CORE JAVA

MANUAL V8.3

MODULE CODE:

ANUDIP FOUNDATION


ICONS AND THEIR MEANING
HINTS:
Get ready for helpful insites on difficult topics and questions.
STUDENTS:
This icon symbolize important instrcutions and guides for the students.
TEACHERS/TRAINERS:
This icon symbolize important instrcutions and guides for the trainers.

Module 2: Object Oriented Programming and Package

Chapter 2

Objective: After completing this lesson you will be able to : * Gain an understanding of Java constructors and constructor overloading * Gain an introduction to inheritance in Java	**Materials Required:** 1. Computer 2. Internet access
Theory Duration: 60 minutes	**Practical Duration:** 60 minutes
Total Duration: 120 minutes	

Chapter 2

Constructor, Constructor Overloading, Inheritance

2.1 Constructor

In Java, constructor refers to a code block that initializes new objects. A Java constructor is similar to an instance method despite not being a method itself. It does not possess a return type. Constructors are often referred to as special method type.

A constructor name is the same as a class name. The three sections of a constructor are -

* Constructor name
* Access modifier
* Parameters

Example of constructor syntax:

```
public class MyClass{
   MyClass( ){
   }
   ..
}
```

The three constructor types are -

i) Default

ii) No-arg

iii) Parameterized

i) **Default constructor –** A default constructor is assigned to a Java class if no constructor is written by a programmer. This constructor is not a part of source code but becomes a part of the code during the compilation process. A Java

default constructor initializes member data variables to produce default values. It initializes booleans as 'false', numeric values as '0', and references as 'null'. It is a 'nullary' constructor as it does not take any arguments.

Example of default constructor

```
package com.myjava.constructors;
public class MyDefaultConstructor {
        public MyDefaultConstructor( ){
System.out.println('This is default constructor.' );
   }
 public static void main(String a[]){
   MyDefaultConstructor mdc = new MyDefaultConstructor( );
   }
}
```

Output: This is default constructor.

ii) **No-arg constructor –** A constructor that does not have any arguments is referred to as a no-arg constructor. It has similar syntax to a default constructor. However, the body section of a no-arg constructor can contain code. Default and no-arg constructor cannot be considered the same due to the presence of code in a no-arg.

Example: no-arg constructor

```
class Demo
{
    public Demo( )
    {
        System.out.println('Constructor is no argument type.' );
```

```
    }

    public static void main(String args[]) {
        new Demo();
    }
}
```

Output:

Constructor is no argument type.

iii) Parameterized constructor

A parameterized constructor is one with parameters or arguments. They are used for passing parameters for object creation.

Example: parameterized constructor

```
public class Employee {

  int empId;
  String empName;
  Employee(int id, String name){
     this.empId = id;
     this.empName = name;
  }
  void info(){
      System.out.println('Id: '+empId+' Name: '+empName);
  }
  public static void main(String args[]){
        Employee obj1 = new Employee(100,'Jack');
```

```
        Employee obj2 = new Employee(109,'John');
        obj1.info();
        obj2.info();
    }
}
```

Output:

Id: 100 Name: Jack

Id: 109 Name: John

ii) Constructor Overloading

Constructor overloading is a Java process where a class is capable of having multiple constructors with different parameter lists. The constructors are differentiated by the Java compiler by considering the number and type of parameters in the list. Each differentiated constructor can be made to perform varying tasks.

Overloading enables programmers to use default constructors along with constructors that have parameters.

Program exhibiting constructor overloading

```
class Student{
 int id;
   String name;
int age;
   Student(int i,String n){
 id = i;
name = n;
 }
 Student(int i,String n,int a){
id = i;
```

```
name = n;
age=a;
   }
void display(){System.out.println(id+" "+name+" "+age);}
   public static void main(String args[]){


Student s1 = new Student(20,"Alex");
Student s2 = new Student(21,"Bill",50);
s1.display();
s2.display();
  }
}
```

Output

```
20 Alex 0
21 Bill 50
```

iii) Inheritance

Inheritance refers to the process of one Java class acquiring the property of another Java class. The methods and fields of an existing class can be reused with inheritance. Inheritance enables reusability of classes in Java programming.

* **Inheritance types in Java**

There are various types of inheritance in Java:

1. **Single Inheritance:** One class can extend another class
2. **Multiple Inheritance:** One class extends to multiple classes. Not achievable in Java.

3. **Multilevel Inheritance:** One class can be inherited from a derived class. Derived class turns into the base class of new class.
4. **Hierarchical Inheritance:** Many subclasses inherit one class
5. **Hybrid Inheritance**: It is a mix between single and multiple inheritance

Inheritance is used in Java when two classes have an "Is-A" relationship. An inherited class is called the sub class.

Java Inheritance Syntax:

```
class subClass extends superClass

{
  //methods and fields
}
```

Java Inheritance Example:

```
class Employee{
 float salary=20000;
}

class Developer extends Employee{
int bonus=5000;
public static void main(String args[]){
 Developer d=new Developer();
 System.out.println('Salary of developer is:'+d.salary);
 System.out.println('Bonus of developer is:'+d.bonus);
}
}
```

Output:

```
Salary of developer is: 20000.0
Bonus of developer is: 5000.0
```

* **Inheritance types in Java**

Understand inheritance with the example of a person –

Here person is the class and name is a property of the person. Person as a class can be used for defining other classes.

```
public class Person
{
 private String name;


 public Person()
 {
  name = 'No name yet.';
 }

 public Person(String initialName)
 {
  name = initialName;
 }

 public void setName(String newName)
 {
  name = newName;
 }


 public String getName()
 {
  return name;
 }


 public void writeOutput()
 {
  System.out.println('Name: ' + name);
 }
```

```
 public boolean sameName(Person otherPerson)
 {
   return (this.name.equalsIgnoreCase(otherPerson.name));
 }


}
```

* **Function Overloading –**

Function overloading or method overloading refers to the action of a developer creating multiple methods having identical names but differing parameters. Method overloading occurs when a class contains two or more methods of the same name. Overloading functions/methods increases the readability of Java programs.

Two different function overloading processes in Java –

1. **Modifying the number of arguments –** One process of function overloading involves changing the total number of arguments.

Code example –

```
class Adding{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class OverloadTest{
public static void main(String[] args){  System.out.println(Adding.add(6,6)); System.out.println(Adding.add(6,6,6));
}}
```

Output –

```
12
18
```

2. **Modifying the data type of arguments –** Another process of function overloading involves changing the data type of

arguments.

```
class Adding{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){  System.out.println(Adding.add(7,7)); System.out.println(Adder.add(9.1,9.2));
}}
```

Output –

```
14
18.3
```

Changing argument numbers and data type of arguments are valid ways of function/method overloading in Java. But, it has to be noted that changing the return type of a method cannot be used for overloading.

Do constructors have return statement?

Constructors do not have a return statement, and they do not return any values. A constructor lacks this statement because it is not directly called upon by code.

There is simply no need for a constructor to have a return statement, as its main purpose is to assign values for variables.

Garbage collector – In Java, garbage collector refers to a programme that performs automatic de-allocation of objects to free up memory. Many objects are created during a Java programming process. While some objects are actively referenced by a programmer, others that are not being used still fill up the memory. Abandoned objects are then swept up by the garbage collector, and the memory space is reclaimed. The garbage programme runs on the JVM.

Destructor – Destructor refers to an automatically-called method that functions at the end of an object's lifecycle. It can be considered as the opposite of a constructor as it releases heap memory and deletes objects. It closes database connections and frees up network resources. A destructor being present in a programme prompts the garbage

collector to invoke it during the collection process. The 'finalize' method is used in Java to call this method.

Code example of internally calling the finalize method to invoke destructor–

```
public class Example
{
public static void main(String[] args)
{
Example ex = new Example();
ex = null;
System.gc();
System.out.println('Within the Method');
}
protected void finalize()
{
System.out.println('garbage collection occurs');
}
}
```

Output:

```
Within the Method
garbage collection occurs
```

Keywords that cannot be logically used with constructor –

* **Final –** Using the Final keyword signifies that a programmer does not want any class to override the method. This is not valid in case of a constructor, as constructors cannot be overridden by default.

* **Abstract –** Using the Abstract method signifies that the method does not have a body and will be used in another class in the future. But, constructors by default create bodies i.e a constructor has a body. Since a constructor cannot be overridden, it cannot have the implementation of an abstract class.

* **Static –** Setting a Static method signifies that it belongs to a class and not any object in particular. Objects are not created for a Static method, but the purpose of a constructor is to create an object. So, there is no logic in using the Static keyword with a constructor.

* **Native –** The Native keyword is a method modifier employed to enable the use of non-Java code. The JVM needs to use a different compiler for compiling native code. Hence, adding a native constructor means having to supplement the JVM with another library. So, attaching a Native keyword to a constructor makes no sense.

* **Synchronized –** The Synchronized keyword is used to lock an object in multi-thread scenarios where object access by multiple threads must be avoided. Java constructors cannot be synchronized as an object would be locked and further creation would be halted.

Practical (60 minutes)

See the example programme for Java constructor overloading below. Write the same programme to showcase constructor overloading for two students named Ram and Ratan, who have student IDs 16 and 17, and ages 42 and 27, respectively. Show the resulting output.

```
class Student{
 int id;
  String name;
```

```
int age;

  Student(int i,String n){
 id = i;
name = n;
 }
 Student(int i,String n,int a){
id = i;
name = n;
age=a;
   }
void display(){System.out.println(id+" "+name+" "+age);}
   public static void main(String args[]){
Student s1 = new Student(20,"Alex",30);
Student s2 = new Student(21,"Bill",50);
s1.display();
s2.display();
  }
}
```

Instructions: The progress of students will be assessed with the exercises mentioned below.

MCQ

1. Constructor refers to a code _____ in Java

a) extension

b) block

c) box

d) None of the mentioned

2. Which of these is section of a constructor?

a) Access modifier

b) Value modifier

c) Switcher

d) None of the mentioned

3. When is a default constructor assigned to a Java class?

a) programmer has written constructor

b) programmer has written no constructor

c) programmer has written all constructors

d) None of the mentioned

4. Booleans are initialized as _____ by default constructors.

a) true

b) semi-true

c) false

d) All of the mentioned

5. Which type of constructor does not take any arguments?

a) primary

b) nullary

c) preliminary

d) None of the mentioned

6. A parameterized constructor has _____.

a) Arguments

b) Involvements

c) Connection statements

d) None of the mentioned

7. Overloading lets programmers use default constructors and those with _____ simultaneously.

a) box patterns

b) consoles

c) parameters

d) None of the mentioned

8. Inheritance involves one Java class inheriting the _____ of a different Java class.

a) length

b) property

c) type

d) None of the mentioned

9. In single inheritance one class can extend to _____ _____.

a) another class

b) multiple classes

c) infinite classes

d) None of the mentioned

10. Hybrid Inheritance is a combination of single and _____ inheritance.

a) Double

b) triple

c) multiple

d) None of the mentioned