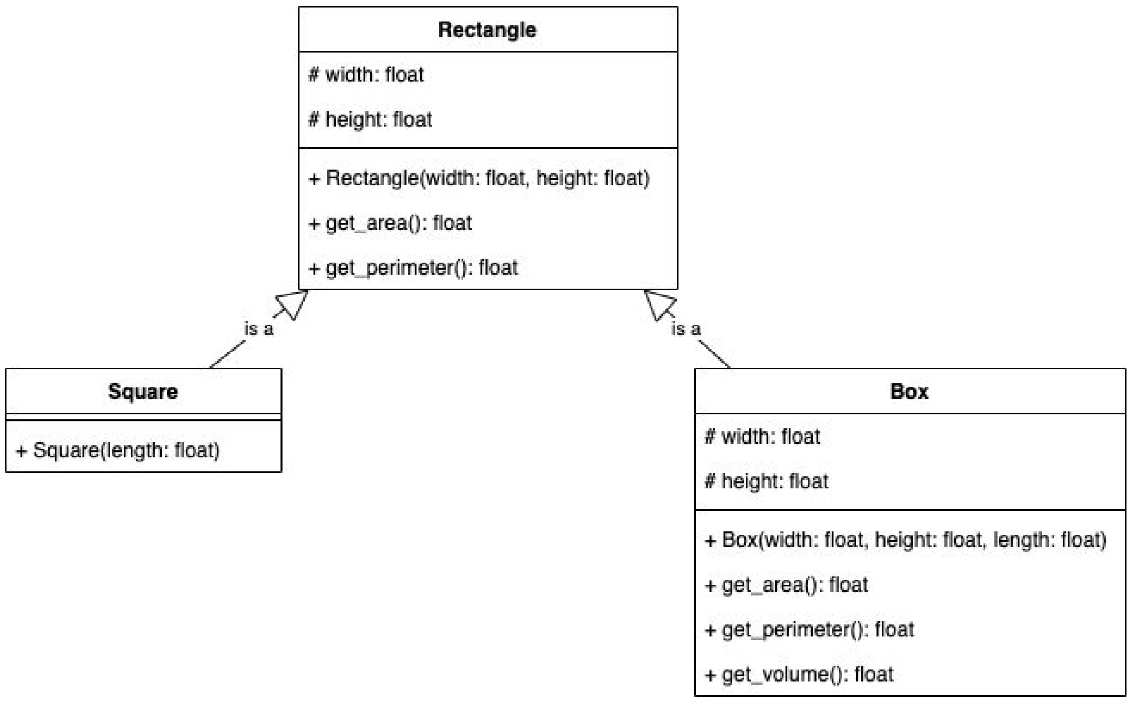
- Minimising side effects.
- Using verb names for methods.

4. Inheritance & polymorphism

4.1 Basic inheritance

A square can be thought of as a special type of rectangle. A box is also just a rectangle that has been extruded into a third dimension. Together these three shapes provide an ideal way to explore the idea of inheritance.

The UML diagram for our 3 classes will look like



Start by creating a **`Rectangle`** class. This will be our *parent* or *super* class that **`Square`** and **`Box`** will inherit from.

```
# Python
class Rectangle:
    def __init__(self, w:float, h:float):
        self.__w = w
        self.__h = h

    def get_area(self) -> float:
        return self.__w * self.__h

    def get_perimeter(self) -> float:
        return 2 * (self.__w + self.__h)
```

```java
// Java
class Rectangle {
    private double w;
    private double h;

    Rectangle(double w, double h) {
        this.w = w;
        this.h = h;
    }

    public double getArea() {
        return w * h;
    }

    public double getPerimeter() {
        return 2 * (w + h);
    }
}
```

You can test your Rectangle works by instantiating a couple of test rectangles.

```python
# Python
r1 = Rectangle(10.0, 5.0)
print(r1.get_area())
print(r1.get_perimeter())
```

```java
// Java
class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10.0, 5.0);
        System.out.println(r1.getArea());
        System.out.println(r1.getPerimeter());
    }
}
```

Once you have your Rectangle working, it's time to inherit the Rectangle into a Square. A square is just a special rectangle so all the properties and methods are identical. We don't need to add anything new other than a constructor, so this is a great first example to see how to structure our code to do inheritance.

In Python, we put the name of the super class name in parentheses following the name of the class we are creating. So to create a class Square that inherits from Rectangle, we write `class Square(Rectangle)`. Our Parent class still needs to run its constructor, so the first line of our Square constructor is to call the constructor Rectangle supplying the parameters it expects.

```
# Python
class Square(Rectangle):
    def __init__(self, length:float):
        Rectangle.__init__(self, length, length)
```

In Java, the **`extends`** keyword is used to denote the act of inheritance. To create a class Square that inherits from Rectangle, use **`class Square extends Rectangle`**. Our Parent class still needs to run its constructor, so that is expressed with the **`super()`** method to call the constructor for Rectangle.

```
// Java
class Square extends Rectangle {
    Square(double length){
        super(length, length);
    }
}
```

Now test your Square...

```
# Python
r1 = Rectangle(10.0, 5.0)
print(r1.get_area())
print(r1.get_perimeter())
s1 = Square(8.0)
print(s1.get_area())
print(s1.get_perimeter())
```

```
// Java
class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10.0, 5.0);
        System.out.println(r1.getArea());
        System.out.println(r1.getPerimeter());
        Square s1 = new Square(8.0);
        System.out.println(s1.getArea());
        System.out.println(s1.getPerimeter());
    }
}
```

4.2 Overriding

Overriding occurs when a child class creates an attribute or method of the same name as the parent class, therefore overriding it. You have actually already seen polymorphism by overriding but not had it pointed out to you that that is what was happening.

To look at overriding we will continue our previous example by adding a third class, **`Box`**, that will also inherit from **`Rectangle`**.

However, there is a catch. We should not have to double up our variables (storing a separate copy of `w` and `h` in the **Box**), but yet we need access to the width and height to perform our calculations.

This is where the **protected** access modifier comes in.

You will recall that a **protected** access modifier allows access to the variables by subclasses, which is what we are creating! So, we need to go back to our original **Rectangle** class and change those variables to **protected**.

Python: To convert the w & h variables in the Rectangle class from private to projected, change the double underscore variables to single underscore variables. That is **`self.__w`** and **`self.__h`** should become **`self._w`** and **`self._h`**.

```
# Python
class Rectangle:
    def __init__(self, w:float, h:float):
        self._w = w
        self._h = h

    def get_area(self) -> float:
        return self._w * self._h

    def get_perimeter(self) -> float:
        return 2 * (self._w + self._h)
```

Java: To convert the w & h variables in the Rectangle class from private to projected, change the keyword in front of the variable declaration from **private** to **protected** as shown below.

```
// Java
    protected double w;
    protected double h;
```

We can now create our Box class...

```python
# Python
class Box(Rectangle):
    def __init__(self, width:float, height:float, length:float):
        Rectangle.__init__(self, width, height)
        self._l = length

    def get_area(self) -> float:
        return 2 * (self._h * self._w + self._h * self._l + self._w * self._l)

    def get_perimeter(self) -> float:
        return 4 * (self._h + self._w + self._l)

    def get_volume(self) -> float:
        return self._w * self._h * self._l
```

```java
// Java
class Box extends Rectangle {
    private double l;
    Box(double w, double h, double l) {
        super(w,h);
        this.l = l;
    }

    public double getArea() { // Surface area
        return 2 * (w*h + w*d + h*d);
    }

    public double getPerimeter() { // All edges
        return 2 * super.getPerimeter() + 4 * l;
    }

    public double getVolume() {
        return d * super.getArea();
    }

    public double getSurfaceArea() {
        return getArea();
    }
}
```

Test that it works as you'd expect before moving on.

4.3 Overloading

Overloading occurs when the one method (function) can take multiple forms. This is generally when the combination and significance of the parameters required can vary.

In the examples below, are overloading the constructor so that a person's date of birth may be provided three different ways. While we are overloading the constructor in this example, it could just as easily be any other method within the class.

With Python we only define the method once and provide for overloading through the use of optional parameters. After a name parameter, any other parameters provided will be packaged into a Python list called args. Our method must then inspect the content of this list to determine what to do.

```
# Python
from datetime import datetime

class Person:
    def __init__(self, name, *args):
        # Accepts args as a string formatted to "yyyy-dd-mm", or a datetime object,
        # or three integers representing year, month and day
        self.__name = name
        if len(args) == 1:
            if type(args[0]) is str:            # is a string
                self.__birth = datetime.strptime(args[0], "%Y-%m-%d")
            elif type(args[0]) is datetime:     # is a datetime object
                self.__birth = args[0]
        elif len(args) == 3:
            self.__birth = datetime(args[0], args[1], args[2])

    def get_age(self):
        today = datetime.now()                  # Get today's date
        age = today.year - self.__birth.year
        if today.month < self.__birth.month:
            age = age - 1
        elif today.month == self.__birth.month and today.day < self.__birth.day:
            age = age - 1
        return age

p = Person("Margaret Hamilton", "1936-08-17")
print(p.get_age())
p = Person("Margaret Hamilton", 1936, 8, 17)
print(p.get_age())
p = Person("Margaret Hamilton", datetime(1936,8,17))
print(p.get_age())
```

It is important to point out that in order to keep this example brief, there is minimal checking of `args` occurring. Best practice would see the above code raising `ValueError` exceptions if the parameters provided are outside of design expectations.

In Java, you can't have optional parameters, but you can have multiple instances of a method name appearing provided the number and type of parameters vary. So, the Java solution would be to have three possible constructors, and let Java execute the one that matches the parameters received.

```java
// Java
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
import java.time.format.DateTimeFormatter;

class Person {
    private String name;
    private LocalDate birth;

    Person(String name, String birthdate) {     // Constructor
        this.name = name;
        this.birth = LocalDate.parse(birthdate, DateTimeFormatter.ISO_LOCAL_DATE);
    }

    Person(String name, int y, int m, int d) {  // Alternative constructor
        this.name = name;
        this.birth = LocalDate.of(y, m, d);
    }

    Person(String name, LocalDate birthdate) {  // Alternative constructor
        this.name = name;
        this.birth = birthdate;
    }

    public int getAge() {
        LocalDate today = LocalDate.now();
        int age = (int)ChronoUnit.YEARS.between(this.birth, today);
        return (age);
    }

    public static void main(String[] args) {
        Person p1 = new Person("Margaret Hamilton", "1936-08-17");
        Person p2 = new Person("Margaret Hamilton", 1936, 8, 17);
        Person p3 = new Person("Margaret Hamilton", LocalDate.of(1936, 8, 17));
        System.out.println(p1.getAge());
        System.out.println(p2.getAge());
        System.out.println(p3.getAge());
    }
}
```

5. Abstract classes

Sometimes you may find yourself creating a superclass that, while you know you will create subclasses from it, you never have any intention of actually instantiating the superclass itself. It might be more useful to be able to define a list of method names that should be created by the subclasses, without bothering to code them in the superclass.

Take the example of a geometric shape class, **`Shape`**. It may make sense to require that all subclasses will have a **`get_area()`** function and a **`get_perimeter()`** function, but it doesn't make sense to provide any code for either of those in the superclass because the logic will be entirely dependant on the form of shape the subclass is for.

Writing an *abstract class* is a mechanism by which we can create a template of methods that inheriting subclasses must provide without having to provide baseline functionality in the super.

Abstract classes in Python

In Python, we import the `abc` module, and inherit from `ABC` (for Abstract Base Class). Each method we are defining to be a requirement for subclasses should have the `@abstractmethod` decorator before the method declaration. It is also required to import the Abstract Base Class. Refer to the Python documentation here for more detail, https://docs.python.org/3/library/abc.html

```python
# Python
from abc import ABC, abstractmethod    # required imports
import math

class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass

    @abstractmethod
    def get_perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def get_area(self):
        return self._width * self._height

    def get_perimeter(self):
        return 2 * (self._width + self._height)

class Square(Rectangle):
    def __init__(self, length):
        Rectangle.__init__(self, length, length)
```

```
r = Rectangle(10,4)
print(r.get_area())
print(r.get_perimeter())
s = Square(5)
print(s.get_area())
print(s.get_perimeter())
```

Abstract classes in Java

In Java the *abstract* keyword is used for any class, attribute or method we wish to require in any derived class, as shown in the `Shape` class that follows.

```
// Shape.java
abstract class Shape{
   public abstract double getArea();
   public abstract double getPerimeter();
}
```

```
// Rectangle.java
class Rectangle extends Shape {
    protected double width;
    protected double height;

    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return this.width * this.height;
    }

    @Override
    public double getPerimeter() {
        return 2 * (this.width + this.height);
    }
}
```

```
// Square.java
class Square extends Rectangle {
    Square(double length) {
        super(length, length);
    }
}
```

```java
// Main.java
class Main {
  public static void main(String[] args) {
      Rectangle r = new Rectangle(10.0, 4.0);
      Square s = new Square(5.0);
      System.out.println(r.getArea());
      System.out.println(r.getPerimeter());
      System.out.println(s.getArea());
      System.out.println(s.getPerimeter());
  }
}
```

So far this makes sense, but consider the scenario of creating a **`Circle`** class. The abstract class **`Shape`** requires **`get_area()`** and **`get_perimeter()`** yet in circles the perimeter is usually known by the term circumference instead. What can be done about it? Well, you can always provide over and above the template, so there is no reason a **`get_circumference()`** method can't be written, but the **`get_perimeter()`** method must still exist otherwise the program will produce an exception.

One significant benefit of using Abstract classes in Java (it is less of an issue with Python due to it being less strict with typing), is if you want to build an array (or other collection) of mixed objects that all have a common ancestor. Because the use of the Abstract class guarantees any derived subclass will have the \`getArea()\` and \`getPerimeter()\` function, Java will allow us to execute our individually customised versions of those in an array of mixed shapes as the following example demonstrates.

```java
// Main.java
class Main {
  public static void main(String[] args) {
    Shape[] shapes = new Shape[4];
    shapes[0] = new Rectangle(20.0, 12.0);
    shapes[1] = new Rectangle(3.0, 7.0);
    shapes[2] = new Square(10.0);
    shapes[3] = new Circle(7.0); // Note this assumes you've created a Circle class

    for (Shape s : shapes) {
        System.out.println(s.getArea());
        System.out.println(s.getPerimeter());
    }
  }
}
```

Likewise if you'd prefer to work with `ArrayList` you can do the following:

```java
// Main.java
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {
    ArrayList<Shape> shapes = new ArrayList<Shape>();
    shapes.add(new Rectangle(20.0, 12.0));
    shapes.add(new Rectangle(3.0, 7.0));
    shapes.add(new Square(10.0));
    shapes.add(new Circle(7.0));

    for (Shape s : shapes) {
        System.out.println(s.getArea());
        System.out.println(s.getPerimeter());
    }
  }
}
```

Abstract class exercises

Continue extending the shapes system.

- Build classes for
  - Ellipse
  - Sphere
  - Cylinder
  - Triangle, and
  - RightAngledTriangle

  into your system. The two dimensional shapes should have `getArea()` and `getPerimeter()`. The three dimensional shapes should additionally support a `getSurfaceArea()` and `getVolume()`.

- Question: What do you do with the `getPerimeter()` function for shapes such as `Sphere` where it carries no meaning? Read this interesting discussion on StackOverflow and decide for yourself how to deal with the situation (there are a couple of methods proposed)...
  https://stackoverflow.com/q/5486402

6. Class methods & attributes

Class variables and methods, are those which belong to the class rather than an individual object. Because it belongs to the class, a class variable is shared in common across all the objects of that class. If one object changes the value, all the objects see that change.

Consider the example of a store that is using object orientated programming to maintain their product information. A class called **`Product`** may exist, but each product needs a unique **`ProductID`** number. Having a class variable helps us do that very easily.

```python
# Python
class Product:
    next_product_id = 0     # a class variable

    def __init__(self, name):
        self.__name = name
        self.__product_id = Product.next_product_id # read the class variable
        Product.next_product_id += 1                # assign the class variable

    def __str__(self):
        return (f"Product {self.__product_id} is {self.__name}")

a = Product("Milk")
b = Product("Bread")
c = Product("Apple")
d = Product("Mushroom")
print(a)
print(b)
print(c)
print(d)
```

```java
// Java
class Product {
    private String name;
    private static int nextProductID = 0;   // a class variable
    private int productID;

    Product(String name) {
        this.name = name;
        this.productID = nextProductID;     // read the class variable
        nextProductID++;                    // assign the class variable
    }

    public String toString() {
        return ("Product "+Integer.toString(productID)+" is "+name);
    }
    public static void main(String[] args) {
        Product a = new Product("Milk");
        Product b = new Product("Bread");
        Product c = new Product("Apple");
        Product d = new Product("Mushroom");
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}
```

The output of both the Python and Java versions should be:

```
Product 0 is Milk
Product 1 is Bread
Product 2 is Apple
Product 3 is Mushroom
```

In the same way we have class variables, we also have class methods. In fact, you can't execute a Java program without at least one class method, **`main()`**. Like the variables, the class methods belong to the class rather than any particular object. Because this means they exist before any objects are created, that is why the **`main()`** in Java must have the keyword **`static`** since it has to be able to run before any object has been created.

One frequent use of class methods is as alternative forms of constructors. Class methods can be used to create new instances of the object. An example of that is provided in the next section where the deserialisation function is a static method.

7. Serialising and deserialising

Once you've gone to all the effort of populating some amazing objects, your program loses all the information once it quits. The idea of serialisation and deserialisation addresses this. Serialisation will inspect the content of your object attributes and convert it into a stream of bytes for writing to disk. Deserialisation is the reverse, it will create populated objects from data loaded from disk. Fortunately both Python and Java make this process relatively painless.

The following adapts our previous Product example and adds serialisation and deserialisation to it.

```python
# Python
# Pickle is the Python module that performs serialisation/deserialisation
import pickle

class Product:
    next_product_id = 0

    def __init__(self, name):
        self.__name = name
        self.__product_id = Product.next_product_id
        Product.next_product_id += 1

    def __str__(self):
        return (f"Product {self.__product_id} is {self.__name}")

    def serialise(self):
        with open(f"product-{self.__product_id}.data", "wb") as f:
            pickle.dump(self, f)

    @classmethod
    def deserialise(cls, product_id):
        with open(f"product-{product_id}.data", "rb") as f:
            return pickle.load(f)

a = Product("Milk")
b = Product("Bread")
c = Product("Apple")
b.serialise()                # Save to product-1.data
d = Product.deserialise(1)   # Load from product-1.data
print(d)                     # "Product 1 is Bread"
```

```java
// Java
import java.io.*;   // Contains Serializable and the file io functionality

class Product implements Serializable {     // Must implement Serializable
    private String name;
    private static int nextProductID = 0;
    private int productID;

    Product(String name) {
        this.name = name;
        this.productID = nextProductID;
        nextProductID++;
    }

    public String toString() {
        return ("Product "+Integer.toString(productID)+" is "+name);
    }

    public void serialise() {
        String fileName = "product-"+Integer.toString(this.productID)+".data";
        try {
            FileOutputStream f = new FileOutputStream(new File(fileName));
            ObjectOutputStream o = new ObjectOutputStream(f);
            o.writeObject(this);
            o.close();
            f.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        }
    }

    public static Product deserialise(int id)
    throws FileNotFoundException, IOException, ClassNotFoundException {
        String fileName = "product-"+Integer.toString(id)+".data";
        FileInputStream f = new FileInputStream(new File(fileName));
        ObjectInputStream o = new ObjectInputStream(f);
        Product p = (Product)o.readObject();
        o.close();
        f.close();
        return p;
    }

    public static void main(String[] args) {
        Product a = new Product("Milk");
        Product b = new Product("Bread");
        Product c = new Product("Apple");
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        a.serialise();      // Save to product-0.data
        b.serialise();      // Save to product-1.data
        c.serialise();      // Save to product-2.data
        try {
            Product e = Product.deserialise(1);     // Load from product-1.data
            System.out.println(e);                  // "Product 1 is Bread"
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

8. OOP programming in context

Advantages and disadvantages

Once you start programming in Java, you'll quickly realise the importance of packages. Packages in Java are a mechanism to encapsulate a group of classes, interfaces and sub packages.

Packaging of classes is used as loading the entire suite of functionality would consume too many system resources, especially when many/most programs don't need it. A programmer can therefore make their application more efficient by loading in the various subsets of functionality to suit their needs.

Packages are also a convenient way of grouping together classes we have written for reuse in other programs. You can easily package several classes together, and bundle them into a JAR file to share with others in their library of packages.

As a result, complex algorithms and processes do not have to be “re-invented” but shared and re-used.

Some added advantages of this modularity is

- It (in theory) makes for easier debugging and testing. If your classes are properly encapsulated with no other classes affecting them, and they don't have side effects on others, then you should be able to fully test and debug a class in isolation for other parts of the code base.
- It (again, in theory) should also ease the process of programming in teams. Individuals can take responsibility for the programming of certain classes, and provided the internal programming is sound, it should be able to be placed into the larger project with ease.

Object oriented programming is popular in big companies, because it suits the way they write software. At big companies, software tends to be written by large (and frequently changing) teams of mediocre programmers. OOP imposes a discipline on these programmers that prevents any one of them from doing too much damage. The price is that the resulting code is bloated with protocols and full of duplication. This is not too high a price for big companies, because their software is probably going to be bloated and full of duplication anyway.

Where there are advantages, there are of course, disadvantages. It is also the case that some of the "advantages" are increasingly contentious so make up your own mind.

The most glaring disadvantages of OOP are:

- it increases complexity for small problems where a simple procedural approach would suffice
- it is a methodology that is unsuited to particular classes of problem
- Also the Java incarnation of OOP is known for being particularly verbose. You'll particularly notice this whenever trying to deal with any I/O.

The following is a very interesting paper, looking at some of the risks with our industry's increased reliance on software dependencies, particularly the use of 3rd party modular libraries in our codebase. I highly recommend reading it. Cox, Russ (2019) "Our Software Dependency Problem" available at https://research.swtch.com/deps

Programmer considerations

Some considerations for you when programming your projects:

- Inclusionary practices
- Features related to internationalisation and multi-language support
  - https://www.youtube.com/watch?v=0j74jcxSunY
- Different time zones
  - https://www.youtube.com/watch?v=-5wpm-gesOY
- Unicode (necessary for multi-language character support)
  - https://www.youtube.com/watch?v=MijmeoH9LT4

Do programmers have ethical & moral duties in their work? For instance:

- Is it the programmers duty to ensure adequate testing of products to prevent the possibilities of commercial or other damage?
- Acknowledging the work of other programmers?
- What is the Open Source movement, and ethically how important is it to contribute to it if you plan on taking advantage of open source code in your project?