Core Java

Day 08 Agenda

- equals() implementation
- Abstract class/method
- Interfaces
- Marker interfaces

Inheritance vs Association

- Inheritance: is-a relation
  - Book is-a Product
  - Album is-a Product
  - Labor is-a Employee
  - Employee is-a Person
  - Batter is-a Player
  - ...
- Association: has-a relation
  - Employee has-a joining Date
  - Person has-a birth Date
  - Cart has Products
  - Bank has Accounts
  - ...

Object class

equals() method

- Non-final method of java.lang.Object class.

    - public boolean equals(Object other);
- Definition of Object.equals():

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- To compare the object contents/state, programmer should override equals() method.
- This equals() must have following properties:
    - Reflexive: for any non-null reference value x, x.equals(x) should return true.
    - Symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
    - Transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
    - Consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
    - For any non-null reference value x, x.equals(null) should return false.
- Example:

```
class Employee {
    // ...
    @Override
    public boolean equals(Object obj) {
        if(obj == null)
            return false;

        if(this == obj)
            return true;

        if(! (obj instanceof Employee))
            return false;

        Employee other = (Employee) obj;
```

```
        if(this.id == other.id)
            return true;
        return false;
    }
}
```

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 Number class are:
  - abstract int intValue();
  - abstract float floatValue();
  - abstract double doubleValue();
  - abstract long longValue();

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.
- If object of a class is not logical (corresponds to real-world entity), then class can be declared as abstract.
- Example:
  - java.lang.Number
  - java.lang.Enum

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
    - Java support multiple interface inheritance

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. In Day 07 assignment 2, make Player class as abstract. It ensures that Player object cannot be created (but references can be created).
2. In Day 07 assignment 3, make Product class as abstract and its calcPrice() method as abstract method.
3. 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.
4. Create an abstract Player class with id, name, age, and matchesPlayed as fields. Create a Batter interface with methods like getRuns(), getBallsPlayed(), getAverage(), and getStrikeRate(). Creater a Bowler interface with methods like getWickets(), getBallsBowled(), and getEconomy(). Create a class Cricketer 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 cricketers and input their details from end user. Create a new (utilility) class Players that contains static methods to count number of batters (if ballsPlayed > 0), number of bowlers (if ballsBowled > 0), total batter runs, total bowler wickets, return a batter with maximum runs, and return a bowler with maximum wickets.
5. Implement appropriate equals() method in all classes implemented in today's assignments.