Chapter 8

Object oriented programming

Consider the example of the class `Rectangle` discussed in the previous chapter. It is repeated below.

```
// class defintion
class Rectangle {
// fields
   private int height, width;
   private Point center;
   String desc = "Rectangle";

// constructors
   Rectangle() {    // default constructor
      height = 10;
      width = 10;
      center = new Point(0, 0);
   }

   Rectangle(int height, int width, int cx, int cy)
             throws IllegalValueException {    // constructor with arguments
                          // this is an example of overloading

      if (height < 0) throw new IllegalValueException();
      if (width < 0) throw new IllegalValueException();

      this.height = height;
      this.width = width;
      center = new Point(cx, cy);
   }

// methods
```

```
   public int getWidth() {
      return this.width;
   }

   public int findArea() {
      return height*width;
   }

   public String toString() {
      return "(" + height + ", " + width +")";
   }

   public void print() {
      System.out.println(this);
   }

   public void draw() {
       System.out.println("Drawing a rectangle");
   }

   public static void main(String[] args) { // only for testing
                                            // otherwise, no need for "main"
      Rectangle r;   // only a reference is created

      r = new Rectangle();  // construct
      int a = r.findArea();  // send a "message"
      r.print();  // send a "message", again!
      System.out.println(a + " " + r);  // procedural thinking

      try {      // an example of execeptions
        r = new Rectangle(-10, 10, 0, 0);
      } catch (Exception e) {
        System.out.println(e);
      }
   }
}
```

Observe how the class is used.

1. Define a class.
2. Create *instances* of the class. The declaration `Rectangle r` creates a reference for the instance. The object construction takes place by using the constructor.

3. Computation takes place by sending messages to objects.

In an object-oriented language, computation takes place through messages between objects.

8.1 Inheritance

Suppose the class `Rectangle` is so well-designed that it is used everywhere. Over a period of time, users realize that its functionality is limited. In particular, it does not have the attribute “color”. How to incorporate color? There are two ways of doing it. One way is to design an object which *has* rectangle and color attributes as shown below.

```
class ColorRect {
   public Rectangle rect;
   public Color color;
}
```

This is called *object composition* and is a well-known notion in many languages. For example, one can compose `struct`s in C.
Let us consider a different way of thinking: a colored rectangle is a rectangle - with additional attributes. Since a colored rectangle is a rectangle, most of the behavior should be the same. Can we preserve the “look-and-feel” of rectangle? We can, if we write extra code. Can we reuse the code from the base class? The following code illustrates this point.

```
// note the keyword "extends" which indicates that some fields and
// methods of "Rectangle" are available. (Not all!)
class ColorRectangle extends Rectangle {
   Color color;
   String desc = "Colored Rectangle"; // shadow the parent's field

// constructor - the constructor of rectangle is called before this
   ColorRectangle() {
       color = new Color();
   }

// methods - most of the methods of Rectangle are available

// we can override some methods
   public void draw() {
       System.out.println("Drawing a colored rectangle");
   }
}
```

The difference in behavior is illustrated by the following.

```
   public static void main(String[] args) {
      ColorRectangle cr = new ColorRectangle();
      ColorRect cr1 = new ColorRect();

      System.out.println(cr.getWidth()); // "getWidth" method for free
      System.out.println(cr1.getWidth()); // compiler complains
   }
```

In many realistic problems, it is often difficult to decide whether to use is a or has a relation between objects. This is the major source of design errors. In this case, do we consider a ColorRectangle as a Rectangle with Color attributes or a Color with Rectangle attributes? In nontrivial situations, this decision is not easy.
Inheritance forms the underlying skeleton structure supporting objects. Rectangle, for example, is a convex polygon. We can group triangles, rectangles, etc. into a single category called convex polygon. All the code and properties relating to convex polygons can be related to this single (super)-class rather than to each of of the (infinity) of polygons. Other specialized compulations like rendering and area calculation can be stored with the (sub)-classes. This organizational principle is called *inheritance*.

8.2 Polymorphism

The final feature of OOP is *polymorphism*. Suppose we want to render the object on the screen. Instead of finding the object type and calling an appropriate rendering routine, we can call a single routine `draw` with the object as argument. The compiler or the run-time calls the appropriate routine since it keeps track of the type of the object. In this context, it is helpful to think of computation taking place through message passing. The message "render" is passed to the object. If the object can do the task, it replies to the message. If not, it passes it to the superclass. For example, we can have the message "isConvex?". Because of inhertiance, this is not stored with the subclasses. The class rectangle (for example), passes it to the super class and the super class replies to the message.
Note that a derived class can *override* some of the methods of the base class. This enables the implementation of specialized behavior. Consider the following example.

```
   public static void main(String[] args) {
      ColorRectangle cr = new ColorRectangle();
      Rectangle r = new Rectangle();

// method behavior
      cr.draw();
      r.draw();
      r = cr; // upcasting
      r.draw(); // what is drawn here?
```

```
// field behavior
      System.out.println(r.desc);
      System.out.println(cr.desc);
   }
```

What is the need for upcasting? In a general programming scenario of drawing shapes. This has a rendering engine. How should this rendering engine be designed? What should be its behavior? The engine should take a shape and `draw()` it – or send the object the `draw()` message. Suppose the engine was designed years ago. How can it handle a shape, say `Fractal`, developed now? Should the engine's code be changed?
If the engine is designed to take a `Shape` object as input, we can make `Fractal` a subclass of `Shape` and define a `draw()` method for `Fractal`. If we pass a `Fractal` object to the engine, the correct `draw()` method is called at *runtime*. This is the concept of *polymorphism*. Polymorphism helps code maintainance.
Since polymorphism requires runtime support, we can ask the question: how much can we extend the runtime support? In Java, complete information about classes is available at runtime. This is illustrated by the following.

```
import java.util.*;

class Flower { }

class Frog { }

public class FandF {
  public static void main(String[] args) {
    Object[] arr = new Object[4]; // "Object" is the root of class hierarchy
// because of upcasting, class information is lost in the following
    for(int i = 0; i < 3; i++) arr[i] = new Flower();
    arr[3] = new Frog();
    for(int i = 0; i < arr.length; i++) {
      Class c = arr[i].getClass(); // but correct information is available
      System.out.println(c);
    }
  }
}
```

8.3 Programming with classes

The central ideas of OOP are

1. data abstraction

2. message passing
3. inheritance
4. polymorphism

To turn these ideas into a concrete programming language, more work is needed: syntax, semantics, integration with other ideas, etc. The following discusses some of these for Java. A class is the unit of an object oriented programming language. Classes are generalizations of types. **Computation is the interaction between instances**. Ideally, all the data fields of a class should be invisible to other classes. But for efficiency considerations, the data fields and methods are associated with access restrictions. Java defines four levels of visibility. By default, the fields and methods are freely accessbible over the current package (This will be discussed later). This can be changed by explicitly declaring the access qualifiers as: `private`, `protected`, or `public`.
Data and methods can be qualified by `static` which means that they are specific to the class and not instances. An example of a class field is a counter which keeps track of the number of instances created. Class methods work on class fields.
When a class/variable/method is declared `final`, it means that the class cannot be subclassed/the variable's value cannot be changed/the method cannot be overridden.
When a variable is declared as belonging to a class, only an null-initialzed *reference* is created but no storage is allocated. Storage allocation takes place only when a *constructor* to the class is explicitly called.
An instance can refer to itself using `this`. It can use its parent's methods or shadowed fields using `super`.
An important feature of Java is that *a superclass reference variable can refer to a subclass object*. This necessitates *dynamic method dispatch* or *runtime type identification* or *dynamic binding*.
And finally, a class can leave some of its methods unimplemented. Such classes and methods should be declared `abstract`. It is the job of the subclasses to implement the methods. It is not possible to create instances of abstract classes but we can create reference variables of abstract classes.
A class can have a method `public string toString()`. This is used when an object of the class needs to be printed. More generally, it gives a string description of the object.

8.4 Multiple inheritance

Can a class extend two classes simultaneously? Such an behavior is called multiple inheritance. Multiple inheritance can lead to some conceptual difficulties. Suppose class Z extends classes X and Y. If there are fields with the same name in X and Y, the name conflict has to be resolved. The compiler can detect such conflicts.

C++ supports multiple inheritance. Java has a mechanism called `interface` which is meant to support multiple inheritance. But interfaces also suffer from the name conflicts. A real solution to multiple inheritance is through some naming convention.

8.5 Interfaces

`abstract` classes give implementations of some functions. `interfaces` take this idea to the extreme. They do not provide implementations of any method. This is fully consistent with the object-oriented principle of data hiding. Classes which `implement` interfaces typically have all their data `private`. We can have multiple classes implementing the same interface. Also a single class implementing multiple interfaces. In fact, this is the only way to do multiple inheritance in Java. Note that the data fields of interfaces are treated implicitly as `final`. One interface can extend another interface by adding more (unimplemented) methods.

8.6 Names

The notion of packages has multiple uses: bundling, naming, and access control. Let us consider the first: bundling classes. As we said the unit of a Java program is a class. There should be a way of keeping operationally related classes together. (Inheritance is a way of tying semantically related classes.) We can declare a class as belonging to a particular package by using the keyword `package`.
Each package has a different name space. Thus a class `silly` in the package `mistakes` is different from the class `silly` in the package `jokes`. (The two class are actually `mistakes.silly` and `jokes.silly`.) Package names are actually qualified by the reverse domain names. Hence `jokes.silly`s of `foo.com` and `foo.edu` are actually `com.foo.jokes.silly` and `edu.foo.jokes.silly`. (This is neither silly mistake or a joke but a good idea.)
Java extracts one more use from the notion of packages – that of access control. The default access scope of a particular variable is the package in which it appears. We have the following table:

	`private`	no modifier	`protected`	`public`
same class	yes	yes	yes	yes
same package, subclass	no	yes	yes	yes
same package, non-subclass	no	yes	no	yes
different package, subclass	no	no	yes	yes
different package, non-subclass	no	no	no	yes

As mentioned before, the notion of packages is to increase the number of name spaces. While entering code, it is difficult to use fully qualified class names. The `import` feature of Java simplifies name references. One way to refer to the method `narrate()` of `jokes.silly` is `jokes.silly.narrate()`. A simpler way is to `import` the appropriate package and "call by the first name".

```
import jokes.*;

class funny {
   void crack() {

      narrate();

   }
}
```

8.7 History

OOP started as a generic modeling idea in early 1960s and later became very popular because of its usefulness in multiprogrammer development situations. An object has an *internal state* and an *external behavior*. States are implemented through data structures and the behavior is implemented through functions or procedures.
Consider the following situation: You want to develop a treasure hunt program. This can be used either as a game or as a simulation program. The entities in the problem are: hunters, treasure objects, general environment, etc. Note that the hunters have some specific information they have gathered which should be internal to them. For example, this information could be derived through decoding some clues. This should not be visible to other objects. The internal state determines their external behavior which is visible to other objects.
Another scenario is three progammers working on the same project. The author programmer developes a class, the client programmer extends the class, and the user programmer uses the class. How do the programmers divide and share the work? *How much each programmer should know about the other programmer's work?* In the OOP terminology, the user programmer or the user knows only about the behavior. In this case, the external behavior is called the *contract* between the desiger or user. The developer is free to change the details (or internal state representation) as long as the contract is met. The contract specifies only *what* is done not *how*. The relation between the author and the client is more subtle. Does the client programmer extend the behavior or state? This calls for careful design decisions.