Core Java

Day 07 Agenda

- Inheritance types
- Upcasting & Downcasting
- instanceof operator
- Method overriding
- Runtime Polymorphism
- Final class/method
- Object class
    - toString()

Inheritance

Types of inheritances

- Single inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
```

- Multiple inheritance

```
class A {
    // ...
}
class B {
    // ...
}
class C extends A, B // not allowed in Java
{
    // ...
}
```

```
interface A {
    // ...
}
interface B {
    // ...
}
```

```
class C implements A, B // allowed in Java
{
    // ...
}
```

- Hierarchial inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
class C extends A {
    // ...
}
```

- Multi-level inheritance

```
class A {
    // ...
}
class B extends A {
    // ...
}
class C extends B {
    // ...
}
```

- Hybrid inheritance: Any combination of above types

Up-casting & Down-casting

- Up-casting: Assigning sub-class reference to a super-class reference.
    - Sub-class "is a" Super-class, so no explicit casting is required.
    - Using such super-class reference, only super-class methods inherited into sub-class can be called. This is "Object slicing".
    - Using such super-class reference, super-class methods overridden into sub-class can also be called.

```
Employee e = new Employee();
Person p = e; // up-casting
p.setName("Nilesh");    // okay - calls Person.setName().
p.setSalary(30000.0);   // error
p.display();    // calls overridden Employee.display().
```

- Down-casting: Assigning super-class reference to sub-class reference.
    - Every super-class is not necessarily a sub-class, so explicit casting is required.

```
Person p1 = new Employee();
Employee e1 = (Employee)p1; // down-casting - okay - Employee reference will
point to Employee object
```

```
Person p2 = new Person();
Employee e2 = (Employee)p2; // down-casting - ClassCastException - Employee
reference will point to Person object
```

instanceof operator

- Java's instanceof operator checks if given reference points to the object of given type (or its sub-class) or not. Its result is boolean.

```
Person p = new Employee();
boolean flag = p instanceof Employee; // true
```

```
Person p = new Person();
boolean flag = p instanceof Employee; // false
```

```
Person p = new Manager();
boolean flag = p instanceof Employee; // true
```

- Typically "instanceof" operator is used for type-checking before down-casting.

```
Person p = new SomeClass();
if(p instanceof Employee) {
    Employee e = (Employee)p;
    System.out.println("Salary: " + e.getSalary());
}
```

Polymorphism

- Poly=Many, Morphism=Forms
- Polymorphism is taking many forms.
- OOP has two types of Polymorphism

- Compile-time Polymorphism / Static Polymorphism
  - Implemented by "Method overloading".
  - Compiler can identify which method to be called at compile time depending on types of arguments. This is also referred as "Early binding".
- Run-time Polymorphism / Dynamic Polymorphism
  - Implemented by "Method overriding".
  - The method to be called is decided at runtime depending on type of object. This is also referred as "Late binding" or "Dyanmic method dispatch". ###Access Modifier
- private (lowest)
- default
- protected
- public (highest)

Method overriding

- Redefining a super-class method in sub-class with exactly same signature is called as "Method overriding".
- Programmer should override a method in sub-class in one of the following scenarios
  - Super-class has not provided method implementation at all (abstract method).
  - Super-class has provided partial method implementation and sub-class needs additional code. Here sub-class implementation may call super-class method (using super keyword).
  - Sub-class needs different implementation than that of super-class method implementation.
- Rules of method overriding in Java
  - Each method in Java can be overridden unless it is private, static or final.
  - Sub-class method must have same or wider access modifier than super-class method.

```
class SuperClass {
    protected Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    /*protected or*/ public Number calculate(Integer i, Float f) {
        // ...
    }
}
```

  - Arguments of sub-class method must be same as of super-class method. The return-type of sub-class method can be same or sub-class of the super-class's method's return-type. This is called as "covarient" return-type.

```
class SuperClass {
    public Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
```

```
    // Double is inherited from Numer (i.e. return-type of super-class
method)
    public Double calculate(Integer i, Float f) {
        // ...
    }
}
```

- Checked exception list in sub-class method should be same or subset of exception list in super-class method.

```
class SuperClass {
    public void testMethod() throws IOException, SQLException {
        // ...
    }
}
class SubClass extends SuperClass {
    public void testMethod() throws IOException {
        // ...
    }
}
```

- If these rules are not followed, compiler raises error or compiler treats sub-class method as a new method.

```
class A {
    void method1() {
    }
    void method2() {
    }
}
```

```
class B extends A {
    void method1() {
        // overridden
    }
    void method2(int x) {
        // treated as new method
    }
}
```

```
// In main()
A obj = new B();
obj.method1();  // B.method1() -- overridden
obj.method2();  // A.method2() -- method2() is not overridden
```

- Java 5.0 added @Override annotation (on sub-class method) informs compiler that programmer is intending to override the method from the super-class.
- @Override checks if sub-class method is compatible with corresponding super-class method or not (as per rules). If not compatible, it raise compile time error.
- Note that, @Override is not compulsory to override the method. But it is good practice as it improves readability and reduces human errors.

final keyword

final variables/fields

- Cannot be modified once initialized.
- Refer earlier notes.

final method

- If implementation of a super-class method is logically complete, then the method should be declared as final.
- Such final methods cannot be overridden in sub-class. Compiler raise error, if overridden.
- But final methods are inherited into sub-class i.e. The super-class final methods can be invoked in sub-class object (if accessible).

final class

- If implementation of a super-class is logically complete, then the class should be declared as final.
- The final class cannot be extended into a sub-class. Compiler raise error, if inherited.
- Effectively all methods in final class are final methods.
- Examples of final classes
    - java.lang.Integer (and all wrapper classes)
    - java.lang.String
    - java.lang.System

Object class

- Non-final and non-abstract class declared in java.lang package.
- In java, all the classes (not interfaces) are directly or indirectly extended from Object class.
- In other words, Object class is ultimate base class/super class.
- Object class is not inherited from any class or implement any interface.
- It has a default constructor.
    - Object o = new Object();

Object class methods (read docs)

- Parameter less constructor
    - public Object();
- Returns string representation of object state
    - public String toString();
- Comparing current object with another object

    - public boolean equals(Object);
- Used while storing object into set or map collections
    - public native int hashCode();
- Create shallow copy of the object
    - protected native Object clone() throws CloneNotSupportedException;
- Called by garbage collector (like C++ destructor)
    - protected void finalize() throws Throwable;
- Get metadata about the class
    - public final native Class<?> getClass();
- For thread synchronization
    - public final native void notify();
    - public final native void notifyAll();
    - public final void wait() throws InterruptedException;
    - public final native void wait(long) throws InterruptedException;
    - public final void wait(long, int) throws InterruptedException;

toString() method

- Non-final method of java.lang.Object class.
    - public String toString();
- Definition of Object.toString():

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- To return state of Java instance in String form, programmer should override toString() method.
- The result in toString() method should be a concise, informative, and human-readable.
- It is recommended that all subclasses override this method.
- Example:

```
class Person {
    // ...
    @Override
    public String toString() {
        return "Name=" + this.name + ", Age=" + this.age;
    }
}
```