**JAVA and the Object Oriented Paradigm**

Chapters 6 and 7 in the book

**Features**

- Java programs run on a wide variety of hardware platforms.
  - Java programs are portable.
    - The Java code is translated into byte code by the Java compiler. The byte code is executed by the Java Virtual Machine.
    - JVM performs run-time checks that complement the compile-time checks performed by all compilers.
- Java provides features that facilitate secure and robust behavior.
  - Example: No Java program running through a browser can open, read or write files on your computer.
- Java is completely object-oriented.
- Java programs can work on multiple tasks simultaneously.
  - multi-threading
- Java programs automatically recycle memory.
  - Automatic garbage collection.

**Classes, Instances, Instance Variables**

- A class describes a set of objects.
- Each object in a class is called an instance of that class.
  - To declare and instantiate a new instance:

```java
class name variable name = new class name();
```
- Each object is associated with a set of instance variables (fields/attributes of the class).
- Instance variables can be defined private, public or protected.
  - Why? protection from misuse, access but hiding implementation details.
    - public: universal access.
    - private: access from the class in which they are introduced (getter and setter methods are introduced).
    - protected: access from to the package in which the are introduced and to the subclasses of the class in which they are introduced.
    - nothing: package in which they are introduced (default).
- We define a class for Circle. Each circle which is an object in this class has the following attributes: center_x, center_y, and radius. We will assume that each instance variable of Circle is of type int.

```java
class Circle {
    public double center_x, center_y;
    public double radius;
}
```

Let c be an instance of Circle created using

```java
Circle c = new Circle();
```

Then the center coordinates and radius of c can be accessed as: c.center_x, c.center_y, and c.radius respectively.
- **Method-abstraction**: act of moving computational details into a method.
  - Exploiting the *divide-and-conquer* problem-solving heuristic.
  - Debugging, Reading, Augmenting, Improving and Adapting are easier.

- A method can be define **public**, **private** and **protected**.

- Java does not require to define a method before using it in the code (it has to be defined eventually however).

- We talk about an **Overloaded Method Name** whenever there is more than one definition for a method and we distinguish the methods by examining the number or types of their parameters. The methods have different semantics.
  Example: `+` is an overloaded operator because it has several different meanings (for ints and strings).

---

**Constructors**

- Constructors perform computations such as initial instance variable assignment that you want to occur when your program creates an instance.

- Constructors do not return values.

- Java defines a **default constructor** with 0 parameters. It initializes the instance variables with their respective default values.

- Example:

  ```java
  public class C{
    public int i, j;
    public C(){
      i = 5;
      j = 7;
    }
    public C(int ni, int nj){
      i = ni;
      j = nj;
    }
  }
  
  What mean C c;? C c = new C();?
  ```

---

**Data abstraction**

- **Data abstraction**: Act of moving implementation details into a set of access methods (constructors, getter and setter methods).

- Data are hidden.

- Data abstraction has many virtues:
  - Your programs become easier to read and reuse.
  - You can easily augment what a class provides.
  - You can easily improve the way that data are stored.
  - You can anticipate the changes.

- We talk about **Overriding** or **Shadowing** when methods with the same name, same number of parameters and the same parameter types are defined in different classes linked with the subclass-superclass relation (direct or not).
Memory

- Memory at run time divides in three parts:
  - static area (for values whose storage requirements are known before run-time and remains constant)
  - run-time stack (for active methods, their local variables, and their parameter-argument linkages)
  - heap (for values that are dynamically allocated during the run-time life of a program)

- Java hides pointers. Objects are initialized in the heap.

- Garbage collector.

Public class Stack

    private class Node{
        int val;
        Node next;
        Node(int v, Node n){
            val = v;
            next = n
        }
        private Node stack = null;
        public int pop(){
            int result = stack.val;
            stack = stack.next;
            return result;
        }
        public void push(int v){
            stack = new Node(v,stack);
        }
    }

- Draw the heap for the following execution:

  Stack s = new Stack();
  s.push(1);
  s.push(2);
  s.pop();
  s.push(5);

- What is a clone? a deep clone of an object of type Stack

Parameter passing

- Java is said to be a **called-by value** language.
  - If a parameter is of a primitive type (byte, short, int, long, char, float, double or boolean) the parameter is initialized with the value of the argument. Subsequent changes to the parameter do not affect the argument.
  - If a parameter is a reference, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to the parameter do affect the argument.

- Example: (Location.java handout)

  Location p = new Location(1.0,2.0);
  Location s = new Location(1.5,2.1);

  public static double distance (Location p1, Location p2){
      ...
  }

  Draw the memory.

Exceptions in Java

- An exception is a condition detected by an operation that cannot be resolved in the context of the operation.
  Example: Division by 0.

- To throw an exception is to signal that an exception has occurred.

- To catch an exception means that the exceptions thrown causes control to be transferred to an exception handler.

- In Java an exception is a subclass of Throwable.

- Exceptions are divided in three classes:
  - Errors in the JVM (Error and its subclasses)
  - Errors at runtime (RunTimeException and its subclasses)
  - All others
### Object-oriented Paradigm

- **Encapsulation (based on ADTs - a type is a set of data with operations), Inheritance, Polymorphism**
- An object can be viewed as a machine having both data and operations on the data.
- A way to think of a program is as a collection of interacting objects that communicate via **message passing**.
- When an object passes a message to another object, it invokes a particular method. The parameters are the message contents.

### Abstract classes and methods

- A class representing an abstract concept and, as such, should not be instantiated. To prevent the instance creation the class is marked as **abstract**.

Examples: Food is an abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist but it makes sense for an instance of carrot, apple or chocolate to exist.

- An **abstract method** is a method without implementation.

- Any class that has an abstract method or that does not provide an implementation for any abstract method must be declared as an abstract class.

```
// Syntax:
try{
    // Statement with potential to throw exception
}
catch (exception class name parameter 1){
    // Exception handling code 1
}
catch (exception class name parameter 2){
    // Exception handling code 2
}
... finally{
    // Clean up statements
}

Note: Finally is **always** executed.

- Good programming practice!

```

```
abstract class GraphicObject {
    int x, y;
    ... void moveTo(int newX, int newY) {
        ... }
    abstract void draw();
}

class Circle extends GraphicObject {
    void draw() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
}
```

- **Example:**
**Interfaces**

- Interfaces describe a protocol of behavior that can be **implemented** by any class anywhere in the class hierarchy (There are no obligation of inheritance relation between the interface and the class that implements the interface).
- Use of the keyword `implements` in JAVA.
- The methods of an interface are all abstract. They will be implemented in the class that implements the interface.
- A class can implement different interfaces.
- An interface cannot grow.
- An interface can impose requirements without forcing those requirements into the ordinary superclass-subclass chain.
- An interface can serve as a type indicator.

---

**Inheritance**

- In Java the class hierarchy forms a (strict) tree rooted in the most general class **Object**.
- The Class hierarchy reflects the subclass-superclass relation among classes.
- Interests to arrange classes in a hierarchy:
  - To parallel natural categories.
  - To prevent avoidable duplication and to simplify maintenance.
  - To avoid introducing bugs into previously debugged code.
  - To use purchased code.
- Java does not allow multiple-inheritance. (C++ and EIFFEL do - Problems of multiple inheritance).

---

**Interface versus Abstract class**

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

---

**Let's describe the set of colored circles: circles whose interior is filled with some color. Note that each colored circle is also a circle. Thus, we can define a class of colored circles `ColoredCircle` that is a subclass of `Circle`. In Java syntax, this can be written as:**

```java
class ColoredCircle extends Circle{
    int color;
}
```

The subclassing means that each instance of `ColoredCircle` is also an instance of `Circle`. In addition, `ColoredCircle` have an extra attribute, i.e. their color. Thus, if `d` is an instance of `ColoredCircle`, then `d.color` is valid, as well as `d.center.x, d.center.y, and d.radius`. Also, we can apply the methods applicable to any instance of `Circle` to `d`. 
** Polymorphism

- Polymorphism = having many forms.

- In object-oriented languages it refers to the late binding of a call to a specific instance variable or method in an object.

- Let \( d \) be an instance of \( ColoredCircle \), and \( c \) be a variable of type \( Circle \).

  What do we get if we do the following:
  
  \[
  c = d; \\
  c\text{.draw}(); \\
  \]

  Not obvious!!

- The technique used to find out which method (or instance variable) to use for an object is called method (or field/instance variable) resolution.

- What is implemented in Java?

- Note: The term polymorphism has a different meaning in functional programming.

** Field and Method Access

- Let \( c \) be an instance of class \( Circle \) and \( d \) be an instance of class \( ColoredCircle \).
  
  - \( c\text{.radius} \) accesses the \( radius \) field defined in \( Circle \)
  
  - \( d\text{.radius} \) also accesses the \( radius \) field defined in \( Circle \) (by inheritance)
  
  - \( d\text{.color} \) accesses the \( color \) field defined in \( ColoredCircle \).

- Field access is done at compile time in Java.

  - Algorithm:

    Say we want to access the field \( f \) of an object of class \( X \).

    We first check if \( f \) is defined in \( X \) itself; if not, we check if \( f \) is defined in the superclass of \( X \) (say, \( X' \)); if not, we check if \( f \) is defined in the superclass of \( X' \); and so on, until we reach the root of the class hierarchy. If \( f \) is defined nowhere in this path, we know there is an error. Otherwise, we take the field \( f \) that is defined closest to the class \( X \) on the path from \( X \) to the root of the class hierarchy.

- Java resolves method invocations at run time.

  - Similar algorithm but at run time.

** Listing

```java
** class Circle {
    public double center_x, center_y;
    public double radius;
    public int window_number;

    public double circumference() {
        return 2.0 * 3.141529 * radius;
    }

    public double area() {
        return 3.141529 * radius * radius;
    }

    public boolean overlaps(Circle other) {
        return ((center_x - other.center_x) * (center_x - other.center_x) + (center_y - other.center_y) * (center_y - other.center_y)) < ((radius + other.radius) * (radius + other.radius));
    }
}

** class ColoredCircle extends Circle{
    public int color;

    public void draw() { // (4)
        // method to draw the colored circle
    }
}

/**
 * method to draw circle on the screen...
 */

public void draw(Color color) { // (2)
    // method to draw circle on the screen with a
    // given color
}

public void draw(TextureMap tm) { // (3)
    // method to draw the circle and fill it
    // with given texture map
}
```
Example

- Circle c1;
  ColoredCircle d1;
  c1 = new Circle();
  d1 = new ColoredCircle();
  c1.draw(); -- What draw is it?
  c1 = d1;
  d1.draw(); -- What draw is it?
  c1.draw(); -- What draw is it?

- When we try to find out which method we mean by the last c1.draw(), we realize that although the variable c1 may be of declared to be of type Circle, it actually refers (at run time) to an object in class ColoredCircle. Hence, we search for an appropriate method starting from ColoredCircle.

Fields versus methods resolution

- Why are fields and methods treated differently in Java?

- Advantage of checking the resolving method invocations at run time.
  Consider an array A that can hold Circle objects. Since each ColoredCircle is also a Circle, some elements of A may actually be ColoredCircles. Now if we want to draw all the objects in A, we can simply do A[i].draw() for each element i in the array.
  If we resolve method invocations dynamically, we will invoke the method most appropriate for the objects that are actually stored in the array. In contrast, if we resolve method invocations at compile time, we will always invoke the draw() method of Circle, rendering all circles without color!

Reflexivity in Java

- Why are field accesses resolved at compile time?

  - Field accesses are more common, and it becomes expensive to search for the appropriate field every time we want to access some data stored in an object.
  - We can always write access methods for each field (a get- and a set- method for each field: to get its value and to store a new value respectively) and use these methods whenever we want to simulate run-time field resolution.

- How to get information about a Java class?

  - For each class, the Java Runtime Environment (JRE) maintains a Class object that contains information about a class.
    - Retrieve the class object of a class.
    - Finding the superclasses of a class
    - Identifying the class fields of a class.
    - Discovering class constructors
    - Obtaining methods information...

  - http://java.sun.com/docs/books/jdk1.6/docs/api/java/reflect/Class.html
Summary

- Features of the language
- Memory management
- Exceptions
- Abstract class
- Interface
- Inheritance
- Field/Method resolution
- Reflexivity
- http://java.sun.com/docs/books/tutorial/