COCSIT

METHOD OVERRIDDING / POLYMORPHISM

Method Overriding / Polymorphism

Agenda

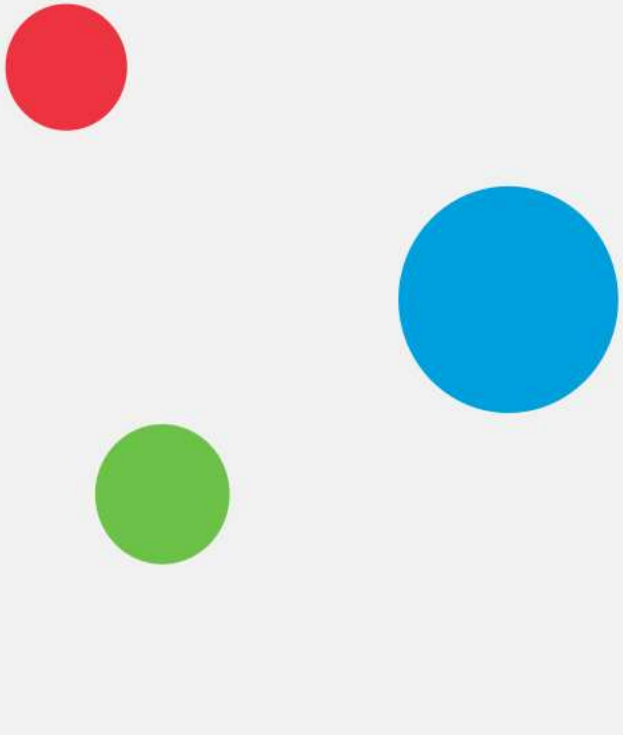
1. Method Overriding
2. Runtime Polymorphism
3. instanceof Operator

Objectives

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

- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Understand the use of instanceof operator

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
- A method in a subclass has the same prototype as a method in its superclass if it has the same name, type signature (the same type, sequence and number of parameters), and the same return type as the method in its superclass. In such a case, a method in a subclass is said to override a method in its superclass.

Method Overriding(Contd.).

```
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);
  }
}
```

Method Overriding (Contd.).

```
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();
  }
}
```

The output produced by this program is shown below:

c: 6

When display() is invoked on an object of class B, the version of display() defined within B is invoked. In other words, the version of display() inside B **overrides** the version of display declared in its superclass, i.e., class A.

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();
  }
}
```

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

- The following is the output of the aforesaid program:

  a and b: 4 5
  c: 6

- Method overriding occurs only when the names and the type signatures of two methods across at least two classes (i.e., a superclass and a subclass) in a class hierarchy are identical.
- If they are not, then the two methods are simply overloaded.
- Elaborating this theme further, if the method in a subclass has a different signature or return type than the method in the superclass, then the subclass will have two forms of the same method.

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

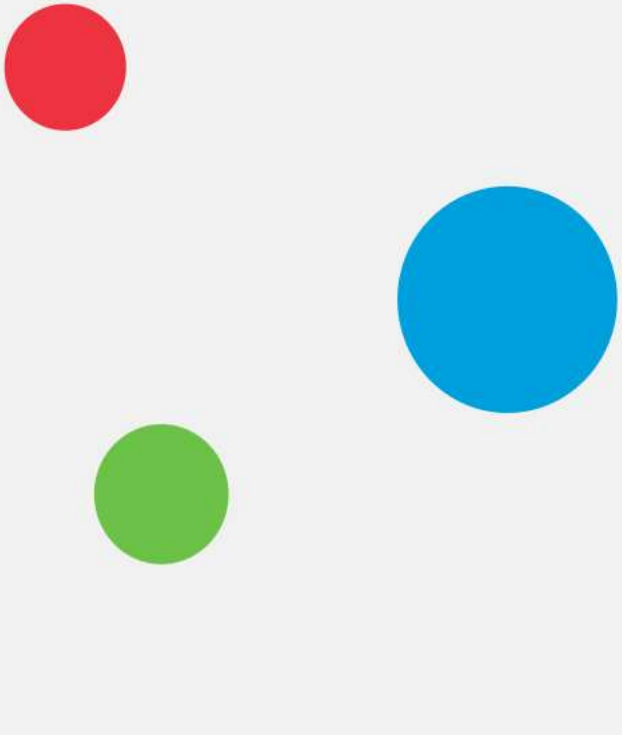
- Overridden methods in a class hierarchy is one of the ways that Java implements the "**single interface, multiple implementations**" aspect of polymorphism
- Part of the key to successfully applying polymorphism is understanding the fact that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization
- The superclass provides all elements that a subclass can use directly
- It also declares those methods that the subclass must implement on its own

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

- 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 and 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 and 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 and 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();
}
}
```

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.

Quiz

What would be the output of the following program?

```
class Sample{
 public static void main(String args[]){
 Sample s=new Sample();
 System.out.println(s instanceof Sample);
 }
}
```

- True
- False
- Compilation error
- Runtime error

Quiz(Contd.).

- Which of the following classes will be the parent class of all user- defined classes in Java?
  a. Object
  b. Class
  c. String
  d. No parent class for user defined classes

Quiz- Solutions

- Answer 1:

  True

- Answer 2:

  Object

Summary

- In this session, you were able to:
  - Describe method overriding
  - Describe dynamic method dispatch, or runtime polymorphism
  - Understand the use of instanceof operator

Thank You