Object Oriented Programming

Sujit Chakrabarti

1 Inheritance

Let's return to our shape classes by implementing a `Rectangle` class.

```
class Rectangle {

  protected final float length;
  private final float breadth;

  public Rectangle(float l, float b) {
    this.length = l;
    this.breadth = b;
  }

  public float area() {
    return this.length * this.breadth;
  }

}
```

Figure 1: A rectangle class

The code in fig. 1 can be tested as follows:

```
public class Geometry2 {

  public static void main(String[] a) {
    Rectangle r = new Rectangle(10, 20);
    System.out.println("area = " + r.area());
  }
}
```

Next, let's also add a `Square` class to the code.

```
class Square {

  private final float length;
  public Square(float l) {
    this.length = l;
  }

  public float area() {
    return this.length * this.length;
  }
}
```

Figure 2: A square class

The code in fig. 2 can be tested as follows:

```
public class Geometry2 {

  public static void main(String[] a) {
    Rectangle r = new Rectangle(10, 20);
    System.out.println("area = " + r.area());
    Square s = new Square(200);
    System.out.println("area = " + s.area());
  }
}
```

The `Rectangle` and `Square` classes fig. 1 and fig. 2 aren't similar by coincidence. The fact is, a square is a rectangle with its length equal to its breadth. Unfortunately, the code here doesn't capture this fact. It would be nice if we could make this knowledge an explicit part of our code. Does Java allow us to do that? Yes, it does!

```
class Square extends Rectangle {

  private final float length;
  public Square(float l) {
    this.length = l;
  }

  public float area() {
    return this.length * this.length;
  }
}
```

Figure 3: `Square` class declared a sub-class of `Rectangle`

The code in fig. 3 modifies that in fig. 2 by declaring `Square` as a sub-class of `Rectangle`, by using the `extends` keyword as shown. This is good. But it doesn't do much functionally. However, following version of the `Square` class does the real magic!

```
class Square extends Rectangle {
  public Square(float l) {
    super(l, l);
  }
}
```

Figure 4: `Square` class declared a sub-class of `Rectangle`

Several lines from the `Square` have been reduced. Does this code even compile? Sure enough, it does! In particular, the call to `s.area()` from the `main` method compiles in spite of there being no `area` method anymore in the `Square` class. And if you run the code, it seems to work just as fine as before, giving the same result. How could this happen?

The reason for this is: `Square` being the *child* of the `Rectangle` class, *inherits* all its properties. This is called *inheritance*, the most important feature of all object oriented programming languages. We say that:

- `Square` inherits from `Rectangle`.
- `Square` is the sub-class/child-class/sub-type/child-type/derived-class of `Rectangle`.

- `Rectangle` is the super-class/parent-class/super-type/parent-type of `Square`.

Thus, `area` method in `Rectangle` also becomes a property of `Square`. But this `area` method needs the `length` and `breadth` attributes of `Rectangle` to be set to the correct values (in this case, equal to the `length` attribute (now deleted) of the `Square` class). Where and how does this happen?

It happens in the constructor of `Square` class, when `super(l, l)` executes. To understand what this does consider the pictures in fig. 5.

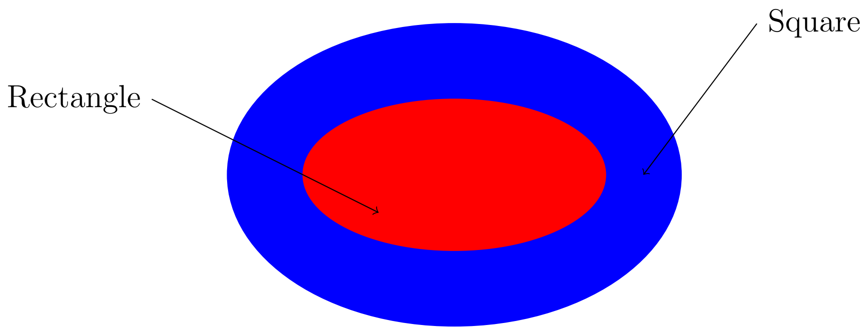


Figure 5: `Square` object with an object of `Rectangle` embedded within itself

Each object of a sub-class can be visualised as embedding within it an object of its super-class. Thus, an object of `Square` class has within it an object of `Rectangle` as shown in fig. 5. During the construction of a sub-class object, the first step to complete is the construction of the embedded super-class object. The call to `super` method does precisely this. It calls the constructor of the super-class with the given arguments, in this case `l` and `l` (`l` being the parameter to `Square`'s constructor).

This initialises the embedded `Rectangle` to have its `length` and `breadth` attributes both set to the argument passed to the constructor to `Square`. Thus, a subsequent call to `s.area` in the `main` method calls the `area` method of the embedded `Rectangle` object. This, in turns returns the product of `length` and `breadth` attributes (which, remember, are equal to each other) thus giving us the correct `area` of the `Square`.

So, `super` is a new keyword we have learned, it used in the context of a sub-class, is essentially a reference to the embedded instance of the superclass.

Can we derive further classes from `Square`? Yes, and we present an example in fig. 6.

```
class Point extends Square {
  public Point() {
    super(0);
  }
}
```

Figure 6: `Square` class declared a sub-class of `Rectangle`

Again, a very rudimentary class, with hardly any code! It just creates the new type `Point` as a sub-type of `Square`, making point a special type of square with zero length, which is indeed a reasonable way to look at things.

The code in fig. 6 can be tested with the following lines added to the `main` method:

```
    Square p = new Point();
    System.out.println("area = " + p.area());
```

This works just smoothly, giving us the expected results:

```
area = 0.0
```

Indeed, a point is a square (and hence a rectangle) with zero area. Pictorially, the scenario can be depicted as in fig. 7.

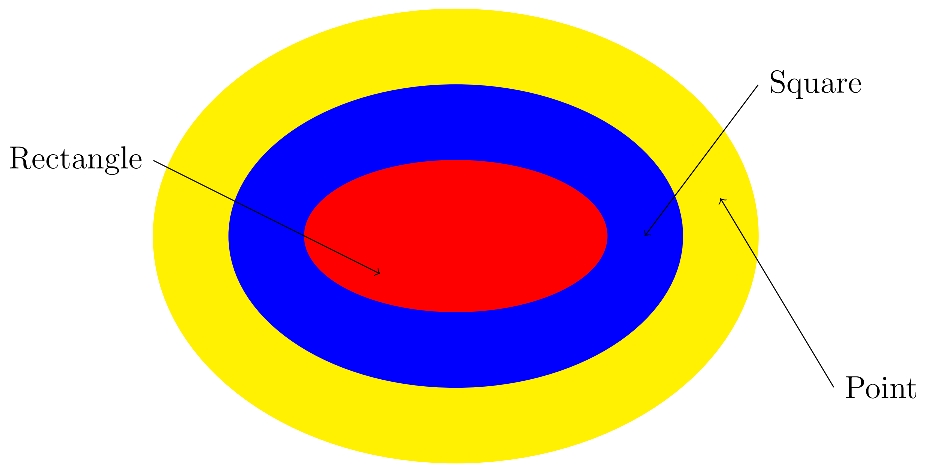


Figure 7: `Point` object with an object of `Square` embedded within itself

1.1 Access Specifier `protected`

Any method from within `Square` class won't be able to access `Rectangle`'s `length` attribute directly. This may be OK sometimes, but sometimes this may be too restrictive. For example, the following piece of code

```
  public float circumference() {
    return 4.0f * this.length;
  }
```

if added to the `Square` class would lead to a compilation error:

```
error: length has private access in Rectangle
    return 4.0f * this.length;
```

In other words, the designer of the `Rectangle` class may want the implementers of its sub-classes to have direct access to `length`. One option would be to turn `length` into a `public` attribute. This would work, but this is an overkill, and too permissive. We would like to tune the visibility of `length` to just the level where the sub-classes have direct access to it, but it remains invisible to any other class in the program. For this we use the `protected` access specifier:

```
  protected float length;
```

With this, it is possible to write code within `Square` class that directly refers to `length`. For instance, the code added to `Square` above, now works!

2 Polymorphism

```
  public static void main(String[] a) {
    Rectangle r = new Rectangle(10, 20);
    System.out.println("area = " + r.area());
    r = new Square(10);
    System.out.println("area = " + r.area());
    r = new Point();
    System.out.println("area = " + r.area());
  }
```

Figure 8: `Square` class declared a sub-class of `Rectangle`

Consider the modified `main` method shown in fig. 8. The notable point here is that the variable `r`, which is of the type `Rectangle` is first initialised to a `Rectangle`, which is familiar. However, subsequently, we assign to it an instance of a `Square`, and then a `Point`. We print the area in each case

The output of running the code in fig. 8 is as shown below:

```
area = 200.0
area = 100.0
area = 0.0
```

... And look, the area gets printed correctly for all the three shapes.

At this point, a slight refinement of terminology. Here, rather than thinking of `r` as a variable of type `Rectangle`, it's more proper to think of it as a reference of the type `Rectangle`. This means that it can point to an object of the type `Rectangle`. What we observe further in the code in fig. 8 is that it is allowed for `r` to point to any object whose type is a sub-type of `Rectangle`. In fact, references in Java are called polymorphic.

... And this property of a language which implements polymorphic references in the above sense is called *polymorphism*, more precisely, *dynamic polymorphism*. We discuss the meaning of this term a little later. But let's try to appreciate what this feature can do for us. Consider the modified driver code shown in fig. 9. The `main` calls another method `printRectangles`. As argument, it passes an array of `Rectangle`s constructed out of three `Rectangles`: `r` (indeed a `Rectangle`), `s` (actually, a `Square`) and `p` (which is in fact a `Point`). Firstly, note that Java allows us to construct an array of `Rectangle`s, wherein the elements can be objects of any sub-class of `Rectangle`. Secondly, passing this array to `printRectangles` gives us just the expected output: the areas of all the `Rectangle`s in the array getting printed.

What does this mean? This means that `printRectangles` method couldn't care less what the precise type of the objects in the `rarray` array are. The Java type system assures that they all are instances of `Rectangle` or one of its sub-classes. In fact, there's no need for `Square` and `Point` classes to even exist at the time of implementing `printRectangles`. Even if these classes are implemented afterwards – much after the time `printRectangles` is implemented – everything here is guaranteed to work perfectly.

```
public static void main(String[] a) {
  Rectangle r = new Rectangle(10, 20);
  Square s = new Square(200);
  Square p = new Point();

  Rectangle[] rarray = {r, s, p };
  printRectangles(rarray);
}

public static void printRectangles(Rectangle[] rarray) {
  for(Rectangle rec : rarray) {
    System.out.println("area = " + rec.area());
  }
}
```

Figure 9: `printRectangles` method prints an array of `Rectangles`

The above idea is not new, but has existed for a long time in engineering. Wherever there is a system with components that interact and interoperate, engineers go about designing them by defining what we call interfaces. Consider the USB port, the VGA, power audio ports of your computer. As long as a VGA cord following the specifications of VGA is inserted into your computer's VGA port, it is kind of guaranteed to work. It doesn't matter who manufactured the VGA cord. Similarly, the Android OS can be installed on any Android compatible device. It could be any of hundreds of phone brands, it could be a tablet, a PC, a TV or anything else. Are these devices identical? No. But they follow the interfaces specified by the creators of Android OS. Internally, each one of them may have many variations, but Android doesn't concern itself with them.

Similarly, the super-class (here, `Rectangle`) is kind of an interface which the `printRectangles` method accepts. The inheritance rules of Java guarantee that all sub-classes of `Rectangle` adhere to its interface, i.e. if `area` method is called on them, it will be available. And therefore, `printRectangle` is able to work with any array of `Rectangles`, even when it may actually contain objects of other types. All that's needed is those other types must be sub-types of `Rectangle`. Java's type-system makes sure that requirement is fulfilled: an attempt to populate a `Rectangle[]` array with an object of a type which isn't a sub-type of `Rectangle` will fail at compile-time.

In a short while, we will have a bit more to say about interfaces, which are a very important concept in Java and OOP in general.

3 Method Overriding

What we have learned so far about inheritance is good to create sub-classes which are specialisations of their super-classes. In other words, they are the same as their super-classes, but for some additional constraints. This is useful, but not useful enough. Often there are situations when we wish to modify our super-classes as per need. I will present here a simple example.

Let's add a method `printName` in the rectangle class:

```
  public void printName() {
    System.out.println("I'm a rectangle.");
  }
```

... and let's call this function from the `main` for all the `Rectangle`s we have created there.

```
    Rectangle[] rectangles = { r, s, p };
    for(Rectangle rec : rectangles) {
      rec.printName();
    }
```

This will produce the following output:

```
I'm a rectangle.
I'm a rectangle.
I'm a rectangle.
```

... which is technically correct, but not interesting. It would be nice if we could print the correct name as per the sub-class. For example, for a `Square`, the message should be `"I'm a square."`. Is it possible to have this output? Given the fact that in the context of `main`, each shape is being accessed through a `Rectangle` type reference, this looks unlikely. Nevertheless, let's go ahead and implement the methods that we would have liked to be called to print the correct shape names.

In `Square` class, we add:

```
  public void printName() {
    System.out.println("I'm a square.");
  }
```

In `Point` class, we add:

```
  public void printName() {
    System.out.println("I'm a point.");
  }
```

... hardly any hope that this would be useful, because `main` treats them all as `Rectangles`. Morosely, we compile the code and run it:

```
I'm a rectangle.
I'm a square.
I'm a point.
```

Woah! Magic! Looks like, for each object, the version of `printName` as defined in the sub-class was called. Indeed, that's what happened. Even though `printDetails` is called from the context of `main`, with a reference to the `Rectangle` class, the implementations in the sub-classes are called. In fact, it's quite allowed to implement the `main` method in a separate source file, and compile it even before the sub-classes like `Square` and `Point` are written. These can be written and added to the program later, and yet, everything would work seamlessly. This feature is realised with a mechanism called *dynamic dispatch*.

To make things further interesting, let's implement a method named `printDetails` in the `Rectangle` class along with the `main` method as shown in fig. 10.

```
  public void printDetails() {
    this.printName();
    System.out.println("... and my area is " + this.area());
  }
```

Figure 10: `printDetails` method prints the details of the object.

Note that `printDetails` has a call to `this.printName`. If this method is called on a sub-class of `Rectangle` using a `Rectangle` reference, which version of `printName` would be called? `Rectangle`'s or the sub-class's? To test, we also modify the driver code as follows:

```
    Rectangle r = new Rectangle(10, 20);
    Square s = new Square(200);
    Square p = new Point();

    Rectangle[] rectangles = { r, s, p };
    for(Rectangle rec : rectangles) {
      rec.printDetails();
    }
  }
```

When we compile and run the modified code, we get the following output:

```
I'm a rectangle.
... and my area is 200.0
I'm a square.
... and my area is 40000.0
I'm a point.
... and my area is 0.0
```

Sure enough, and suprisingly, the correct versions of `printName` gets called from within `Rectangle.printDetails`.

This is one of the most powerful features of object-oriented programming and should be mastered well by an object-oriented programmer. Let's say, in a class C_1 a method m_1 calls another m_2 in the same class, which has implementations in sub-class C_2. Now, if a reference r points to an instance of C_1 and a call $r.m_1$ is called. Internally, $r.m_1$ will call $C_2.m_2$. Because of this, we say that $C_2.m_2$ *overrides* $C_1.m_2$.

4 Abstract Classes

Let's add another class `Circle` into our family of classes as shown in fig. 11.

```
class Circle {
  private float radius;
  public static final float PI = 3.141f;

  public Circle(float r) {
    radius = r;
  }
}
```

Figure 11: `Circle` class.

If we try to instantiate a `Circle` in the `main` and try to refer to it using an `Rectangle` reference, this will lead to a compilation error: the types simply don't match.

The first cut solution is very simple. Define a class `Shape` as the super-class of both `Circle` and `Rectangle`. Let's do it.

```
class Shape {
  public printName() {
    "I'm a shape.";
  }

  public float area() {
    return 0;
  }

  public void printDetails() {
    this.printName();
    System.out.println("... and my area is " + this.area());
  }
}

class Circle extends Shape {
...

class Rectangle extends Shape {
...
```

Figure 12: `Circle` class.

In `Shape`, we provide a default definition of the `area` method and `printName` methods. Note that these implementations don't make much sense. The `printName` method doesn't provide adequate information about the `Shape`, and `area` method would simply give a wrong result for all but `Shape`s with zero area. Nevertheless, they are needed. Otherwise, the code will not compile.

The `main` methods gets modified as follows to accommodate the above:

```
  public static void main(String[] a) {
    Rectangle r = new Rectangle(10, 20);
    Square s = new Square(200);
    Square p = new Point();
    Circle c = new Circle(10);

    Shape[] shapes = { r, s, p, c };
    for(Shape sh : shapes) {
      sh.printDetails();
    }
  }
```

When we compile and run the above code, we get the following output:

```
I'm a rectangle.
... and my area is 200.0
I'm a square.
... and my area is 40000.0
I'm a point.
... and my area is 0.0
I'm a shape.
... and my area is 0.0
```

Note that the outputs corresponding to the `Circle c` are wrong. In case of the `Circle`, it was the `Shape`'s implementation of `printName` and `area` that get called. This is not merely undesirable, but completely wrong.

On looking at the problem a little more closely, we realise the following:

1. We have forgotten to implement `printName` and `area` methods in `Circle` class.

2. The default implementation of `printName` and `area` methods provided in `Shape` class are really unnecessary, and are doing more harm than good. They have been put there just to satisfy the compiler (which is a very bad reason to implement a method). Not surprisingly, they are becoming the source of a bug which could be quite hard to detect.

What's the nature of this bug? This bug happens whenever there are default implementation of methods provided in the super-class which are necessarily supposed to implemented in the sub-classes, and somehow the implementer of the sub-class forgets to provide one. What we really wish is:

1. **Wish 1.** No enforcement to provide useless dummy implementations of methods in the super-classes just to appease the compiler.

2. **Wish 2.** If we forget to implement such methods in the sub-classes, the compiler should alert us of our mistake.

Can we design our code to get the above? Yes, we can. By using abstract classes. To fulfil *wish 1* above, we make the following change to the `Shape` class:

```
abstract class Shape {
  public abstract void printName();
  public abstract float area();
  public void printDetails() {
    this.printName();
    System.out.println("... and my area is " + this.area());
  }
}
```

Figure 13: `Circle` class.

The modified code in fig. 13 declares `Shape` as an `abstract` class: a class with one or more `abstract` methods. Abstract methods are methods which have a declaration in the class, but have not been implemented. In fig. 13, `printName` and `area` methods are `abstract` methods of `Shape`.

A very important characteristic of abstract classes is that they can't be instantiated directly. That is, an attempt to have something like `Shape s = new Shape()` in the program would not be accepted by the compiler. The only way to instantiate abstract classes is through their *concrete* (which are not abstract themselves) sub-classes.

OK. So, we try to compile the above code, and get the following compiler error:

```
error: Circle is not abstract and does not override
 abstract method area() in Shape
class Circle extends Shape {
^
```

This says that `Circle`, which is not declared `abstract` doesn't provide necessary implementation for the abstract methods of its super-class `Shape`. This takes care of *wish 2* above: the compiler has pointed us out our mistake of having forgotten to provide implementations for `printName` and `area` methods in `Circle` class.

To correct this, we add the implementations of `printName` and `area` methods in the `Circle` class as shown in fig. 14.

```
class Circle extends Shape {
  private float radius;
  public static final float PI = 3.141f;

  public Circle(float r) {
    radius = r;
  }

  public void printName() {
    System.out.println("I'm a circle.");
  }
  public float area() {
    return PI * radius * radius;
  }
}
```

Figure 14: `Circle` class with `printName` and `area` methods implementated.

Now the code compiles and on running it, we get the following output:

```
I'm a rectangle.
... and my area is 200.0
I'm a square.
... and my area is 40000.0
I'm a point.
... and my area is 0.0
I'm a circle.
... and my area is 314.1
```

... which is exactly what we were looking for. Note that the errors in the last two lines of the output have now been corrected.

5 Interfaces

A possible design of the `Shape` class would to make it completely abstract, i.e. when all its methods are abstract, as shown in fig. 15.

```
abstract class Shape {
  public void printName();
  public float area();
}
```

Figure 15: A completely abstract `Shape` class.

Java presents another syntactic way to define such completely abstract classes. They are called interfaces, as shown in fig. 16.

```
interface Shape {
  void printName();
  float area();
}

class Circle implements Shape {
  ...
}

class Rectangle implements Shape {
  ...
}
```

Figure 16: `Shape` as an interface.

Interfaces are more significant than just being able to build completely abstract classes. This fact will gradually reveal itself over the discussion. But, let's get some other syntax out of our way. An `interface` is not extended, but implemented, by its sub-classes, as shown in the modified code for `Circle` and `Rectangle`.

5.1 Interfaces – Another Example

Let's take a break from our shapes example to discuss something else.

We wish to implement a function that takes an integer array as an input and returns an integer array of the same length but each of its elements double the corresponding element in the input array. One possibly reasonable implementation is shown in fig. 17.

```
public class Hof {

        public static void main(String[] args) {
                int[] a = { 1, 2, 3};
                printArray(doubleArray(a));
        }

        private static int dbl(int n) {
                return n * 2;
        }

        private static int print(int n) {
                System.out.print(n + " ");
                return 0;
        }

        private static int[] doubleArray(int[] array) {
                int[] ans = new int[array.length];
                for(int i = 0; i < array.length; i++) {
                        ans[i] = dbl(array[i]);
                }
                return ans;
        }

        private static int[] printArray(int[] array)  {
                int[] ans = new int[array.length];
                for(int i = 0; i < array.length; i++) {
                        ans[i] = print(array[i]);
                }
                return ans;
        }
}
```

Figure 17: A Java program implementing a doubling an array.

On running this program, we get the following output:

```
2 4 6
```

Now, we additionally wish to implement a function that squares an integer array, we may have to insert the pieces in fig. 18.

```
public class Hof {

        public static void main(String[] args) {
                int[] a = { 1, 2, 3};
                printArray(doubleArray(a));
                printArray(squareArray(a));
                System.out.println("");
        }

    // rest of the code here

        private static int square(int n) {
                return n * n;
        }

        private static int[] squareArray(int[] array) {
                int[] ans = new int[array.length];
                for(int i = 0; i < array.length; i++) {
                        ans[i] = square(array[i]);
                }
                return ans;
        }
}
```

Figure 18: A Java program implementing a squaring an array.

On running this program, we get the following output:

```
2 4 6
1 4 9
```

However, Java does badly when it comes to economy of expression. Let's just look at how the same code would look in another programming language, Python, shown in fig. 19

```
print map(lambda x: 2 * x, [1, 2, 3])
print map(lambda x: x * x, [1, 2, 3])
```

Figure 19: A Python program implementing a doubling and squaring an array.

```
[2, 4, 6]
```

```
[1, 4, 9]
```

One of the reasons for this enormous of economy of syntax is a *higher order function* called `map`. A higher order function is a function that takes other functions as inputs (and possibly computes functions as output). `map` is one such function that takes a function, say `f`, and a list, say `l`, and returns a list each of whose elements is `f` applied to the corresponding element in `l`. HOFs are a common feature of functional programming. In fact, what inheritance and polymorphism is to object-oriented programming, HOFs are the same to functional programming: the main abstraction mechanism for code reuse and modularisation.

Unfortunately, Java (until Java 8) doesn't have higher order functions. But the case for higher order functions is strong enough that we wish to implement something similar in Java. How do we do it?

With our knowledge of object-oriented programming, we can come out with a fairly close approximation to the idea of higher order functions. The central idea is: in OOP, we can't pass functions to functions, but we can pass objects which have functions in them. Thus, all we need to do is pack the function that we want to pass to a HOF in a class, and pass an object of that class to the HOF. The solution for the problem is presented in fig. 20.

```java
public class Hof {
  public static void main(String[] a) {
    int[] ar = {1, 2, 3};

    int[] newarray = Hof.map(new Dbl(), ar);
    for(int n : newarray) {
      System.out.println(n);
    }
    newarray = Hof.map(new Dbl(), ar);
    for(int n : newarray) {
      System.out.println(n);
    }
  }

  public static int [] map(Function f, int [] a) {
    int[] newarray = new int[a.length];
    for(int i = 0; i < a.length; i++) {
      newarray[i] = f.execute(a[i]);
    }
    return newarray;
  }
}

interface Function {
  int execute(int n);
}

class Dbl implements Function {
  public int execute(int n) {
    return 2 * n;
  }
}

class Sqr implements Function {
  public int execute(int n) {
    return n * n;
  }
}
```

Figure 20: A Java program implementing map higher order function.

which gives us the following output:

```
2
4
6
1
4
9
```

Just in case it's not clear already, let me help you walk through the code.

1. The first thing to notice is the interface `Function`. It has a method declaration `execute` that takes an integer and returns an integer. This represents the function that would be passed to the higher order function map and would be applied to each element of the integer array.

2. The next thing to observe is the `map` method which takes two parameters: the first is `f`, an instance of `Function`; and the other is the array of `ints a`. It returns the resultant array after applying `f` on each element of `a` as shown.

3. Finally, an instance of `Dbl` and `Sqr` are passed in turn to `Hof.map` in the `main` method to be applied to `array`.

This application is to demonstrate how object-oriented principles can be used to create highly reusable code. The `map` method here is a boiler-plate code that can be used wherever there's a need to apply a function on all elements of an integer array to create a new one. What that function to applied should be is a parameter to the method, and hence is flexible.

6 Anonymous Classes

The next question we ask is: In Java, is there something equivalent to lambda expressions as seen in Python. Lambda expressions are like anonymous functions which are typically created for a one time use as a parameter to a higher order function. This avoids creation of extra named functions and saves the namespace (variable environment) from unnecessary clutter. Here, the classes `Dbl` and `Sqr` are similar classes: to be instantiated once to be passed to `Hof.map`, and then to be forgotten. Why do we need to create a named class which will ever be implemented only once? Like anonymous functions (lambda) in Python, do we have something like anonymous classes? Indeed, yes!

We replace the earlier `Hof.main` method with the lines shown in fig. 21.

```
public static void main(String[] a) {
  int[] ar = {1, 2, 3};
  Function triple = new Function() {
      public int execute(int n) {
        return 3 * n;
      }
    };
  newarray = Hof.map(triple, ar);

  for(int n : newarray) {
    System.out.println(n);
  }
}
```

Figure 21: Anonymous class.

Here's the output:

```
3
6
9
```

Here, we used the `map` method to triple the `array`. The `Function` interface is implemented through an anonymous class (a class with no name) set to `Function triple`. All we need to do to make it a concrete class is to provide its own implementation of the `execute` method. We do so. And it just works smoothly. No need to create another class named `Triple` or something. No too many unused class names. No namespace clutter!

7 Multiple Inheritance

To introduce our next topic, let's consider an example of an institute with an application for its functioning. And this app has a class called `Professor`. By the way, for a reason that'll shortly become clear, we use another object oriented programming language C++ to implement our program. To begin with, the code is as shown in fig. 22.

```
#include <iostream>
#include <string>

using namespace std;
class Professor {
private:
  const string name;
public:
  Professor(string n) : name(n) {}
  void teach(string course, string cl) {
    cout << "Prof. " << name << " teaches course "
      << course << " to class " << cl << "." << endl;
  }
};

int main() {
  Professor prof("Sujit");
  prof.teach("Java", "MT2017");
}
```

Figure 22: Institute program.

Since an institute must also have a warden, we add a `Warden` class to the above code as shown in fig. 23

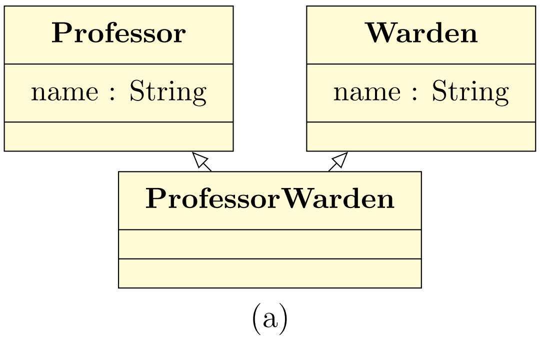
```
class Warden {
private:
  const string name;
public:
  Warden(string n) : name(n) {}
  void approve(string request) {
    cout << "Warden " << name << " approves request "
      << request << "." << endl;
  }
};

int main() {
  Professor prof("Sujit");
  prof.teach("Java", "MT2017");

  Warden warden("Sujit");
  warden.approve("Dinner out");

  return 0;
}
```

Figure 23: `Warden` class.

The above program gives us the following output:

```
Prof. Sujit teaches course Java to class MT2017.
Warden Sujit approves request Dinner out.
```

Now, let's say that the institute decides that it would wish one of its professors to double up as a warden. This indicates towards the creation of a new class `ProfessorWarden` with functionalities of a `Professor` and `Warden`, as shown pictorially in fig. 24. Let's do so, as shown in fig. 27.



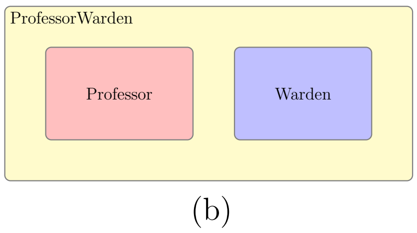


Figure 24: (a) `ProfessorWarden` as a common sub-class of both `Professor` and `Warden`; (b) A `ProfessorWarden` object with `Professor` and `Warden` objects embedded within.

```
class ProfessorWarden : public Professor, public Warden {
public:
  ProfessorWarden(string n) : Professor(n), Warden("ditto") {}
  void doWork() {
    cout << "Prof. " << Professor::name
      << " teaches and approves requests." << endl;
  }
};

int main() {
  Professor prof("Sujit");
  prof.teach("Java", "MT2017");

  Warden warden("Sujit");
  warden.approve("Dinner out");

  ProfessorWarden profwarden("SKC");

  Professor *profptr = &profwarden;
  Warden * wardenptr = &profwarden;

  profptr->teach("Python", "IMT2017");
  wardenptr->approve("party request");

  ProfessorWarden * profwardenptr = &profwarden;
  profwardenptr->teach("Programming Languages", "IMT2013");
  profwardenptr->approve("Cultural festival request");
  profwardenptr->doWork();

  return 0;
}
```

Figure 25: `ProfessorWarden` class.

The code in fig. 27 gives us the following output:

```
Prof. Sujit teaches course Java to class MT2017.
Warden Sujit approves request Dinner out.
Prof. SKC teaches course Python to class IMT2017.
Warden ditto approves request party request.
Prof. SKC teaches course Programming Languages to class IMT2013.
Warden ditto approves request Cultural festival request.
Prof. SKC teaches and approves requests.
```

The kind of inheritance shown in fig. 24 and fig. 27 – where a class (ProfessorWarden) is a sub-class of two super-classes (Professor and Warden) – is called *multiple inheritance*. Multiple inheritance allows re-use of the properties of multiple classes by defining a common sub-class.

Note that in main(fig. 27), the pointers profptr and wardenptr and profwardenptr point to the same instance of ProfessorWarden. Through, profptr, it's possible to call teach method; through wardenptr, we can call approve, and through profwardenptr, we can call all three methods – Professor::teach, Warden::approve and ProfessorWarden::doWork.

The implication of this is that any client code, which uses Professor will work just as well as before with an instance of ProfessorWarden. Same is the case with client code using Warden. Hence, ProfessorWarden class is 'backward compatible' with Professor and Warden.

Note that not everything is perfect in the above code. The warden is named ditto, an absurd default value we have used to initialise the Warden part of the ProfessorWarden. We could get around the problem by initialising Warden::name by the same value as Professor::name. However, that's not a complete fix. The fact that there are really two name attributes in a ProfessorWarden – one as Warden::name and the other as Professor::name – is not quite a very elegant thing to have. Why should any object have two names like this, one of which gets arbitrarily chosen to be ignored because it is there accidentally as a side-effect of multiple inheritance? This point becomes further important in the next section.

8 The Diamond Problem

How would we implement the multiple inheritance scenario of fig. 24 in Java? An intuitive syntax may be:

```
class ProfessorWarden extends Professor, Warden {
  ...
```

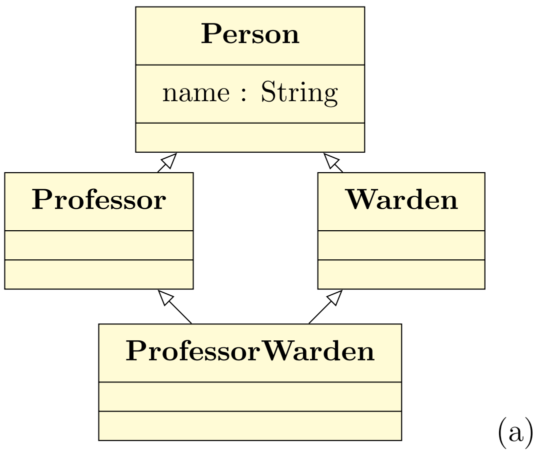
Unfortunately, this doesn't compile. May be a syntax error. Let's try:

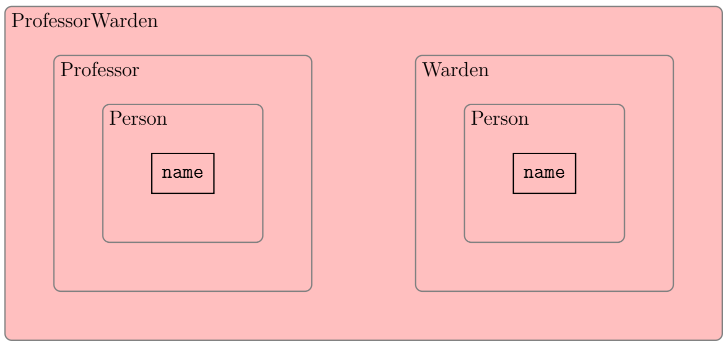
```
class ProfessorWarden extends Professor, extends Warden {
  ...
```

This too doesn't work. In fact, nothing seems to work when we try to implement the scenario in Java. Turns out that Java doesn't have any notion of multiple inheritance, at least not the way C++ allows. Wonder why?

To answer that question, let's consider the scenario that arises by considering the fact that both Professor and Warden have a common attribute `name`. And that's natural because both of them are basically *persons*. Doesn't that make it rather natural that they should both be derived from the same super-class `Person`? Let's do that, and make `name` an attribute of `Person`.

The classes are now related to each other as shown in fig. 26(a).





(b)

Figure 26: The diamond problem.

```
#include <iostream>
#include <string>

using namespace std;
class Person {
protected:
  const string name;
public:
  Person(string n) : name(n) {}
};

class Professor : public Person {
public:
  Professor(string n) : Person(n) {}

  void teach(string course, string cl) {
    cout << "Prof. " << Person::name << " teaches course "
      << course << " to class " << cl << "." << endl;
  }
};

class Warden : public Person {
public:
  Warden(string n) : Person(n) {}
  void approve(string request) {
    cout << "Warden " << Person::name <<
      " approves request " << request << "." << endl;
  }
};

class ProfessorWarden : public Professor, public Warden {
public:
  ProfessorWarden(string n) : Professor(n), Warden("ditto") {}
  void doWork() {
    cout << "Prof. " << Person::name <<
      " teaches and approves requests." << endl;
  }
};

...
```

Figure 27: `Warden` class.

The code corresponding to this scenario – again written in C++ (since Java doesn't seem to allow multiple inheritance anyway) – is shown in fig. 27. Note the changes:

1. `name` attribute's definition has been bumped up into `Person` class. There's not `name` attribute anymore in `Professor` or `Warden` classes.

2. The constructors of `Professor` and `Warden` now call the `Person`'s constructor for initiatialisation of `name` attribute.

3. At all places, the `name` attribute is referred to as `Person::name`.

We compile this code and ... Oops! We get the following compile error!

```
error: ‘Person’ is an ambiguous base of ‘ProfessorWarden’
```

Why does the code not compile? In fact, no amount of tweaking seems to fix this problem. To understand why? Let's take a look at the object structure of fig. 27(b). Remember that in every object of a class, there's embedded an object of its super-class. Therefore, each object of `ProfessorWarden` has an object of `Professor` and an object of `Warden`. Each of these objects (of `Professor` and of `Warden` classes) has in turn an object of `Person` class within it. This means, that there are really two objects of `Person` embedded in the belly of a `ProfessorWarden` object! So, when we are accessing the value of `Person::name` from within anywhere in `ProfessorWarden` class, which of these two `Person` objects are we talking about? There's no clear answer! This is the classical *diamond problem* of multiple inheritance (owing to the diamond-like structure that the inheritance relations create as shown in fig. 27(a)), and it has bothered programming language designers perhaps from the day multiple inheritance was first thought of. And no one can claim to have come up with a perfect solution to this problem.

Various programming languages have settled on their own favourite strategy. C++ allows multiple inheritance as long as the superclasses themselves haven't been derived from a common ancestor. Python uses a tie-breaking algorithm, called *method resolution order* (MRO), whenever an ambiguity arises. Java, which always leans towards defensive programming, has decided to simply disallow multiple inheritance, except in a very restricted manner.

9 Multiple Inheritance with Interfaces

Let's just see how the whole thing of implementing a `ProfessorWarden` class could be done in Java. First, the top level entities, `IProfessor` and `IWarden` interfaces, and `Person` class, as shown in fig. 28.

```
interface IProfessor {
  void teach(String course, String cl);
}

interface IWarden {
  void approve(String request);
}

class Person {
  protected String name;

  public Person(String n) {
    this.name = n;
  }
}
```

Figure 28: `IProfessor`, `IWarden` interfaces and `Person` class.

Things to note:

1. `IProfessor` and `IWarden` interface declare `teach` and `approve` methods respectively. The implementation of these methods isn't available as yet.

2. `Person` class is there just to give us a technically correct place to define the `name` attribute.

Next, we define the `Professor` and `Warden` classes which implement the `IProfessor` and `IWarden` interfaces respectively.

```
class Professor extends Person implements IProfessor {
  public Professor(String n) {
    super(n);
  }

  public void teach(String course, String cl) {
    System.out.println("Prof. " + this.name + " teaches "
      + course + " to " + cl + ".");
  }
}

class Warden extends Person implements IWarden {
  public Warden(String n) {
    super(n);
  }

  public void approve(String request) {
    System.out.println("Warden " + this.name
      + " approves request " + request + ".");
  }
}
```

Figure 29: `Professor`, `Warden` classes.

Here, we see that `Professor` class extends `Person` simply because it wants its `name` attribute. It implements the `IProfessor` interface because it needs to be known by its user (client code) as one of `IProfessor` that `teaches`. Similar is the case with `Warden`.

Now, when we want to define a `ProfessorWarden`, we want it to have a `name` (and so we extend it from `Person`), we want it to be available to `teach` (so it `implements IProfessor`), and we want it to be available to `approve` (so it `implements IWarden`). However, it has to do the implement `teach` and `approve` methods all over again, since it really has nothing to do with the `Person` and `Warden` classes (except having a common super-class `Person` and implementing the common interfaces) which have already implemented these methods.

Does this limit the amount of re-use that Java's version of multiple inheritance allows us? Well, of course it does! But let's look at the matter this way: there are really two ways in which inheritance facilitates code re-use,

1. By allowing us to define sub-classes that re-use the implemented parts

of their super-classes.

2. By allowing us to write client classes which use re-usable interfaces, which can be implemented as many times as we like as long as the interface is honoured.

If you were given the choice of picking any one of the above two types of re-uses, which one would you pick? The answer is: the second kind of re-use is far more important than the first kind. The success of any piece of code – like any other product – can measured by the amount of code that uses it. In other words, the client code for any piece of successful/important code is far more voluminous than itself. Hence, it is this part of the code which really needs to be preserved during software evolution. Interfaces allow us to define contract and protocols assuming which arbitrary amount of client code can be written. Implementing those interfaces will never let those pieces of code break (i.e. stop functioning). To align Java's stance on multiple inheritance with defensive programming, it has sacrificed an important aspect of re-usability. But it still gives us the more important benefit: that of re-usable interfaces and clients that continue working seamlessly with new implementations of those interfaces.

Morale of the story:

1. Client code are more important than the server code because they are more numerous and distributed among multiple users.

2. Interfaces are more important than implementations, because they are what the clients see and assume. As long as interfaces are preserve, the underlying implementations can be changed seamlessly.