Java Programming

Object Oriented Programming II

Module 4

Agenda

1. Method Overriding
2. Runtime Polymorphism
3. Abstract classes
4. instanceof Operator
5. Garbage Collection

Objectives

At the end of this module, you will be able to:

- Describe the usage of method overriding
- Describe the importance of Runtime Polymorphism
- Describe the usage of abstract classes
- Describe the usage of instanceof operator
- Describe the functioning of Garbage Collector

Method Overriding

Method Overriding

- When a method in a subclass has the same prototype as a method in the superclass, then the method in the subclass is said to override the method in the superclass
- When an overridden method is called from an object of the subclass, it will always refer to the version defined by the subclass
- The version of the method defined by the superclass is hidden or overridden

```
class A{
  int a,b;
  A(int m, int n){
    a = m;
    b = n;
  }
```

Method Overriding (Contd.).

```
  void display(){
    System.out.println("a and b are :" + a +" " +
  b);
  }
}
class B extends A{
  int c;
  B(int m, int n, int o){
    super(m,n);
    c = o;
  }
  void display() {
    System.out.println("c :" + c);
  }
}
```

Method Overriding (Contd.).

```
class OverrideDemo{
  public static void main(String args[]){
    B subOb = new B(4,5,6);
    subOb.display();
  }
}
```

Using super to Call an Overridden Method

```
class A{
  int a,b;
  A(int m, int n){
    a = m;
    b = n;
  }
  void display(){
    System.out.println("a and b are :" + a +" " +
  b);
  }
}
class B extends A{
  int c;
```

Using super to Call an Overridden Method (Contd.).

```
  B(int m, int n, int o){
    super(m,n);
    c = o;
  }
  void display() {
    super.display();
    System.out.println("c :" + c);
  }
}
class OverrideDemo{
  public static void main(String args[]){
    B subOb = new B(4,5,6);
    subOb.display();
  }
}
```

Superclass Reference Variable

A reference variable of type superclass can be assigned a reference to any subclass object derived from that superclass.

```
class  A1 {
}

class A2 extends A1 {
}

class A3 {
   public static void main(String[] args) {
      A1 x;
      A2 z = new A2();
      x = new A2();//valid
      z = new A1();//invalid
   }
```

A Superclass Reference Variable Can Reference a Subclass Object

- Method calls in Java are resolved dynamically at runtime
- In Java all variables know their dynamic type
- Messages (method calls) are always bound to methods on the basis of the dynamic type of the receiver
- This method resolution is done dynamically at runtime

Rules for method overriding

- Overriding method must satisfy the following points:
  - They must have the same argument list.
  - They must have the same return type.
  - They must not have a more restrictive access modifier
  - They may have a less restrictive access modifier
  - Must not throw new or broader checked exceptions
  - May throw fewer or narrower checked exceptions, or any unchecked exceptions.
- Final methods cannot be overridden.
- Constructors cannot be overridden

Will be explained later with Packages

Will be explained later with Exception Handling

Quiz

- What will be the result, if we try to compile and execute the following code :

```
class A1 {
   void m1() {
    System.out.println("In method m1 of A1");
  }
}
class A2 extends A1 {
  int m1() {
    return 100;
  }
  public static void main(String[] args) {
    A2 x = new A2();
    x.m1();
  }
}
```

Compilation Error..What is the reason?

Why Overridden Methods? A Design Perspective

- Overridden methods in a class hierarchy is one of the ways that Java implements the "**single interface, multiple implementations**" aspect of polymorphism

- Part of the key to successfully applying polymorphism is understanding the fact that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization

Why Overridden Methods? A Design Perspective (Contd.).

- The superclass provides all elements that a subclass can use directly
- It also declares those methods that the subclass must implement on its own
- This allows the subclass the flexibility to define its own method implementations, yet still enforce a consistent interface
- In other words, the subclass will override the method in the superclass

Runtime Polymorphism

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding forms the basis of one of Java's most powerful concepts: **dynamic method dispatch**
- Dynamic method dispatch occurs when the Java language resolves a call to an overridden method at runtime, and, in turn, implements runtime polymorphism
- Java makes runtime polymorphism possible in a class hierarchy with the help of two of its features:
  - superclass reference variables
  - overridden methods

Dynamic Method Dispatch or Runtime Polymorphism (Contd.)

- A superclass reference variable can hold a reference to a subclass object
- Java uses this fact to resolve calls to overridden methods at runtime
- When an overridden method is called through a superclass reference, Java determines which version of the method to call based upon the type of the object being referred to at the time the call occurs

Overridden Methods & Runtime Polymorphism - An Example

```
class Figure {
  double dimension1;
  double dimension2;

  Figure(double x, double y){
    dimension1 = x;
    dimension2 = y;
  }

  double area() {
    System.out.println("Area of Figure is
  undefined");
    return 0;
  }
}
```

Overridden Methods & Runtime Polymorphism - An Example (Contd.).

```
class Rectangle extends Figure {
    Rectangle(double x, double y) {
       super(x,y); }
     double area() //method overriding
     {
      System.out.print("Area of rectangle is :");
      return dimension1 * dimension2;
    }
}
class Triangle extends Figure {
  Triangle(double x, double y) {  super(x,y); }
double area()  //method overriding {
    System.out.print("Area for triangle is :");
    return dimension1 * dimension2 / 2;
  }
}
```

Overridden Methods & Runtime Polymorphism - An Example (Contd.).

```
class FindArea {
  public static void main(String args[]){
    Figure f          = new Figure(10,10);
    Rectangle r       = new Rectangle(9,5);
    Triangle t        = new Triangle(10,8);
    Figure fig;    //reference variable

    fig = r;
    System.out.println("Area of rectangle is :" +
  fig.area());
    fig = t;
    System.out.println("Area of triangle is :" +
  fig.area());
    fig = f;
    System.out.println(fig.area());
  }
}
```

Runtime Polymorphism – Another Example

```
class BigB {
public void role() {
  System.out.println(" My name is BigB");
}
}

class FatherRole extends BigB
{
  // child class is overriding the role() method

public void role(){
    System.out.println("My role is Father when I am
  with my son !");
}
}
```

Runtime Polymorphism – Another Example (Contd.).

```
class DriverRole extends BigB{
  //child class is overriding the name() method
public void role(){
  System.out.println(“ My role is Driver when I am
  driving a car!");
}
}

class CEORole extends BigB{
  //child class is overriding the name() method
public void role(){
  System.out.println(“ My role is CEO when I am
  inside my own company ");
}
}
```

Runtime Polymorphism – Another Example (Contd.).

```
public   class Dyanmic_dispatch  {
public static void main(String ss[])  {

  System.out.println(“ To demonstrate Runtime
  Polymorphism: ");

   BigB v;
// Parent class reference variable can point to
// any of its CHILD class objects....

  v = new BigB();          v.role();

  v= new FatherRole();       v.role();

  v= new DriverRole();       v.role();

  v= new CEORole();        v.role();
}
}
```

Abstract classes

Abstract Classes

- Often, you would want to define a superclass that declares the structure of a given abstraction without providing the implementation of every method
- The objective is 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 provide for its own specific implementations
  - Such a class determines the nature of the methods that the subclasses ***must implement***
  - Such a superclass is unable to create a meaningful implementation for a method or methods

Abstract Classes (Contd.).

- The class **Figure** in the previous example is such a superclass.
  - Figure is a pure geometrical abstraction
  - You have only kinds of figures like **Rectangle**, **Triangle** etc. which actually are subclasses of class **Figure**
  - The class **Figure** has no implementation for the **area( )** method, as there is no way to determine the area of a **Figure**
  - The **Figure** class is therefore a partially defined class with no implementation for the **area( )** method
  - The definition of **area()** is simply a placeholder

Abstract Classes (Contd.).

- abstract method -- a mechanism which shall ensure that a subclass must compulsorily override such methods.
- Abstract method in a superclass has to be overridden by all its subclasses.
- The subclasses cannot make use of the abstract method that they inherit directly(without overriding these methods).
- These methods are sometimes referred to as subclasses' responsibility as they have no implementation specified in the superclass

Abstract Classes (Contd.).

- To use an abstract method, use this general form: **abstract type name(parameter-list);**
- Abstract methods do not have a body
- Abstract methods are therefore characterized by the lack of the opening and closing braces that is customary for any other normal method
- This is a crucial benchmark for identifying an abstract class

Abstract Classes (Contd.).

- Any class that contains one or more abstract methods **must** also be declared abstract
  - It is perfectly acceptable for an abstract class to implement a concrete method
  - You cannot create objects of an abstract class
  - That is, an abstract class cannot be instantiated with the new keyword
  - Any subclass of an abstract class must **either implement all of the abstract methods in the superclass**, or **be itself declared abstract.**

Revised Figure Class – using abstract

- There is no meaningful concept of area( ) for an undefined two-dimensional geometrical abstraction such as a Figure

- The following version of the program declares area( ) as abstract inside class Figure.

- This implies that class Figure be declared abstract, and all subclasses derived from class Figure must override area( ).

Improved Version of the Figure Class Hierarchy

```
abstract class Figure{
  double dimension1;
  double dimension2;
  Figure(double x, double y){
    dimension1 = x;
    dimension2 = y;
  }
  abstract double area();
}
```

Improved Version of the Figure Class Hierarchy (Contd.).

```
class Rectangle extends Figure{
  Rectangle(double x, double y){
    super(x,y);    }

  double area(){
    System.out.print("Area of rectangle is :");
    return dimension1 * dimension2;
  }
}

class Triangle extends Figure{
  Triangle(double x, double y){   super(x,y); }
  double area(){
    System.out.print("Area for triangle is :");
    return dimension1 * dimension2 / 2;
  }
}
```

Improved Version of the Figure Class Hierarchy (Contd.).

```
class FindArea{
  public static void main(String args[]){
    Figure fig;
    Rectangle r = new Rectangle(9,5);
    Triangle t  = new Triangle(10,8);
    fig = r;
    System.out.println("Area of rectangle is :" +
  fig.area());
    fig = t;
    System.out.println("Area of triangle is :" +
  fig.area());
  }
}
```

The Role of the Keyword final in Inheritance

- The **final** keyword has two important uses in the context of a class hierarchy. These uses are highlighted as follows:
- Using final to Prevent Overriding
  - While method overriding is one of the most powerful feature of object oriented design, there may be times when you will want to prevent certain critical methods in a superclass from being overridden by its subclasses.
  - Rather, you would want the subclasses to use the methods as they are defined in the superclass.
  - This can be achieved by declaring such critical methods as final.

The Role of the Keyword final in Inheritance (Contd.).

- Using final to Prevent Inheritance
  - Sometimes you will want to prevent a class from being inherited.
  - This can be achieved by preceding the class declaration with final.
  - Declaring a class as final implicitly declares all of its methods as final too.
  - It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide concrete and complete implementations.

instanceof Operator

Use of instance of operator

The instanceof operator in java allows you determine the type of an object

The instanceof operator compares an object with a specified type

It takes an object and a type and returns true if object belongs to that type. It returns false otherwise.

We use instanceof operator to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements an interface

instanceof operator in real world!

In this image, a passport is being verified.
This is an example for usage of instanceof operator in real world.

instanceof operator example

```
class A {
    int i, j;
}
class B extends A {
    int a, b;
}
class C extends A {
    int m, n;
}
```

Contd..

instanceof operator example 1

```
class InstanceOfImpl 1{
    public static void main(String args[ ]) {
        A a = new A( );
        B b = new B( );
        C c = new C( );
        A ob=b;
        if (ob instanceof B)
            System.out.println("ob now refers to B");
        else
            System.out.println("Ob is not instance of B");
        if (ob instanceof A)
            System.out.println("ob is also instance of A");
        else
            System.out.println("Ob is not instance of A");
        if (ob instanceof C)
            System.out.println("ob now refers to C");
        else
            System.out.println("Ob is not instance of C");
    }
}
```

ob now refers to B
ob is also instance of A
Ob is not instance of C

instanceof operator example 2

```
class InstanceOfImpl 1{
    public static void main(String args[ ]) {
        A a = new A( );
        B b = new B( );
        C c = new C( );
        A ob=c;
        if (ob instanceof B)
            System.out.println("ob now refers to B");
        else
            System.out.println("Ob is not instance of B");
        if (ob instanceof A)
            System.out.println("ob is also instance of A");
        else
            System.out.println("Ob is not instance of A");
        if (ob instanceof C)
            System.out.println("ob now refers to C");
        else
            System.out.println("Ob is not instance of C");
    }
}
```

Ob is not instance of B
ob is also instance of A
ob now refers to C

The Cosmic Class – The Object Class

- Java defines a special class called **Object.** It is available in java.lang package
- All other classes are subclasses of **Object**
- **Object** is a superclass of all other classes; i.e., Java's own classes, as well as user-defined classes
- This means that a reference variable of type **Object** can refer to an object of any other class

The Cosmic Class – The Object Class (Contd.).

- Object defines the following methods, which means that they are available in every object

Method	Explanation
Object clone()	Create a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is reclaimed from the heap by the garbage collector
final Class getClass()	Obtains the class of an object at runtime
int hashCode	Returns the hash code associated with the invoking object
final void notify()	Resumes execution of a thread waiting on the invoking object
final void notifyAll()	Resumes execution of all waiting threads on the invoking object.
String toString()	Returns a string that describes the object.
final void wait final void wait(long milliseconds) final void wait(long milliseconds, long nanoseconds)	Waits on another thread of execution.

Garbage Collection

Java's Cleanup Mechanism – The Garbage Collector

- Objects on the heap must be deallocated or destroyed, and their memory released for later reallocation
- Java handles object deallocation automatically through **garbage collection**
- Objects without references will be reclaimed

The finalize( ) Method

- Often, an object needs to perform some action when it is destroyed

- The action could pertain to:
  - releasing a file handle
  - reinitializing a variable, such as a counter

- Java’s answer is a mechanism called finalization

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector

The finalize( ) Method (Contd.).

- To add a finalizer to a class, you simply define the finalize( ) method
- The Java runtime calls that method whenever it is about to recycle an object of that class
- Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed

Example Demonstrating finalize() method

```
public class CounterTest {
    public static int count;
        public CounterTest ( ) {
        count++;
    }
}

  public static void main(String args[ ])   {
          CounterTest ob1 = new CounterTest ( ) ;
        System.out.println("Number of objects :" +
                  CounterTest.count) ;
        CounterTest ob2 = new CounterTest ( );
        System.out.println("Number of objects :" +
               CounterTest.count) ;
```

Example Demonstrating finalize() method(Contd.).

```
Runtime r = Runtime.getRuntime( );
ob1 = null;
ob2 = null;
r.gc( );
}
protected void finalize( ) {
System.out.println("Program about to terminate");
      CounterTest.count --;
      System.out.println("Number of objects :" +
         CounterTest.count) ;
   }
}
```

Output:

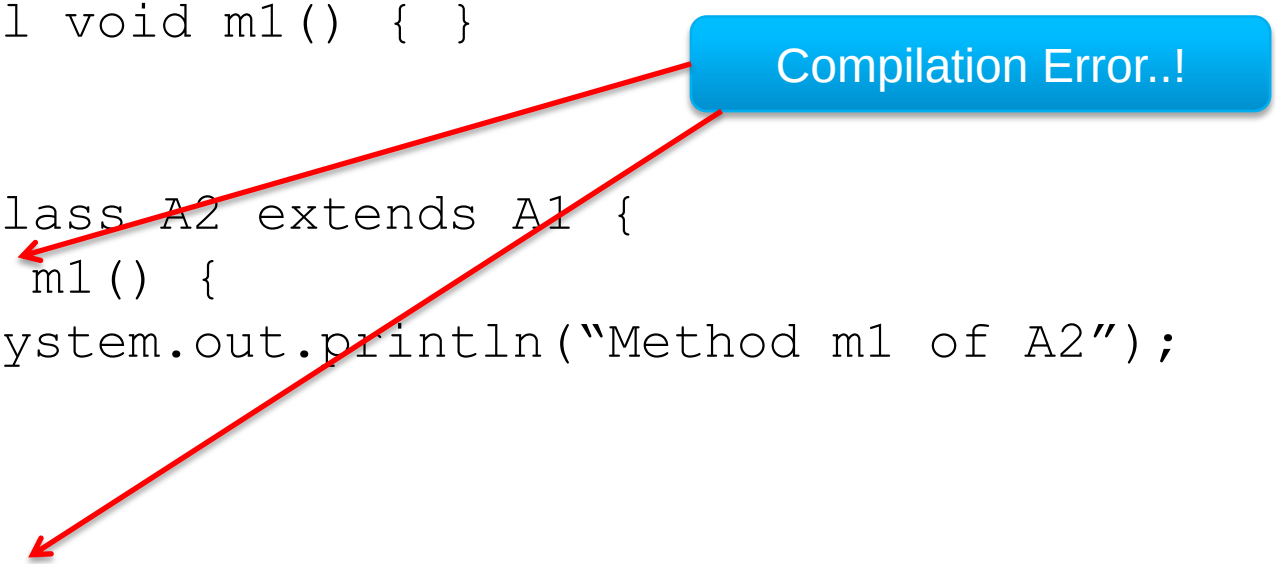
Number of objects :1
Number of objects :2
Program about to terminate
Number of objects :1
Program about to terminate
Number of objects :0

Point out the errors in the following code :

```
1. class A1 {
2.     final void m1() { }
3. }

4. final class A2 extends A1 {
5.     void m1() {
6.         System.out.println("Method m1 of A2");
7.     }
8. }

9. class A3 extends A2{
10.    public static void main(String[] args) {
11.        System.out.println("Executed
   Successfully");
12.    }
13.}
```

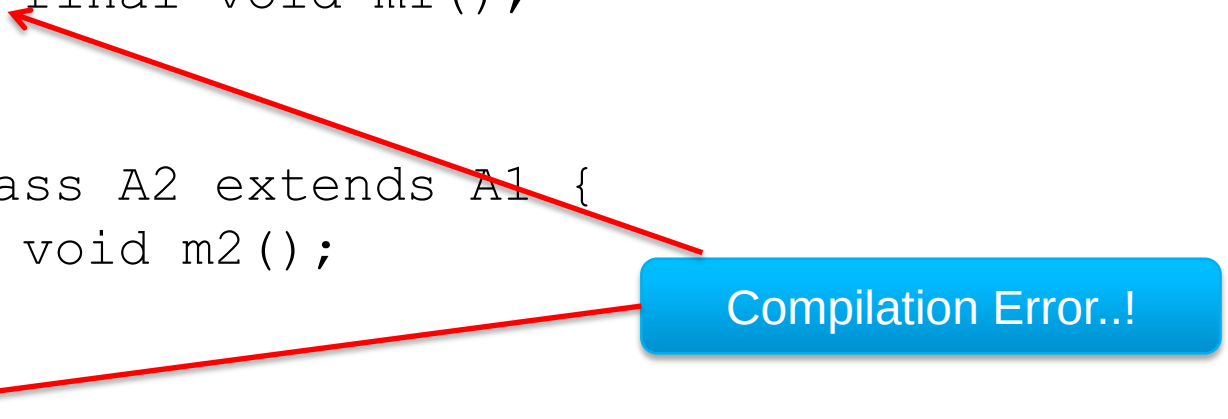


Point out the errors in the following code : (Contd.).

```
1. abstract class A1 {
2.     abstract final void m1();
3. }

4. abstract class A2 extends A1 {
5.     abstract void m2();
6. }

7. class A3 extends A2{
8.     public static void main(String[] args) {
9.         System.out.println(“Executed
   Successfully");
10.    }
11.}
```



Summary

- In this session, you were able to:

- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Describe abstract classes
- Define finalize method
- Get basic information about garbage collection

References

1. Schildt, H. Java: *The Complete Reference. J2SETM*. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.
2. javapoint.com (2012). *Runtime Polymorphism*. Retrieved on April 3, 2012, from, http://www.javatpoint.com/sonoojaiswal/runtime-polymorphism-in-java
3. Kamini (2012). *Java Samples:Abstract classes in Java.* Retrieved on April 13, 2012, from, http://www.javasamples.com/showtutorial.php?tutorialid=288

Thank You