Core Java

Day 08 Agenda

- Revision: Abstract class/method
- Interfaces
- Marker interfaces

abstract keyword

- In Java, abstract keyword is used for
  - abstract method
  - abstract class

abstract method

- If implementation of a method in super-class is not possible/incomplete, then method is declared as abstract.
- Abstract method does not have definition/implementation.

```
// Employee class
abstract double calcTotalSalary();
```

- If class contains one or more abstract methods, then class must be declared as abstract. Otherwise compiler raise an error.
- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- The abstract method cannot be private, final, or static.
- Example: abstract methods declared in java.lang.Number class are:
  - abstract int intValue();
  - abstract float floatValue();
  - abstract double doubleValue();

- abstract long longValue();
- These methods are useful for type conversion.

```
Double d = new Double(3.142);
double x = d.doubleValue(); // 3.142
float y = d.floatValue(); // 3.142f
int z = d.intValue(); // 3

Integer i = new Integer(123);
int p = i.intValue(); // 123
double q = i.doubleValue(); // 123.0
```

abstract class

- If implementation of a class is logically incomplete, then the class should be declared abstract.
- If class contains one or more abstract methods, then class must be declared as abstract.
- An abstract class can have zero or more abstract methods.
- Abstract class object cannot be created; however its reference can be created.
- Abstract class can have fields, methods, and constructor.
- Its constructor is called when sub-class object is created and initializes its (abstract class) fields.
- Example:
  - java.lang.Number
  - java.lang.Enum
- Example:

```
Number m = new Number(123); // compiler error
Number n = new Integer(123); // okay
```

Fragile base class problem

- If changes are done in super-class methods (signatures), then it is necessary to modify and recompile all its sub-classes. This is called as "Fragile base class problem".
- This can be overcomed by using interfaces.

Interface (Java 7 or Earlier)

- Interfaces are used to define standards/specifications. A standard/specification is set of rules.
- Interfaces are immutable i.e. once published interface should not be modified.
- Interfaces contains only method declarations. All methods in an interface are by default abstract and public.
- They define a "contract" that is must be followed/implemented by each sub-class.

```
interface Displayable {
    public abstract void display();
}
```

```
interface Acceptable {
    abstract void accept(Scanner sc);
}
```

```
interface Shape {
    double calcArea();
    double calcPeri();
}
```

- Interfaces enables loose coupling between the classes i.e. a class need not to be tied up with another class implementation.
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.
- Java 7 interface can only contain public abstract methods and static final fields (constants). They cannot have non-static fields, non-static methods, and constructors.

- Examples:
  - java.io.Closeable / java.io.AutoCloseable
  - java.lang.Runnable
  - java.util.Collection, java.util.List, java.util.Set, ...
- Example 1: Multiple interface inheritance is allowed.

```
interface Displayable {
    void display();
}
interface Acceptable {
    void accept();
}

class Person implements Acceptable, Displayable {
    // ...
    public void accept() {
        // ...
    }
    public void display() {
        // ...
    }
}
```

- Example 2: Interfaces can have public static final fields.

```
interface Shape {
    /*public static final*/ double PI = 3.142;

    /*public abstract*/ double calcArea();
    /*public abstract*/ double calcPeri();
}
```

```
class Circle implements Shape {
    private double radius;
    // ...
    public double calcArea() {
        return PI * this.radius * this.radius;
    }
    public double calcPeri() {
        return 2 * Shape.PI * this.radius;
    }
}
```

- Example 3: If two interfaces have same method, then it is implemented only once in sub-class.

```
interface Displayable {
    void print();
}
interface Showable {
    void print();
}
class MyClass implements Displayable, Showable {
    // ...
    public void print() {
        // ...
    }
}
class Program {
    public static void main(String[] args) {
        Displayable d = new MyClass();
        d.print();
        Showable s = new MyClass();
        s.print();
        MyClass m = new MyClass();
        m.print();
```

```
        }
    }
```

- Interface syntax
    - Interface : I1, I2, I3
    - Class : C1, C2, C3
    - class C1 implements I1 // okay
    - class C1 implements I1, I2 // okay
    - interface I2 implements I1 // error
    - interface I2 extends I1 // okay
    - interface I3 extends I1, I2 // okay
    - class C2 implements C1 // error
    - class C2 extends C1 // okay
    - class C3 extends C1, C2 // error
    - interface I1 extends C1 // error
    - interface I1 implements C1 // error
    - class C2 implements I1, I2 extends C1 // error
    - class C2 extends C1 implements I1,I2 // okay

class vs abstract class vs interface

- class
    - Has fields, constructors, and methods
    - Can be used standalone -- create objects and invoke methods
    - Reused in sub-classes -- inheritance
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism
- abstract class
    - Has fields, constructors, and methods
    - Cannot be used independently -- can't create object
    - Reused in sub-classes -- inheritance -- Inherited into sub-class and "must override abstract methods"
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism

- interface
  - Has only method declarations
  - Cannot be used independently -- can't create object
  - Doesn't contain anything for reusing (except static final fields)
  - Used as contract/specification -- Inherited into sub-class and "must override all methods"
  - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism

Marker interfaces

- Interface that doesn't contain any method declaration is called as "Marker interface".
- These interfaces are used to mark or tag certain functionalities/features in implemented class. In other words, they associate some information (metadata) with the class.
- Marker interfaces are used to check if a feature is enabled/allowed for the class.
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
  - java.io.Serializable -- Allows JVM to convert object state into sequence of bytes.
  - java.lang.Cloneable -- Allows JVM to create copy of the class object.

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.

```
class Date implements Cloneable {
    private int day, month, year;
    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }
}
```

```
class Person implements Cloneable {
    private String name;
    private int weight;
    private Date birth;
    // ...
    // deep copy
    public Object clone() throws CloneNotSupportedException {
        Person temp = (Person)super.clone(); // shallow copy
        temp.birth = (Date)this.birth.clone(); // + copy reference types explicitly
        return temp;
    }
}
```

```
class Program {
    public static void main(String[] args) throws CloneNotSupportedException {
        Date d1 = new Date(28, 9, 1983);
        System.out.println("d1 = " + d1.toString());
        Date d2 = (Date)d1.clone();
        System.out.println("d2 = " + d2.toString());
        Person p1 = new Person("Nilesh", 70, d1);
        System.out.println("p1 = " + p1.toString());
        Person p2 = (Person)p1.clone();
        System.out.println("p2 = " + p2.toString());
    }
}
```

Assignment

1. Create an abstract class BoundedShape with fields x, y. Provide abstract method calcArea(). Inherit it into a Circle class with additional fields radius and override calcArea() method. Inherit BoundedShape into another abstract class Polygon with additional field number of sides. Inherit BoundedShape into classes Triangle (fields: side1, side2, side3), Square (fields: side), and Rectangle (fields: length, breadth). Override calcArea() method in them.
2. Create an abstract Player class with id, name, age, and matchesPlayed as fields. Create a Batter interface with methods like getRuns(), getAverage(), and getStrikeRate(). Creater a Bowler interface with methods like getWickets(), and getEconomy(). Create a class Cricketer (with fields like runs, wickets, etc.) inherited from Player as well as Batter and Bowler interfaces. In all classes write appropriate constructors, getter/setters, accept(), toString(), and equals() methods. In main(), create a team (array) of 11 players and input their details from end user. Create a new (utilility) class Players that contains static methods to count number of batters, number of bowlers, total batter runs, total bowler wickets, return a batter with maximum runs, and return a bowler with maximum wickets.

```
class Players {
    public static int batterTotalRuns(Player[] arr) {
        int runs = 0;
        for(Player p:arr) {
            if(p instanceof Batter) {
                Batter b = (Batter)p;
                runs = runs + b.getRuns();
            }
        }
        return runs;
    }
    public static int bowlerTotalWickets(Player[] arr) {
        // ...
    }
    public static int countBatters(Player[] arr) {
        // ...
    }
    public static int countBowlers(Player[] arr) {
        // ...
    }
    public static Player maxRunBatter(Player[] arr) {
        // ...
    }
    public static Player maxWicketBowler(Player[] arr) {
```

```
        // ...
    }
}
```

```
class Program {
    public static void main(String[] args) {
        // ...
        Player[] team = new Player[11];
        for(int i=0; i<team.length; i++) {
            arr[i] = new Cricketer();
            arr[i].accept();
        }

        for(Player p:team)
            System.out.println(p.toString());

        int totalRuns = Players.batterTotalRuns(team);
        System.out.println("Total runs of all batters: " + totalRuns);
        // ...
    }
}
```

3. Implement Date and Person class with their clone() methods as shown in notes. Ensure that when p1's birth date is modified, it doesn't affect p2's birth date.