Lecture 2: Inheritance & Polymorphism

Learning Objectives

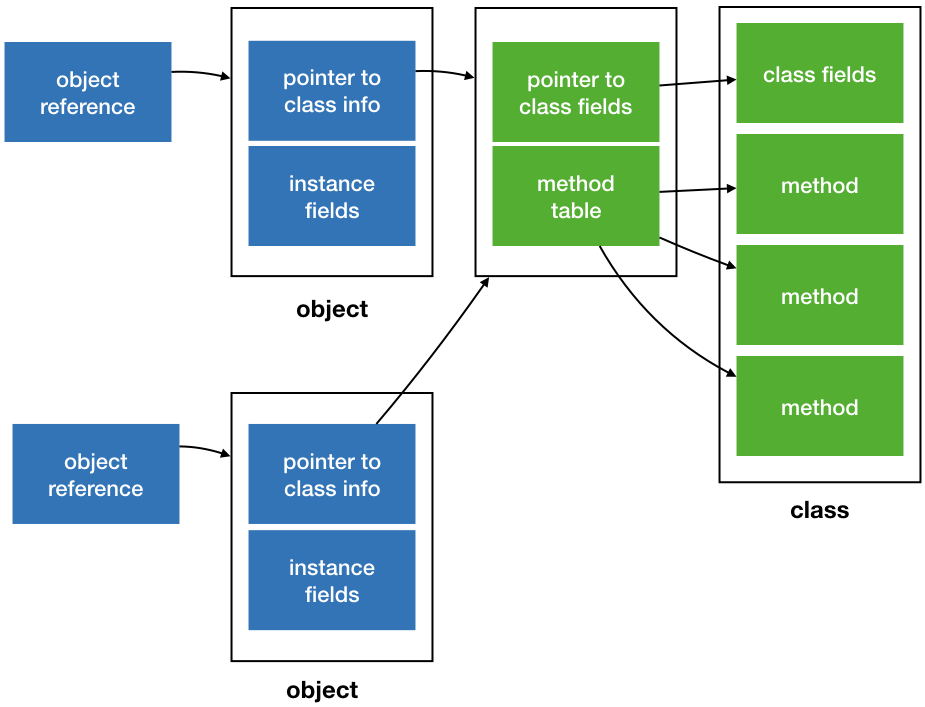
After this lecture, students should:

- be able to build a mental model for how objects and classes are represented in Java
- understand the concepts of object-oriented programming, including interface, polymorphism, late binding, inheritance, method overloading, and the usage of these concepts in programming.
- know the purpose and usage of Java keywords `implements`, `extends`, `super`, `this`, and `@Override`
- understand Java concepts of arrays, enhanced `for` loop, and method signature.

Memory Model for Objects

To help understand how classes and objects work, it is useful to visualize how they are stored in the memory. We mentioned last week that data (e.g., fields) and code (e.g., methods) are stored in two different regions in the memory. Since an object contains both fields and methods, where do we keep an object?

It turned out that different implementations of Java may store the objects differently, but here is one way that we will follow for CS2030:



In the figure above, there are two objects of the same class. An object is referred to through its reference, which is a pointer to the memory location where the instance fields for the object is stored, along with a pointer to a *method table*. A method table stores a table of pointers to the methods, along with a table to the class fields.

As an example, consider the following class:

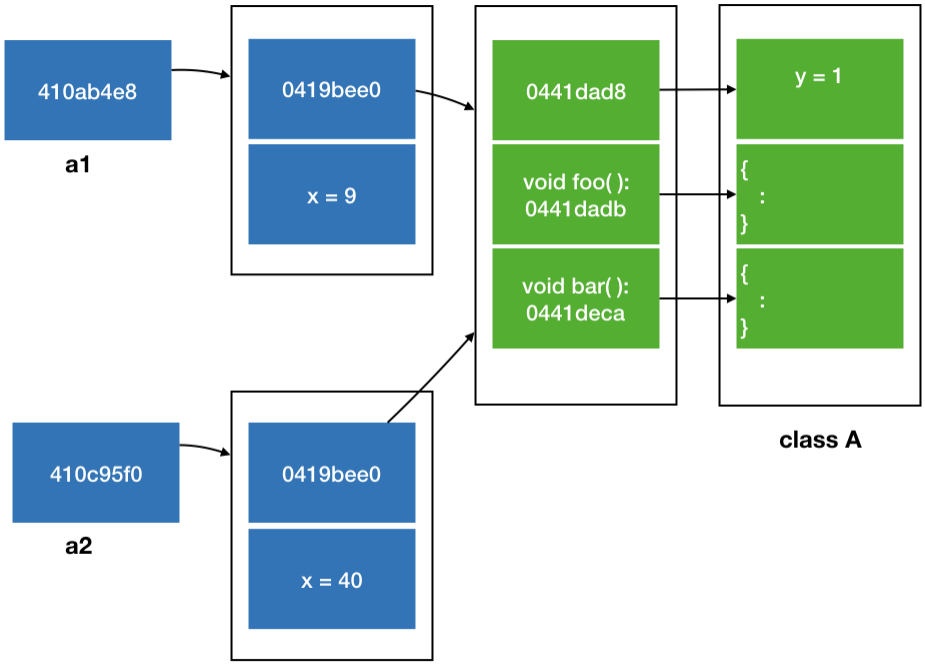
```
class A {
  private int x;
  static public int y;

  public void foo() {
    :
  }

  public void bar() {
    :
  }

    :
}
```

If we have two instances of `A`, `a1` and `a2`, with `A.y = 1`, `a1.x = 9`, `a2.x = 40`, then the memory layout looks like:



Note that, we have only one copy of the `static` class field `y`, regardless of how many instances of `A` we create.

Enforcing Abstraction Barrier with Interface

Recall the concept of encapsulation. When we develop a large piece of software, it is important to hide the details on data representation and implementation, and only expose certain `public` methods for the users to use. We can imagine that there is an abstraction barrier between the code that implements the internals of a class (called the *implementer*) and the code that uses the class (called the *client*) to achieve a higher level task.

We have seen that we use `private` to enforce data hiding -- to hide certain fields and methods from outside of the barrier. Now, we are going to see how we can enforce that the right set of methods are defined, implemented, and used on both sides of the barrier.

The mechanism to do this is through defining an *interface* (aka a *protocol* as it is called in Objective-C or Swift). An interface is like a contract between the implementer of a class and the client of a class. If a class promises to implement an interface, then we are guaranteed that the methods defined in the interface are already implemented in the class. Otherwise, the code would not compile.

In Java, we can define an interface using `interface` keyword:

```
1  interface Shape {
2    public double getArea();
3    public double getPerimeter();
4    public boolean contains(Point p);
5  }
```

The example interface `Shape` above contains only the declaration of the methods, not the implementation.

Now, let's see how the implementer would implement a class using the interface.

```
1  import java.lang.Math;
2
3  class Circle implements Shape {
4    private Point center;
5    private double radius;
6
7    public Circle(Point center, double radius) {
8      this.center = center;
9      this.radius = radius;
10   }
11
12   public void moveTo(Point p) {
13     center = p;
14   }
15
16   @Override
17   public double getArea() {
18     return Math.PI * radius * radius;
19   }
20
21   @Override
22   public boolean contains(Point p) {
23     return (p.distance(center) < radius);
24   }
25
26   @Override
27   public double getPerimeter() {
28     return Math.PI * 2 * radius;
29   }
30 }
```

This is very similar to the code you saw in Lecture 1, except that in Line 2, we say `class Circle implements Shape`. This line informs the compiler that the programmer intends to implement all the methods included in the interface `Shape` exactly as declared (in terms of names, the number of arguments, the types of arguments, returned type,

and access modifier). The rest of the class is the same, except that we renamed `getCircumference` with `getPerimeter`, which is more general and applies to all shapes. We also added *annotations* to our code by adding the line `@Override` before methods in `Circle` that implement the methods declared in `Shape`. This annotation is optional, but it informs the compiler of our intention and helps make the intention of the programmer clearer to others who read the code.

Java Annotation
Annotations are metadata that is not part of the code. They do not affect execution. They are useful to compilers and other software tools, as well as humans who read the code. While we can similarly make the code more human-friendly with comments, an annotation is structured and so can be easily parsed by software. You will see 1-2 more useful annotations in this module.

this
The `this` keyword in Java that refers to the current object. In the example above, we use `this` to disambiguate the argument `center` and the field `center`. In general, it is a good practice to use `this` when referring the instance variable of the current object to make your intention clear.

Note that we can have other methods (such as `moveTo`) in the class beyond what is promised in the interface the class implements.

A class can implement more than one interface. For instance, let's say that we have another interface called `Printable` [^1] with a single method defined:

```
interface Printable {
  public void print();
}
```

The implementer of `Circle` wants to inform the clients that the method `void print()` is implemented, it can do the following:

```
class Circle implements Shape, Printable {
     :
     :
   @Override
   public void print() {
     System.out.printf("radius: %f\n", radius);
     System.out.printf("center:");
     center.print();
   }
}
```

In the above, we call `print()` on the `Point` object as well. How do we know that `Point` provides a `print()` method? Well, we can read the implementation code of `Point`, or we can agree with the implementer of `Point` that `Point` provides a `Printable` interface!

It is important to note that, `interface` provides a *syntactic* contract on the abstraction barrier, but it does not provide a *semantic* contract. It does not, for instance, guarantee that `print()` actually prints something to the screen. One could still implement interface `Printable` as follows:

```
class Circle implements Shape, Printable {
     :
     :
   @Override
   public void print() {
   }
}
```

and the code still compiles!

Not all programming languages that support classes support interface. C++, Javascript, and Python, for instance, do not support similar concepts.

Default Access Modifier for Interface
In the examples above, I explicitly specify the methods in the `Printable` and `Shape` interfaces as `public`. In Java, all methods in an interface are public by default, so the keywords `public` could be omitted.

Interface as Type

In Java, an interface is a type. What this means is that:

- We can declare a variable with an interface type, such as:

  ```
  Shape circle;
  ```

  or

  ```
  Printable circle;
  ```

  We cannot, however, instantiate an object from an interface since an interface is a "template", an "abstraction", and does not have an implementation. For instance:

  ```
  // this is not OK
  Printable p = new Printable();
  // this is OK
  Printable circle = new Circle(new Point(0, 0), 10);
  ```

- Similarly, we can pass arguments of an interface type into a method, and the return type of a method can be an interface.
- An object can be an instance of multiple types. Recall that Java is a statically typed language. We associate a type with a variable. We can assign a variable to an object if the object is an instance of the type of the variable. For example, Line 4 above creates a new circle, which is an instance of three types: `Circle`, `Shape`, and `Printable`. It is ok to assign this new circle to a variable of type `Printable`.

We say that `Shape` and `Printable` are *supertypes* of `Circle`, and `Circle` is a subtype of `Shape` and `Printable`.

Late Binding and Polymorphism

We can now do something cool like this:

```
Printable[] objs;
  :
  // initialize array objs
:
for (Printable o: objs) {
```

```
6       o.print();
7   }
```

Let's examine this code. Line 1 declares an array of objects of type `Printable`. We skip over the code to initialize the content of the array for now, and jump to Line 5-7, which is a `for` loop. Line 5 declares a loop variable `o` of type `Printable` and loops through all objects in the array `objs`, and Line 6 invokes the method `print` of `o`.

Array and For Loops in Java
See Oracle's tutorial on array [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html] and enhanced loop [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html]

The magic happens in Line 6:

- First, since we know that any object in the array has the type `Printable`, this means that they must implement the `Printable` interface and support the method `print()`.
- Second, we do not know, and we do not *need* to know which class an object is an instance of.
- Third, we can actually have objects of completely unrelated classes in the same array. We can have objects of type `Circle`, and objects of type `Point`. We can have objects of type `Factory`, or objects of type `Student`, or objects of type `Cushion`. As long as the objects implement the `Printable` interface, we can put them into the same array.
- Fourth, at *run time*, Java looks at the object `o`, and determines its class, and invokes the right implementation of `print()` corresponding to the `o`. That is, if `o` is an instance of a class `Circle`, then it will call `print()` method of `Circle`; if `o` is an instance of a class `Point`, then it will call `print()` method of `Point`, and so on.

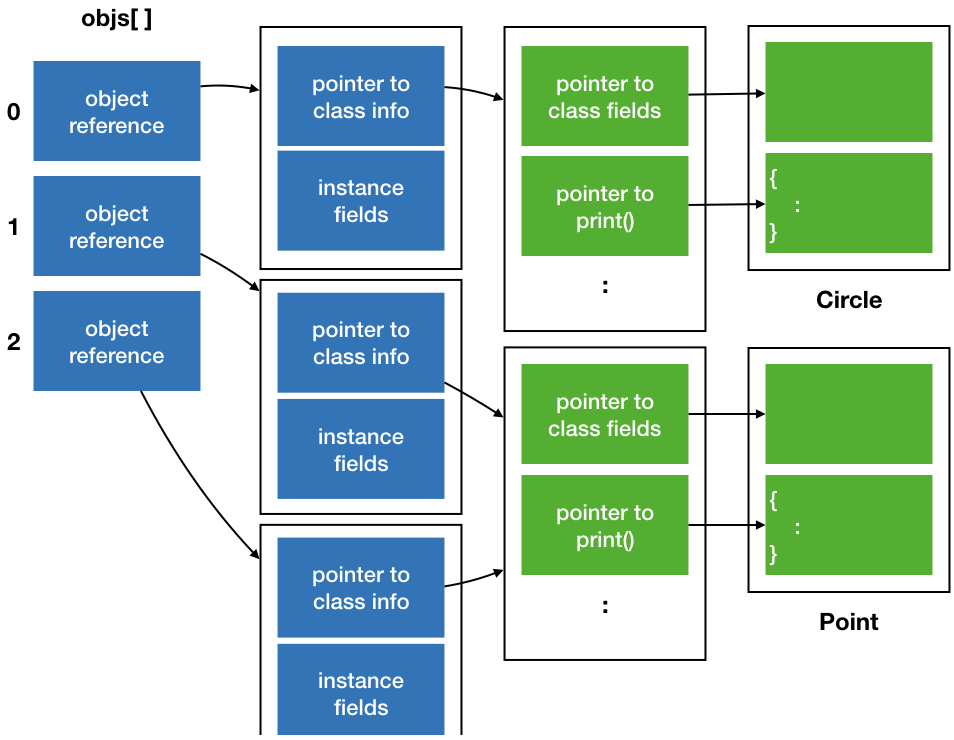
To further appreciate the magic in Line 6, especially on last point above, consider how a function call is done in C. In C, you cannot have two functions of the same name within the same scope, so if you call a function `print()`, you know exactly which set of instructions will be called [1 [#fn:2]]. So, the name `print` is bound to the corresponding set of instructions at compilation time. This is called *static binding* or *early binding*. To have `print()` for different types, we need to name them differently to avoid naming conflicts: e.g., `print_point()`, `print_circle()`.

In a language with static binding, suppose you want to mix objects of different types together in an array, you need to do something like the following pseudocode:

```
for each object in the array
    if object is a point
        print_point(object)
    else if object is a circle
        print_circle(object)
  else if object is a square
        print_square(object)
        :
        :
```

Not only is the code verbose and ugly, it would be cumbersome if you define a new compound data type that supports printing, since you would need to remember to add a new if-else condition to call for a corresponding print function.

In OO languages, you can have methods named `print()` implemented differently in different classes. When we compile the code above, the compiler will have no way to know which implementation will be called. The bindings of `print()` to the actual set of instructions will only be done at run time, after the object `o` is instantiated from a class. This is known as *dynamic binding*, or *late binding*, or *dynamic dispatch*.



If you understand how an object is represented internally, this is not so magical after all. Referring to the figure above, the array `objs[]` contains an array of references to objects, the first one is a `Circle` object, and the following two are `Point` objects. When `o.print()` is invoked, Java refers to the method table, which points to either the method table for `Circle` or for `Point`, based on the class the object is an instance of.

This behavior, which is common to OO programming languages, is known as *polymorphism* [2 [#fn:3]].

The Abstraction Principle

With the interface `Shape`, we can implement other classes, such as `Rectangle`, `Square`, `Polygon` with the same interface. For instance,

```
class Rectangle implements Shape, Printable {
   // left as exercise (See Exercise 2)
}
```

So far, we have been treating our shapes as pure geometric objects. Let's consider an application where we want to paint the shapes. Each shape should have a fill color and a border (with color and thickness).

```
import java.awt.Color;
    :

class PaintedCircle implements Shape, Printable {
  private Color  fillColor;
  private Color  borderColor;
  private double borderThickness;

  public void fillWith(Color c) {
    fillColor = c;
  }

  public void setBorderThickness(double t) {
    borderThickness = t;
  }

  public void setBorderColor(Color c) {
    borderColor = c;
  }

  // other methods and fields for Circle from before

}
```

In the code above, we added the line `import java.awt.Color` to use the Color class [https://docs.oracle.com/javase/8/docs/api/java/awt/Color.html] that Java provides, and added three private members as well as their setters.

We can do the same for `Triangle`

```
import java.awt.Color;
    :

class PaintedTriangle implements Shape, Printable {
  private Color  fillColor;
  private Color  borderColor;
  private double borderThickness;

  public void fillWith(Color c) {
    fillColor = c;
  }

  public void setBorderThickness(double t) {
    borderThickness = t;
  }

  public void setBorderColor(Color c) {
    borderColor = c;
  }

  // other methods and fields written for Triangle

}
```

and for other shapes.

Great! We now have colorful shapes. The code above, however, is not *good* code, even though it is *correct*. Just consider what needs to be done if we decide to support border styles (dotted border, solid border, dashed border, etc). We would have to edit every single class that supports colors and borders!

One principle that we can follow is the *abstraction principle*, which says "Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts." [3] [#fn:4]

Following the principle, we want to implement these style-related fields and methods in just one place. But where?

Inheritance

The OO-way to do this is to create a *parent class*, and put all common fields and methods into the parent. A parent class is defined just like a normal class. For instance:

```
class PaintedShape {
  private Color  fillColor;
  private Color  borderColor;
  private double borderThickness;

  public PaintedShape(Color fillColor, Color borderColor, double borderThickness) {
    this.fillColor = fillColor;
    this.borderColor = borderColor;
    this.borderThickness = borderThickness;
  }

  public void fillWith(Color c) {
    fillColor = c;
  }

  public void setBorderThickness(double t) {
    borderThickness = t;
  }

  public void setBorderColor(Color c) {
    borderColor = c;
  }
}
```

The `PaintedCircle` class, `PaintedSquare` class, etc, can now *inherits* non-private fields and methods from the parent class, using the `extends` keyword.

```
class PaintedCircle extends PaintedShape implements Shape, Printable {
```

```
        :
}

class PaintedSquare extends PaintedShape implements Shape, Printable {
        :
}
```

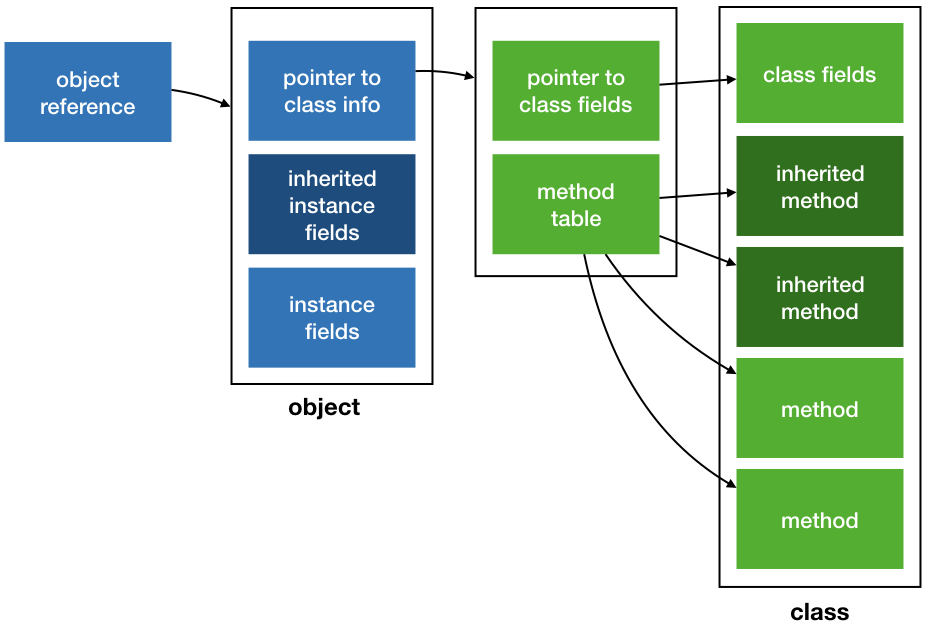
This mechanism for a class to inherit the properties and behavior from a parent is called *inheritance*, and is the fourth and final basic OO principles we cover [4 [#fn:5]].

With inheritance, we do not have to repeat the declaration of fields `fillColor`, `borderColor`, `borderThickness` and the associated methods in them. This software engineering principle is also known as the DRY principle -- "*don't repeat yourself*" principle. We are going to see DRY very regularly in future lectures.

We also call the `PaintedShape` the superclass (or base class) of `PaintedCircle` and `PaintedSquare`, and call `PaintedCircle` and `PaintedSquare` the subclass (or derived class) [5 [#fn:6]] of `PaintedShape`.

A `PaintedCircle` object can now call `fillWith()` even if the method `fillWith()` is not defined in `PaintedCircle` -- it is defined in `PaintedCircle`'s parent `PaintedShape`.

When a class extends a parent class, it inherits all the non-private fields and methods, so we can depict the objects and the class as follows:



The method table now includes pointers to methods defined in the parent (and grandparents, and so on).

Overloading

Now consider the constructor for `PaintedCircle`. We need to initialize the geometric shape as well as the painting style. But, we define the fields `fillColor`, etc `private`, and thus subclasses have no access to `private` fields in the parent. We need to call the constructor of the parent to initialize these private fields. The way to do this is to use the `super` keyword, as such:

```
  public PaintedCircle(Point center, double radius, Color fillColor, Color borderColor, double borderThickness) {
    super(fillColor, borderColor, borderThickness);
    this.center = center;
    this.radius = radius;
  }
```

You can see that the constructor for `PaintedCircle` now takes in five parameters. You can imagine that as the class gets more sophisticated with more fields, we need to pass in more parameters to the class to initialize the fields. It is not uncommon to provide alternative constructors with fewer parameters and assign some *default* values to the fields.

```
  // create circle with default style (white with black border of thickness 1)
  public PaintedCircle(Point center, double radius) {
    super(Color.WHITE, Color.BLACK, 1.0);
    this.center = center;
    this.radius = radius;
  }

  // create circle with customized styles
  public PaintedCircle(Point center, double radius, Color fillColor, Color borderColor, double borderThickness) {
    super(fillColor, borderColor, borderThickness);
    this.center = center;
    this.radius = radius;
  }
```

Two methods in a class can have the same name and still co-exist peacefully together. This is called *overloading*. When a method is called, we look at the *signature* of the method, which consists of (i) the name of the method, (ii) the number, order, and type of the arguments, to determine which method is called. To be precise, the first sentence of this paragraph should read: Two methods in a class can have the same name and still co-exist peacefully together, as long as they have different signatures. Note that the return type is not part of the method signature, so you cannot have two methods with the same name and same arguments but different return type.

Even though the example above shows overloading of the constructor, we can overload other methods as well.

Exercise

1. Consider what happen when we do the following:

```
Circle c = new Circle(new Point(0,0), 10);
Shape c1 = c;
Printable c2 = c;
```

   Are the following statements allowed? Why do you think Java does not allow some of the following statements?

```
c1.print();
c2.print();
c1.getArea();
c2.getArea();
```

2. Write another class `Rectangle` that implements these two interfaces. You should make use of the class `Point` that you implemented from Lecture 1's exercise. Then write another class `PaintedRectangle` that implements the two interfaces and inherits from `PaintedShape` that implements the two interfaces and inherits from `PaintedShape`. You can assume that the sides of the rectangles are parallel with the x- and y-axes (in other words, the sides are either horizontal or vertical).

3. (i) Write an interface called `Shape3D` that supports a method `getVolume`. Write a class called `Cuboid` that implements `Shape3D` and has three private `double` fields `length`, `height`, and `breadth`. `getVolume()` should return the volume of the `Cuboid` object. The constructor for `Cuboid` should allow the client to create a `Cuboid` object by specifying the three fields `length`, `height` and `breadth`.

   (ii) We can extend one interface from another as well. Find out how, and write a new interface `Solid3D` that inherits from interface `Shape3D` that supports a method `getDensity` and `getMass`.

   (iii) Now, write a new class called `SolidCuboid` with an additional private `double` field `density`. The implementation of `getDensity()` should return this field while `getMass()` should return the mass of the cuboid. The `SolidCuboid` should call the constructor of `Cuboid` via `super` and provides two constructors: one constructor allows the client to specify the density, and the other does not and just set the default density to 1.0.

4. Write a class `Rectangle` that implements `Shape`. A `Rectangle` class has two `double` fields, `length` and `width`, and a public method `setSize(int length, int width)` that allows the client to change its size.

   Now, write a class `Square` that inherits from `Rectangle`. A `Square` has an additional constraint that `length` must be the same as `width`. How should `Square` implement the `setSize(int length, int width)` method? Do you think `Square` should inherit from `Rectangle`? Or should it be another way around? Or maybe they should not inherit from each other?

   (Note: to implement the `contains` method in `Shape`, you need to also keep the position of the `Square` (top left corner, for instance). But, it is not important for this question)

---

1. Remember a function is just an abstraction over a set of instructions.
2. In biology, polymorphism means that an organism can have many different forms.
3. This principle is formulated by Benjamin C. Pierce in his book *Types and Programming Languages*.
4. The other three is encapsulation, abstraction, and polymorphism.
5. Again, you see that computer scientist can be quite indecisive when it comes to the terminologies in OOP.