SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE NAME : 23CSB101- OBJECT ORIENTED PROGRAMMING

I YEAR /II SEMESTER

Unit II – INHERITANCE, PACKAGES AND INTERFACES

Topic : super() - DYNAMIC METHOD DISPATCH - ABSTRACT CLASS

CONSTRUCTORS IN SUB – CLASSES

- Constructors are called in order of derivation, from superclass to subclass.

Example:

```
class A
{
A()
{
System.out.println(“ Inside A’s Constructor”);
}
}

class B extends A
{
B()
 {
System.out.println(“ Inside B’sConstructor”);
}
}
class C extends B
{
  C()
  {
System.out.println(“ Inside C’s Constructor”);
     }
}
class CallingCons
{
  public static void main(String args[])
 {
    C objC=new C();
  }
}
```

CONSTRUCTORS IN SUB – CLASSES

Inside A's Constructor

Inside B's Constructor

Inside C's Constructor

Three classes A, B and C created using multilevel inheritance concept. Here, constructors of the three classes are called in the order of derivation. Since **super()** must be the first statement executed in subclass's constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. **When inheriting from another class, super()** has to be **called first** in the constructor. Otherwise the compiler will insert that call. This is why super constructor is also invoked when a Sub object is created.

CONSTRUCTORS IN SUB – CLASSES

- When **compiler inserts the super constructor**, the sub class constructor looks like

```
B()
{
super();
System.out.println("Inside B's Constructor");
} C()
{
super();
System.out.println("Inside C's Constructor");
}
```

“super” keyword

- ✓Super is a keyword that directs the compiler to invoke the superclass members.
- ✓It is used to refer to the parent class of the class in which the keyword is used.
- ✓**super keyword is used for the following three purposes**:
  1. To invoke superclass constructor.
  2. To invoke superclass members variables.
  3. To invoke superclass methods.

“super” keyword

1. Invoking a superclass constructor:

- ✓**super** as a standalone statement(ie. super()) represents a call to a constructor of the superclass.
- ✓A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

```
super();
or
super(parameter-list);
```

“super” keyword

- ✓Here, parameter-list specifies any parameters needed by the constructor in the superclass.
- ✓**super( )** must always be the first statement executed inside a subclass constructor.
- ✓The compiler implicitly calls the base class’s no-parameter constructor or default constructor.
- ✓If the superclass has parameterized constructor and the subclass constructor does not call superclass constructor explicitly, then the Java compiler reports an error.

“super” keyword

2. Invoking a superclass members (variables and methods):

(i) Accessing the instance member variables of the superclass:

Syntax:

```
super.membervariable;
```

(ii) Accessing the methods of the superclass:

- This call is particularly necessary while calling a method of the super class that is overridden in the subclass.

Syntax:

```
super.methodName();
```

“super” keyword

- ✓If a parent class contains a finalize() method, it must be called explicitly by the derived class’s finalize() method.

```
super.finalize();
```

“super” keyword

Example:

```
class A    // super class
{
int i;
A(String str)    //superclass constructor
{
System.out.println(" Welcome to "+str);
}
void show()
{
System.out.println(" Thank You!");
}
}
```

```
class B extends A
{
int i;
// hides the superclass variable i.
B(int a, int b)
  // subclass constructor
{
super("Java Programming");
// invoking superclass constructor
super.i=a;
//accessing superclass member variable
i=b;
System.out.println(“ i in superclass :"+super.i);
System.out.println(" i in subclass :"+i);
}
```

CONT…..

“super” keyword

```
        }
    }
    public class UseSuper {
      public static void main(String[] args)
      {
        B objB=new B(1,2);
// subclass object construction
        objB.show();
// call to  subclass method show()
}
    }
```

OUTPUT:

Welcome to Java Programming
i in superclass : 1
i in subclass : 2
Thank You!

“super” keyword

In the above program, we have created the base class named **A** that contains a instance variable **‘i’** and a method **show().** Class A contains a parameterized constructor that receives string as a parameter and prints that string. Class **B** is a subclass of **A** which contains a instance variable ‘i’ (hides the superclass variable ‘i’) and overrides the superclass method **show().** The subclass defines the constructor with two parameters a **and b.** The subclass constructor invokes the superclass constructor **super(String)** by passing the string “Java Programming” and assigns the value **a** to the superclass variable**(super.i=a)** and **b** to the subclass variable. The show() method of subclass prints the values of ‘i’ form both superclass and subclass & invokes the superclass method as **super.show().**

In the main class, object for subclass **B** is created and the object is used to invoke **show()** method of subclass.

DYNAMIC METHOD DISPATCH

- **Dynamic method dispatch** is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Java implements run-time polymorphism using Dynamic method dispatch

DYNAMIC METHOD DISPATCH

Example

```
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
//override callme()
void callme() {
System.out.println("Inside B's callme method");
} }
```

```
class C extends A
{
//override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
```

CONT…

DYNAMIC METHOD DISPATCH

```
class Dispatch
{
public static void main(String args[])
{
A  a=new A();     //object of type A
B b=new B();      //object of type B
C c=new C();      //object of type C
A  r;// obtain a reference of type A
```

```
r = a;     // r refers to an A object
// dynamic method dispatch
r.callme();       // calls A's version of callme()
r = b;   // r refers to an B object
r.callme();  // calls B's version of callme()
r = c; // r refers to an C object
r.callme();   // calls C's version of callme()
}
}
```

OUTPUT :

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

ABSTRACT CLASSES

Abstraction:

•Abstraction lets you focus on what the object does instead of how it does it.

•For example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

ABSTRACT CLASSES

- A class that is declared as abstract is known as **abstract class**.
- Abstract classes cannot be instantiated, but they can be subclassed.

Syntax

```
abstract class <class_name>
{
Member variables;
Concrete methods { }
Abstract methods();
}
```

Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

ABSTRACT CLASSES

Properties of abstract class:

- **abstract** keyword is used to make a class abstract.
- Abstract class can’t be instantiated.
- If a class has abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Abstract classes can have both concrete methods and abstract methods.
- The subclass of abstract class must implement all the abstract methods unless the subclass is also an abstract class.
- A constructor of an abstract class can be defined and can be invoked by the subclasses.
- We can run abstract class like any other class if it has main() method.

ABSTRACT CLASSES

Example:

```
abstract class GraphicObject { int x, y;
...
void moveTo(int newX, int newY) {
...
}
abstract void draw(); abstract void resize();
}
```

Abstract Methods

A method that is declared as abstract and does not have implementation is known as **abstract method.** It acts as placeholder methods that are implemented in the subclasses.

Syntax to declare a abstract method:

```
abstract class classname
{
abstract return_type
<method_name>(parameter_list);//no braces{}
// no implementation required
……..
}
```

Abstract Methods

Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Properties of abstract methods:

- The abstract keyword is also used to declare a method as abstract.
- An abstract method consists of a method signature, but no method body.
- If a class includes abstract methods, the class itself must be declared abstract.
- Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee { private String name;
private String address; private int number;
public abstract double computePay();
//Remainder of class definition
}
```

Abstract Class

Example program lab exercise number 4

Write a Java program to create an abstract class named Shape that contains 2 integers and an empty method named PrintArea(). Provide 3 classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contain only the method PrintArea() that prints the area of the given shape.

Example

```
// Abstract class Shape
abstract class Shape {
    int x, y;

    // Constructor
    Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Abstract method
    abstract void printArea();
}


// Rectangle class extending Shape
```

```
class Rectangle extends Shape {
    // Constructor
    Rectangle(int length, int breadth) {
        super(length, breadth);
    }



    @Override
    void printArea() {
   System.out.println("Rectangle Area: " + (x * y));
    }
}
```

Example

```
// Triangle class extending Shape
class Triangle extends Shape {
    // Constructor
    Triangle(int base, int height) {
        super(base, height);
    }

    @Override
    void printArea() {
  System.out.println("Triangle Area: " + (0.5 * x * y));
    }
}
```

```
// Circle class extending Shape

class Circle extends Shape {
    // Constructor (Only one parameter is needed for radius)
     Circle(int radius) {
   super(radius, 0);
// second parameter is not used
     }

 @Override
    void printArea() {
 System.out.println("Circle Area: " + (Math.PI * x * x));
    }
}
```

Example

```
// Main class to test the program
public class ShapeDemo {
    public static void main(String[] args) {
        Shape rectangle = new Rectangle(10, 5);
        rectangle.printArea();
// Output: Rectangle Area: 50

        Shape triangle = new Triangle(8, 4);
        triangle.printArea();
// Output: Triangle Area: 16.0

        Shape circle = new Circle(7);
        circle.printArea();
// Output: Circle Area: 153.93804002589985
    }
}
```


Thank
you