COMP2511

Object Oriented Programming (OOP) in Java

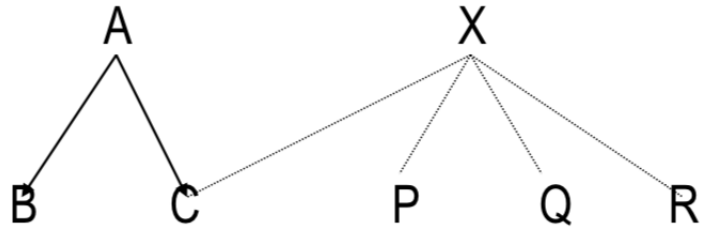
Prepared by

Dr. Ashesh Mahidadia

OOP in Java

- Object Oriented Programming (OOP)
- Inheritance in OOP
- Introduction to Classes and Objects
- Subclasses and Inheritance
- Abstract Classes
- Single Inheritance versus Multiple Inheritance
- Interfaces
- Method Forwarding (Has-a relationship)
- Method Overriding (Polymorphism)
- Method Overloading
- Constructors

Method Forwarding



- Suppose class C extends class A, and also implements interface X.
- As all the methods defined in interface X are abstract, class C needs to implement all these methods.
- However, there are three implementations of X (in P,Q,R).
- In class C, we may want to use one of these implementations, that means, we may want to use some or all methods implemented in P, Q or R.
- Say, we want to use methods implemented in P. We can do this by creating an object of type class P in class C, and through this object access all the methods implemented in P.
- Note that, in class C, we do need to provide required stubs for all the methods in the interface X. In the body of the methods we may simply call methods of class P via the object of class P.
- This approach is also known as **method forwarding**.

Methods Overriding (Polymorphism)

- When a class defines a method using the **same** name, return type, and by the number, type, and position of its arguments as a method in its *superclass*, the method in the class **overrides** the method in the *superclass*.
- If a method is invoked for an object of the class, it's the new definition of the method that is called, and not the superclass's old definition.

Polymorphism

- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called *polymorphism*.

Methods Overriding: Example

In the example below,

- if **p** is an instance of class **B**,
  p.f() refers to **f()** in class **B**.
- However, if p is an instance of class **A**,
  p.f() refers to **f()** in class **A**.

The example also shows how to refer to the overridden method using **super** keyword.

```
class A {
     int i = 1;
     int f() { return i;}
}

class B extends A {
     int i;                          // shadows i from A
     int f() {                       // overrides f() from A
          i = super.i + 1;           // retrives i from A
          return  super.f() + i;     // invokes f() from A
     }
}
```

Methods Overriding: Example

Suppose class C is a subclass of class B, and class B is a subclass of class A.

Class A and class C both define method `f()`.

From class C, we can refer to the overridden method by,

```
super.f()
```

This is because class B inherits method `f()` from class A.

However,

- if all the three classes define `f()`, then calling `super.f()` in class C invokes class B's definition of the method.
- Importantly, in this case, there is no way to invoke `A.f()` from within class C.
- Note that `super.super.f()` is **NOT legal** Java syntax.

Method Overloading

Defining methods with the same name and different argument or return types is called *method overloading*.

In Java,

- a method is distinguished by its method signature - its name, return type, and by the number, type, and position of its arguments

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

Data Hiding and Encapsulation

We can hide the data within the class and make it available only through the methods.

This can help in maintaining the consistency of the data for an object, that means the state of an object.

Visibility Modifiers

Java provides five access modifiers (for variables/methods/classes),

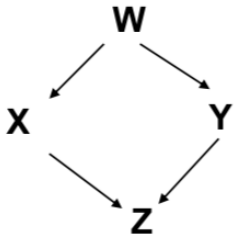
- **public** - visible to the world
- **private** - visible to the class only
- **protected** - visible to the package and all subclasses
- **No modifier (default)** - visible to the package

Constructors

- Good practice to define the required constructors for all classes.
- If a constructor is not defined in a class,
  - **no-argument** constructor is implicitly **inserted**.
  - this no-argument constructor invokes the superclass's no-argument constructor.
  - if the parent class (superclass) doesn't have a visible constructor with no-argument, it results in a compilation **error**.
- If the first statement in a constructor is **not** a call to **super**() or **this**(), a call to **super** () is implicitly inserted.
- If a constructor is defined with one or more arguments, no-argument constructor is **not** inserted in that class.
- A class can have multiple constructors, with different *signatures*.
- The word "**this**" can be used to call another constructor in the same class.

Diamond Inheritance Problem: A Possible Solution

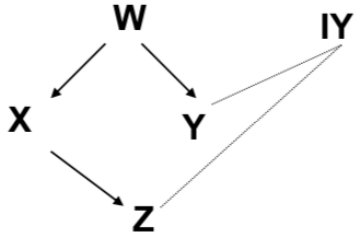
Using **multiple inheritance** (in C++):



we achieve the following:

- In class Z, we can use methods and variables defined in X, W **and** Y.
- Objects of classes Z and Y can be assigned to variables of **type Y.**
- and more ...

Using **single inheritance** in Java:




```
class W { }
interface IY { }
class X extends W { }
class Y extends W implements IY { }
class Z extends X implements IY { }
```

we achieve the following:

- In class Z, we can use methods and variables defined in X and W. In class Z, if we want to use methods implemented in class Y, we can use ***method forwarding*** technique. That means, in class Z, we can create an object of type class Y, and via this object we can access (in class Z) all the methods defined in class Y.
- Objects of classes Z and Y can be assigned to variables of **type IY** (instead of Y).
- and more ....

Some References to Java Tutorials

- https://docs.oracle.com/javase/tutorial/
- https://www.w3schools.com/java/default.asp
- https://www.tutorialspoint.com/java/index.htm