Dayananda Sagar College of Engineering

Department of Information Science and Engineering

Course Name: INTRODUCTION TO JAVA

Course Code: 18IS6IEJVA
Semester: 6

Faculty : Mrs.Radhika T V
Assistant Professor , Dept of ISE
DSCE

Module 3:

INHERITANCE ,INTERFACE, EXCEPTION HANDLING

Module 8:
INHERITANCE

Inheritance basics

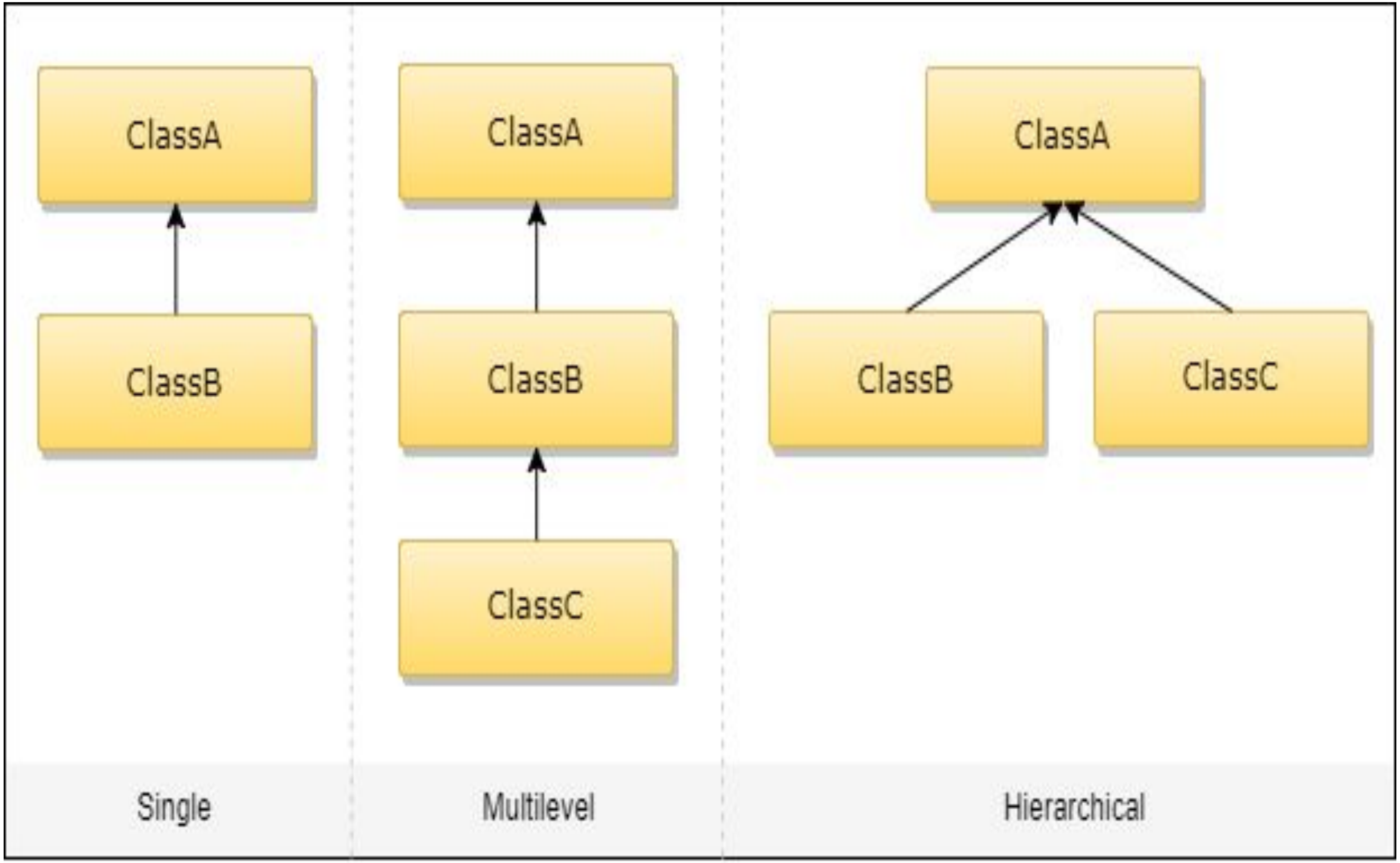
- Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another.
- In the terminology of Java, a class that is inherited is called a *superclass*.
- The class that does the inheriting is called a *subclass*.
- It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- The keyword extends used to inherit the properties of the base class to derived class.

Inheritance basics cntd.,

Syntax

```
class base
{
   .....
   .....
}
class derive extends base
 {
   .....
   .....
}
```

Types of Inheritance



Single Inheritance

When a single class gets derived from its base class, then this type of inheritance is termed as single inheritance.

EXAMPLE:

```
class Teacher {
void teach() {
 System.out.println("Teaching subjects");
}
}
class Students extends Teacher {
 void listen() {
  System.out.println("Listening to teacher");
 }
}
```

```
class CheckForInheritance {
 public static void main(String
args[]) {
  Students s1 = new Students();
  s1.teach();
  s1.listen();
 }
}
```

Multi-level Inheritance

- In this type of inheritance, a derived class gets created from another derived class and can have any number of levels.

EXAMPLE:

```
Class Teacher{
void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
  System.out.println("Listening");
 }
}
class homeTution extends Student {
 void explains() {
  System.out.println("Does homework");
 }
}
```

```
class CheckForInheritance {
 public static void main(String argu[]) {
  homeTution h = new himeTution();
  h.explains();
  h.teach();
  h.listen();
 }
}
```

Hierarchical Inheritance

In this type of inheritance, there are more than one derived classes which get created from one single base class.

```
class Teacher {
 void teach() {
  System.out.println("Teaching subject");
 }
}
class Student extends Teacher {
 void listen() {
  System.out.println("Listening");
 }
}
```

Hierarchical Inheritance cntd.,

```
class Principal extends Teacher {
 void evaluate() {
  System.out.println("Evaluating");
 }
}
 class CheckForInheritance {
 public static void main(String argu[]) {
  Principal p = new Principal();
  p.evaluate();
  p.teach();
  // p.listen(); will produce an error
 }
}
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private.**

Example: // Create a superclass.

```
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
```

CNTD.,

```
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
```

A Superclass Variable Can Reference a Subclass Object

- Areference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
Example: class RefDemo {
public static void main(String args[]) {
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
```

Cntd.,

```
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

Super Keyword in Java

- **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

black white
black white
black white

Example 1: super is used to refer imme parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
Black
white
```

black white
black white
black white
eating... barking...

Example 2: super is used to invoke pare method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
Barking...
```

Example 3: super is used to invoke parent class constructor

```
class Animal{
Animal(){System.out.println("animal is created");
}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
Animal is created
Dog is created
```

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.
- Usage of Java Method Overriding
    - Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
    - Method overriding is used for runtime polymorphism

Method Overriding cntd..,

- Rules for Java Method Overriding
  - The method must have the same name as in the parent class
  - The method must have the same parameter as in the parent class.
  - There must be an IS-A relationship (inheritance).

Example: without method overriding

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child  class object.
//Creating a parent class
class Vehicle{
  void run()
{
System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output:

```
vehicle is running
```

Example: with method overriding

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

Difference between method overloading and method overriding in java

	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.

Abstract class in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- That is, sometimes we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- There are two ways to achieve abstraction in java
  - Abstract class
  - Interface

Cntd.,

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- Rules for abstract class are as follows
  - An abstract class must be declared with an abstract keyword.
  - It can have abstract and non-abstract methods.
  - It cannot be instantiated.
  - It can have constructors and static methods also.
  - It can have final methods which will force the subclass not to change the body of the method.

Example of Abstract class that has an abstract method

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

drawing circle

Example 2: Abstract class

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
```

Output:

```
drawing circle
```

drawing circle

Example 3: Abstract class

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
```

Output:

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member and methods

```
abstract class Bike{
  Bike(){System.out.println("bike is created");}
  abstract void run();
  void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
```

Output:

```
bike is created
gear changed
running safely..
```

Using final with Inheritance

- The keyword **final** has three uses.
- First, it can be used to create the equivalent of a named constant.

Using final to Prevent Overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.

Example:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!"); }}
```

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited.
- To do this, precede the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too.

Example:Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

Chapter 9:
INTERFACE

Defining an Interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also **represents the IS-A relationship**.

Defining an Interface

- However, an interface is different from a class in several ways, including −
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces cntd.,

- The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example 1: /* File name : NameOfInterface.java */

```
import java.lang.*;
// Any number of import statements
public interface NameOfInterface {
   // Any number of final, static fields
 // Any number of abstract method declarations\
}
```

Declaring Interfaces cntd.,

- Interfaces have the following properties −
- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public

Example 2:

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- Toimplement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {

// class-body

}

Implementing Interfaces

Example:

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

- It is both permissible and common for classes that implement interfaces to define additional members of their own.

Implementing Interfaces cntd.,

Example:

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

Implementing Interfaces cntd.,

- **Example 2:**

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
/* File name : MammalInt.java */
public class MammalInt implements Animal {

  public void eat() {
    System.out.println("Mammal eats");
  }

  public void travel() {
    System.out.println("Mammal travels");
  }

  public int noOfLegs() {
    return 0;
  }
```

```
  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

Output:

```
Mammal eats
Mammal travels
```

Implementing Interfaces cntd.,

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Applying Interfaces

- To understand the power of interfaces, let’s look at a more practical example.
- Consider class called **Stack** that implemented a simple fixed-size stack.
- For example, the stack can be of a fixed size or it can be “growable.”
- No matter how the stack is implemented, the interface to the stack

remains the same.

- That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- Example: module3-fixedstack.docx

Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.
- It is as if that class were importing the constant fields into the class name space as **final** variables.
- The example below uses this technique to implement an automated “decision maker”:Module 3-variableOnterface.docx