11 Inheritance, Polymorphism, Interfaces

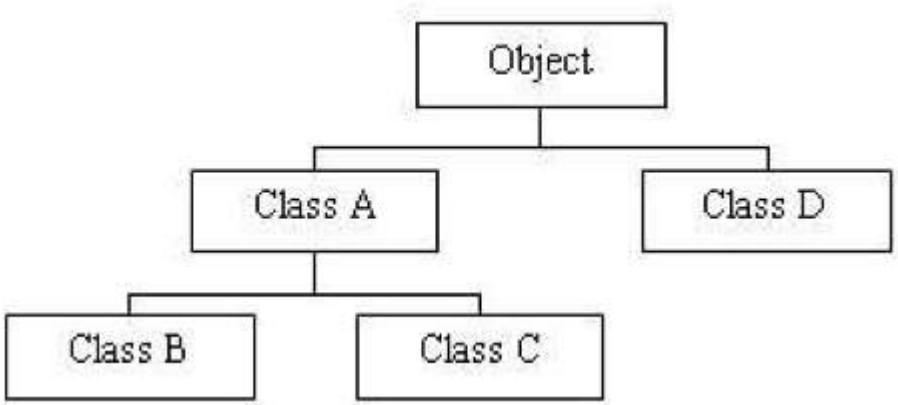
Objectives

At the end of the lesson, the student should be able to:

- Define super classes and subclasses
- Override methods of superclasses
- Create final methods and final classes

Inheritance

- In Java, all classes, including the classes that make up the Java API, are subclassed from the Object superclass.
- A sample class hierarchy is shown below.



- Superclass
  - Any class above a specific class in the class hierarchy.
- Subclass
  - Any class below a specific class in the class hierarchy.

Inheritance

- Superclass
  - Any class above a specific class in the class hierarchy.
- Subclass
  - Any class below a specific class in the class hierarchy.

Inheritance

- Benefits of Inheritance in OOP : Reusability
  - Once a behavior (method) is defined in a superclass, that behavior is automatically inherited by all subclasses.
  - Thus, you can encode a method only once and they can be used by all subclasses.
  - A subclass only needs to implement the differences between itself and the parent.

Inheritance

- To derive a class, we use the extends keyword.
- In order to illustrate this, let's create a sample parent class.
- Suppose we have a parent class called Person.

```
public class Person {
    protected String name;
    protected String address;

    /**
     * Default constructor
     */
    public Person(){
        System.out.println("Inside Person:Constructor");
        name = ""; address = "";
    }
    . . . .
}
```

Inheritance

- Now, we want to create another class named Student.
- Since a student is also a person, we decide to just extend the class Person, so that we can inherit all the properties and methods of the existing class Person.
- To do this, we write,

```
public class Student extends Person {
   public Student(){
      System.out.println("Inside Student:Constructor");
   }
   . . . .
}
```

Inheritance

- When a Student object is instantiated, the default constructor of its superclass is invoked implicitly to do the necessary initializations.
- After that, the statements inside the subclass's constructor are executed.

Inheritance:

- To illustrate this, consider the following code,

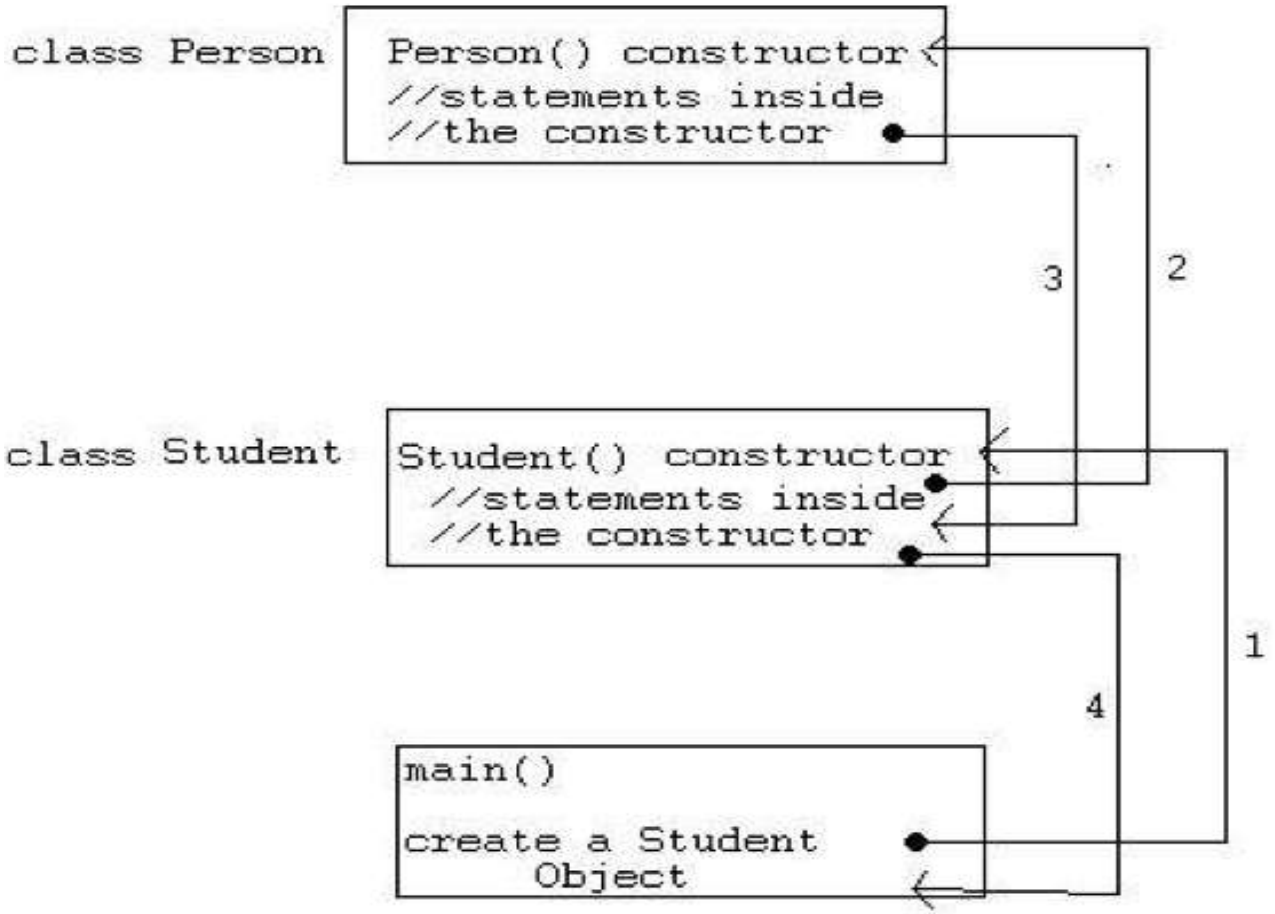
```
public static void main( String[] args ){
   Student anna = new Student();
}
```

- In the code, we create an object of class Student. The output of the program is,

```
Inside Person:Constructor
Inside Student:Constructor
```

Inheritance

- The program flow is shown below.



The “super” keyword

- A subclass can also explicitly call a constructor of its immediate superclass.
- This is done by using the super constructor call.
- A super constructor call in the constructor of a subclass will result in the execution of relevant constructor from the superclass, based on the arguments passed.

The “super” keyword

- For example, given our previous example classes Person and Student, we show an example of a super constructor call.
- Given the following code for Student,

```
public Student(){
    super( "SomeName", "SomeAddress" );
    System.out.println("Inside Student:Constructor");
}
```

The “super” keyword

- Few things to remember when using the super constructor call:
  - The super() call MUST OCCUR AS THE FIRST STATEMENT IN A CONSTRUCTOR.
  - The super() call can only be used in a constructor definition.
  - This implies that the this() construct and the super() calls CANNOT BOTH OCCUR IN THE SAME CONSTRUCTOR.

The “super” keyword

- Another use of super is to refer to members of the superclass (just like the this reference).
- For example,

```
public Student() {
    super.name = “somename”;
    super.address = “some address”;
}
```

Overriding methods

- If for some reason a derived class needs to have a different implementation of a certain method from that of the superclass, overriding methods could prove to be very useful.

- A subclass can override a method defined in its superclass by providing a new implementation for that method.

Example

- Suppose we have the following implementation for the getName method in the Person superclass,

```
public class Person {
    :
    :
    public String getName(){
       System.out.println("Parent: getName");
       return name;
    }
}
```

Example

- To override, the getName method of the superclass Person, we write in the subclass Student,

```
public class Student extends Person{
    :
    :
    public String getName(){
       System.out.println("Student: getName");
       return name;
    }
    :
}
```

- Now, when we invoke the getName method of an object of the subclass Student, the overridden getName method would be called, and the output would be,

```
Student: getName
```

Final Classes

- Final Classes
  - Classes that cannot be extended
  - To declare final classes, we write,

```
public final ClassName{
    . . .
}
```

- Example:

```
public final class Person {
    . . .
}
```

- Other examples of final classes are your wrapper classes and Strings.

Final Methods and Classes

- Final Methods
  - Methods that cannot be overridden
  - To declare final methods, we write,

```
public final [returnType] [methodName]([parameters]){
    . . .
}
```

- Static methods are automatically final.

Example

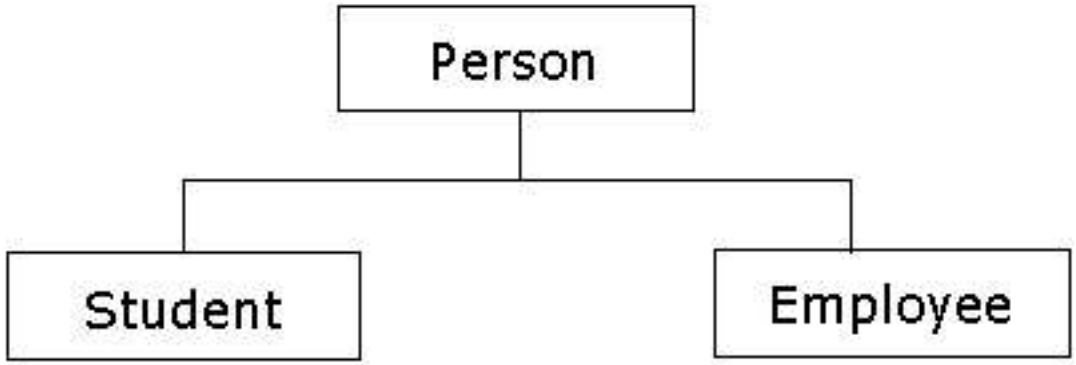
```
public final String getName(){
    return name;
}
```

Polymorphism

- Polymorphism
  - The ability of a reference variable to change behavior according to what object it is holding.
  - This allows multiple objects of different subclasses to be treated as objects of a single superclass, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to.
- To illustrate polymorphism, let us discuss an example.

Polymorphism

- Given the parent class Person and the subclass Student of the previous examples, we add another subclass of Person which is Employee.
- Below is the class hierarchy for that,



Polymorphism

- In Java, we can create a reference that is of type superclass to an object of its subclass. For example,

```
public static main( String[] args ) {

   Person  ref;
   Student studentObject = new Student();
   Employee employeeObject = new Employee();

   ref = studentObject; //Person reference points to a
                        // Student object
}
```

Polymorphism

- Now suppose we have a getName method in our superclass Person, and we override this method in both the subclasses Student and Employee.

```
public class Student {
   public String getName(){
      System.out.println("Student Name:" + name);
      return name;
   }
}

public class Employee {
   public String getName(){
      System.out.println("Employee Name:" + name);
      return name;
   }
}
```

Polymorphism

- Going back to our main method, when we try to call the getName method of the reference Person ref, the getName method of the Student object will be called.

- Now, if we assign ref to an Employee object, the getName method of Employee will be called.

Polymorphism

```
1   public static main( String[] args ) {
2       Person  ref;
3       Student studentObject = new Student();
4       Employee employeeObject = new Employee();
5
6       ref = studentObject; //Person ref. points to a
7                                  // Student object
8
9       //getName of Student class is called
10      String temp=ref.getName();
11      System.out.println( temp );
12
13      ref = employeeObject; //Person ref. points to an
14                                 // Employee object
15
16      //getName of Employee class is called
17      String temp = ref.getName();
18      System.out.println( temp );
19  }
```

Polymorphism

- Another example that illustrates polymorphism is when we try to pass references to methods.

- Suppose we have a static method **printInformation** that takes in a Person reference as parameter.

```
public static printInformation( Person p ){
        . . . .
   }
```

Polymorphism

- We can actually pass a reference of type Employee and type Student to the printInformation method as long as it is a subclass of the class Person.

```
public static main( String[] args )
{
        Student   studentObject = new Student();
        Employee  employeeObject = new Employee();

        printInformation( studentObject );

        printInformation( employeeObject );
}
```

Abstract Classes

- Abstract class
  - a class that cannot be instantiated.
  - often appears at the top of an object-oriented programming class hierarchy, defining the broad types of actions possible with objects of all subclasses of the class.

Abstract Classes

- abstract methods
  - methods in the abstract classes that do not have implementation
  - To create an abstract method, just write the method declaration without the body and use the abstract keyword

- For example,

```
public abstract void someMethod();
```

Sample Abstract Class

```
public abstract class LivingThing {
    public void breath(){
        System.out.println("Living Thing breathing...");
    }

    public void eat(){
        System.out.println("Living Thing eating...");
    }

    /**
     * abstract method walk
     * We want this method to be overridden by subclasses of
     * LivingThing
     */
    public abstract void walk();
}
```

Abstract Classes

- When a class extends the LivingThing abstract class, it is required to override the abstract method walk(), or else, that subclass will also become an abstract class, and therefore cannot be instantiated.

- For example,

```
public class Human extends LivingThing {

   public void walk(){
      System.out.println("Human walks...");
   }

}
```

Coding Guidelines

- Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Interfaces

Interfaces

- An interface
  - is a special kind of block containing method signatures (and possibly constants) only.
  - defines the signatures of a set of methods, without the body.
  - defines a standard and public way of specifying the behavior of classes.
  - allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.
  - NOTE: interfaces exhibit polymorphism as well, since program may call an interface method, and the proper version of that method will be executed depending on the type of object passed to the interface method call.

Why do we use Interfaces?

- To have unrelated classes implement similar methods
  - Example:
    - Class Line and MyInteger
      - Not related
      - Both implements comparison methods
        - isGreater
        - isLess
        - isEqual

Why do we use Interfaces?

- To reveal an object's programming interface without revealing its class

- To model multiple inheritance which allows a class to have more than one superclass

Interface vs. Abstract Class

- Interface methods have no body
- An interface can only define constants
- Interfaces have no direct inherited relationship with any particular class, they are defined independently

Interface vs. Class

- Common:
  - Interfaces and classes are both types
  - This means that an interface can be used in places where a class can be used
  - For example:

```
PersonInterface   pi = new Person();
Person      pc = new Person();
```

- Difference:
  - You cannot create an instance from an interface
  - For example:

```
PersonInterface   pi = new PersonInterface(); //ERROR!
```

Interface vs. Class

- Common:
  - Interface and Class can both define methods
- Difference:
  - Interface does not have any implementation of the methods

Creating Interfaces

- To create an interface, we write:

```
public interface [InterfaceName] {
      //some methods without the body
}
```

Creating Interfaces

- As an example, let's create an interface that defines relationships between two objects according to the “natural order” of the objects.

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Creating Interfaces

- To use an interface, we use the **implements** keyword.
- For example,

```
/**
 * This class defines a line segment
 */
public class Line implements Relation {
      private double x1;
      private double x2;
      private double y1;
      private double y2;

      public Line(double x1, double x2, double y1, double y2){
          this.x1 = x1;
          this.x2 = x2;
          this.y1 = y1;
          this.y2 = y2;
      }
```

Creating Interfaces

```
public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1) +
                        (y2-y1)* (y2-y1));
        return length;
    }

    public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen > bLen);
    }

    public boolean isLess( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen < bLen);

    }

    public boolean isEqual( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen == bLen);
    }
}
```

Creating Interfaces

- When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override
  abstract method isGreater
  (java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
       ^
1 error
```

Relationship of an Interface to a Class

- A class can only EXTEND ONE super class, but it can IMPLEMENT MANY interfaces.
- For example:

```
public class Person implements PersonInterface,
                LivingThing,
                WhateverInterface {

   //some code here
}
```

Relationship of an Interface to a Class

- Another example:

```
public class ComputerScienceStudent extends Student
                implements PersonInterface,
                           LivingThing {

   //some code here
}
```

Inheritance among Interfaces

- Interfaces are not part of the class hierarchy. However, interfaces can have inheritance relationship among themselves
- For example:

```
public interface PersonInterface {
   . . .
}

public interface StudentInterface extends PersonInterface {
   . . .
}
```

Summary

- Inheritance (superclass, subclass)
- Using the super keyword to access fields and constructors of superclasses
- Overriding Methods
- Final Methods and Final Classes
- Polymorphism (Abstract Classes, Interfaces)