WEEK3

Constructors: rules for defining constructor; types; Destructor; Access modifiers; this keyword; Autoboxing and unboxing; Operators; Expressions; Evaluation of expressions;

Constructors

- A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
- A constructor is a block of codes similar to the method. It is called when an instance of the class is created.
- At the time of calling the constructor, memory for the object is allocated in the memory.
- Every time an object is created using the new() keyword, at least one constructor is called.

Rules for creating Java constructor

Following are the rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.
Syntax of default constructor:

```
<class_name>()
{
}
```

Example:

```
//Java Program to create and call a default constructor
class Bike
{
        //creating a default constructor
        Bike()
        {
                System.out.println("Bike is created");
        }
}

class Bike_Main
{
        //main method
        public static void main(String args[])
        {
                //calling a default constructor
                Bike b=new Bike();
        }
}
```

Output:

```
Bike is created
```

Java Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student
{
        int id;
        String name;
        //creating a parameterized constructor
        Student(int i, String n)
        {
                id = i;
                name = n;
        }
        //method to display the values
        void display()
        {
                System.out.println(id+" "+name);
        }
        public static void main(String args[])
        {
                //creating objects and passing values
                Student s1 = new Student(111,"Karan");
                Student s2 = new Student(222,"Aryan");
                //calling method to display the values of object
                s1.display();
                s2.display();
        }
}
```

Output:

```
        111 Karan
        222 Aryan
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Destructor:

It is a special method that automatically gets called when an object is no longer used. When an object completes its life-cycle the garbage collector deletes that object and deallocates or releases the memory occupied by the object.

Access Modifiers in Java

The access modifiers in Java specify the accessibility or scope of a attribute, method, constructor, or class. We can change the access level of attributes, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	**within class**	**within package**	**outside package by subclass (child) only**	**outside package**
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Private

The private access modifier is accessible only within the class.

Example:

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}

public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);//Compile Time Error
                obj.msg();//Compile Time Error
        }
}
```

Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package.

Example:

In this example, we have created two packages **pack** and **mypack**. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
	void msg()
	{
		System.out.println("Hello");
	}
}
//save by B.java
package mypack;
import pack.*;
class B
{
	public static void main(String args[])
	{
		A obj = new A();//Compile Time Error
		obj.msg();//Compile Time Error
	}
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Protected

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.

Example:

In this example, we have created the two packages **pack** and **mypack**. The class A is public, so it can be accessed from outside the package. But msg() method is protected, so it can be accessed from outside the class only through inheritance (child class).

```
//save by A.java
package pack;
public class A
{
        protected void msg()
        {
                System.out.println("Hello");
        }
}
//save by B.java
package mypack;
import pack.*;
class B extends A
{
        public static void main(String args[])
        {
                A obj = new A();
                obj.msg();
        }
}
```

Output: Hello

Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example:

```
//save by A.java
package pack;
public class A
{
        public void msg()
        {
                System.out.println("Hello");
        }
}
//save by B.java
package mypack;
import pack.*;
class B
{
        public static void main(String args[])
        {
                A obj = new A();
                obj.msg();
        }
}
```

Output: Hello

‘this’ keyword:

In Java, ‘this’ is a **reference variable** that refers to the current object.

Example program 1: Understanding the problem without ‘this’ keyword.

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno, String name, float fee)
        {
                rollno=rollno;
                name=name;
                fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}
class ThisMain
{
        public static void main(String args[])
        {
                  Student s1 = new Student(111,"ankit",5000);
                  Student s2 = new Student(112,"sumit",6000);
                  s1.display();
                  s2.display();
        }
}
```

Output:

```
0, null, 0.0
0, null, 0.0
```

- In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Example program 2: Solution to the above program

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno, String name, float fee)
        {
                this.rollno = rollno;
                this.name = name;
                this.fee = fee;
        }
        void display()
        {
                System.out.println(rollno +" "+ name +" "+ fee);
        }
}

class ThisMain
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit",5000);
                Student s2=new Student(112,"sumit",6000);
                s1.display();
                s2.display();
        }
}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

Example program 3: this() : to invoke current class constructor

```
class Student
{
        int rollno;
        String name, course;
        float fee;
        Student(int rollno, String name, String course)
        {
                this.rollno=rollno;
                this.name=name;
                this.course=course;
        }
        Student(int rollno, String name, String course, float fee)
        {
                this(rollno, name, course);
                this.fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+course+" "+fee);
        }
}
class ThisMain
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit","java");
                Student s2=new Student(112,"sumit","java",6000f);
                s1.display();
                s2.display();
        }
}
```

Output:

```
111 ankit java 0.0
112 sumit java 6000.0
```

Wrapper classes in Java

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.
- The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer

long	Long
float	Float
double	Double

Autoboxing

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing.
- For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- Example program

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class AutoboxingDemo
{
        public static void main(String args[])
        {
                //Converting int into Integer
                int a=20;
                Integer i=Integer.valueOf(a); //converting int into Integer explicitly
                Integer j=a; //autoboxing
                System.out.println(a+" "+i+" "+j);
        }
}
```

- Output: **20 20 20**

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
- It is the reverse process of autoboxing.
- Example program

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
```

```
public class UnboxingDemo
{
	public static void main(String args[])
	{
		//Converting Integer to int
		Integer a=new Integer(3);
		int i=a.intValue(); //converting Integer to int explicitly
		int j=a; //unboxing
		System.out.println(a+" "+i+" "+j);
	}
}
```

- Output: **3 3 3**

Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
public class OperatorUnary
{
        public static void main(String args[])
        {
                int x=10;
                System.out.println (x++);
                System.out.println (++x);
                System.out.println (x--);
                System.out.println (--x);
        }
}
```

Output:
```
10
12
12
10
```

Java Unary Operator Example: ~ (Bitwise complement) and ! (Logical complement)

- The bitwise complement works on the binary representation of numbers.
- It is a unary operator, meaning it operates on a single operand.
- For an integer n, the result of ~n is equivalent to -(n + 1) due to the way numbers are stored in two's complement representation in Java.

```
public class OperatorUnary
{
        public static void main(String args[])
        {
                int a=10;
                int b=-10;
                boolean c=true;
                boolean d=false;
                System.out.println(~a);
                System.out.println(~b);
                System.out.println(!c);
                System.out.println(!d);
        }
}
```
Output:
```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example
```
public class OperatorArithmatic
{
        public static void main(String args[])
        {
                int a=10;
                int b=5;
                System.out.println(a+b);
                System.out.println(a-b);
                System.out.println(a*b);
                System.out.println(a/b);
                System.out.println(a%b);
        }
}
```

Output:

```
        15
        5
        50
        2
        0
```

Java Left Shift Operator

The **left shift operator** in Java is represented by <<. It shifts the bits of a number to the left by the specified number of positions, filling the rightmost bits with 0. This effectively multiplies the number by 2^n, where n is the number of positions shifted.

Java Left Shift Operator Example

```
public class LeftShiftExample
{
        public static void main(String[] args)
        {
                int num = 5; // Binary: 00000000 00000000 00000000 00000101
                int result = num << 2; // Shift left by 2 positions
                System.out.println("Original number: " + num);
                System.out.println("After left shift: " + result);
        }
}
```

Output:

```
        Original number: 5
        After left shift: 20
```

Java Right Shift Operator

The **right shift operator** in Java is represented by >>. It shifts the bits of a number to the right by the specified number of positions. It divides the number by 2^n, where n is the number of positions shifted.

Java Right Shift Operator Example

```
public class RightShiftExample
{
        public static void main(String[] args)
        {
                int num = 16; // Binary: 00000000 00000000 00000000 00010000
                int result = num >> 2; // Shift right by 2 positions
                System.out.println("Original number: " + num);
                System.out.println("After right shift: " + result);
   }
}
```

Output:

```
Original number: 16
After right shift: 4
```

Logical && and Bitwise &

- The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.
- The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorAND
{
        public static void main(String args[])
        {
                int a=10;
                int b=5;
                int c=20;
                if(a<b   &&   a++<c)
                {
                         System.out.println("Both the conditions are checked and it is
                                                  true");
                }
                else
                {
                        System.out.println("Second condition has not checked because
                                the first condition was false");
                        System.out.println("Proof a: " + a);
                }
                if(a<b  &  a++<c)
                {
                        System.out.println("Both the conditions are checked and it is
                                true");
                }
                else
                {
                        System.out.println("Both the conditions are checked though the
                                first condition was false");
                        System.out.println("Proof a: " + a);
                }
        }
}
```

Output:

```
Second condition has not checked because the first condition was false
Proof a: 10
Both the conditions are checked though the first condition was false
Proof a: 11
```

Logical || and Bitwise |

- The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.
- The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorORdemo
{
    public static void main(String args[])
    {
                int a=10;
                int b=5;
                int c=20;
                if(a>b  ||   a++<c)
                {
                        System.out.println("If any/both of the condition are true");
                        System.out.println("Proof a: " + a);
                }
                else
                {
                        System.out.println("Both the conditions are false");
                }
                if(a>b  |   a++<c)
                {
                        System.out.println("If any/both of the condition are true");
                        System.out.println("Proof a: " + a);
                }
                else
                {
                          System.out.println("Both the conditions are false");
                }
    }
}
```

Output:

```
If any/both of the condition are true
Proof a: 10
```

```
If any/both of the condition are true
Proof a: 11
```

Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
public class OperatorTernaryDemo
{
        public static void main(String args[])
        {
                int a=2;
                int b=5;
                int min=(a<b)?a:b;
                System.out.println(min);
        }
}
Output:
        2
```

Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example1

```
public class OperatorAsgnDemo1
{
        public static void main(String args[])
        {
                int a=10;
                int b=20;
                a+=4; //a = a + 4
                b-=4; //b = b - 4
                System.out.println(a);
                System.out.println(b);
        }
}
Output:
        14
        16
```

Java Assignment Operator Example2

```
public class OperatorAsgnDemo2
{
        public static void main(String[] args)
        {
                int a=10;
                a+=3;
                System.out.println(a);
                a-=4;
                System.out.println(a);
                a*=2;
                System.out.println(a);
                a/=2;
                System.out.println(a);
        }
}
Output:
        13
        9
        18
        9
```

What is an expression?

In Java, an **expression** is a construct that combines variables, operators, method calls, and literals to produce a value. It represents a computation or action and always evaluates to a single value.

Components of an Expression:

- **Literals**: Fixed values like 5, "Hello", or true.
- **Variables**: Symbols representing data like x or total.
- **Operators**: Symbols used to perform operations like +, -, *, /, etc.
- **Method Calls**: Invoking methods, e.g., sum(20,40).

Types of Expressions in Java:

1. **Arithmetic Expressions**:
   - Perform mathematical computations.

     Example:
     int result = (5 + 3) * 2; // Expression: (5 + 3) * 2
2. **Relational Expressions**:
   - Compare two values and return a boolean (true or false).

     Example:
     boolean isGreater = 10 > 5; // Expression: 10 > 5
3. **Logical Expressions**:
   - Combine boolean values using logical operators like &&, ||, and !.

     Example:
     boolean result = (10 > 5) && (4 < 8); // Expression: (10 > 5) && (4 < 8)
4. **Assignment Expressions**:
   - Assign a value to a variable.

     Example:
     int x = 10; // Expression: x = 10
5. **Method Invocation Expressions**:
   - Call a method and use its result.

     Example:
     int s = sum (20, 40); // Expression: sum (20, 40)
6. **Compound Expressions**:
   - Combine multiple operations.

     Example:
     int result = (5 + 3) * (2 - 1); // Compound expression

What is operator precedence?

- The **operator precedence** represents how two expressions are bind together.
- In an expression, it determines the grouping of operators with operands and decides how an expression will evaluate.
- While solving an expression two things must be kept in mind the first is precedence and the second is **associativity**.

Precedence

- Precedence is the priority for grouping different types of operators with their operands.
- The operators having higher precedence are evaluated first. If we want to evaluate lower precedence operators first, we must group operands by using parentheses and then evaluate.

Associativity

- We must follow associativity if an expression has more than two operators of the same precedence.
- In such a case, an expression can be solved either **left-to-right** or **right-to-left,** accordingly.

Order of Evaluation

1. **Parentheses**: Evaluate sub-expressions in parentheses first.
2. **Unary Operators**: E.g., ++, --, +, - are evaluated next.
3. **Multiplication/Division/Modulo**: Evaluated before addition and subtraction.
4. **Relational Operators**: E.g., <, >, <=, >= are evaluated after arithmetic operators.
5. **Logical Operators**: E.g., &&, || are evaluated after relational operators.
6. **Assignment**: Evaluated last.

Let's understand how the following expression is evaluated step by step.

int x = 2;

result = x++ + x++ * --x / x++ - --x + 3 >> 1 | 2;

1. operands are evaluated first, left to right

   2 + 3 * 3 / 3 - 3 + 3 >> 1 | 2

2. * and / have higher preference than other, * evaluated first as it comes in left

   2 + 9 / 3 - 3 + 3 >> 1 | 2

3. / have higher preference than others

   2 + 3 - 3 + 3 >> 1 | 2

4. + and - have higher preference than others

   5 >> 1 | 2

5. >> have higher preference than |

   2 | 2

Output

2

Program 01:
Write a program to demonstrate the use of default constructor.

```
        //Java Program to create and call a default constructor
        class Bike
        {
                //creating a default constructor
                Bike()
                {
                        System.out.println("Bike is created");
                }
                //main method
                public static void main(String args[])
                {
                        //calling a default constructor
                        Bike b=new Bike();
                }
}
```

Output:

Program 02:

Write a program to demonstrate the use of parameterized constructor.

```
class Student
{
        int id;
        String name;
        //creating a parameterized constructor
        Student(int i, String n)
        {
                id = i;
                name = n;
        }
        //method to display the values
        void display()
        {
                System.out.println(id+" "+name);
        }
        public static void main(String args[])
        {
                //creating objects and passing values
                Student s1 = new Student(111,"Karan");
                Student s2 = new Student(222,"Aryan");
                //calling method to display the values of object
                s1.display();
                s2.display();
        }
}
```

Output:

Program 03:

Find the error in the program

```
public class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}

public class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data);
                obj.msg();
        }
}
```

Error:

Program 04:

Write a program to demonstrate the use of 'this' keyword.

(i) Here we are using **'this'** keyword to distinguish local variable and instance variable

```
class Student
{
        int rollno;
        String name;
        float fee;
        Student(int rollno, String name, float fee)
        {
                this.rollno=rollno;
                this.name=name;
                this.course=course;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+fee);
        }
}

class ThisMain
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
        }
}
```

Output:

(ii) **this()** : to invoke current class constructor

```
class Student
{
        int rollno;
        String name, course;
        float fee;
        Student(int rollno, String name, String course)
        {
                this.rollno=rollno;
                this.name=name;
                this.course=course;
        }
        Student(int rollno, String name, String course, float fee)
        {
                this(rollno, name, course);
                this.fee=fee;
        }
        void display()
        {
                System.out.println(rollno+" "+name+" "+course+" "+fee);
        }
}
class ThisMain
{
        public static void main(String args[])
        {
                Student s1=new Student(111,"ankit","java");
                Student s2=new Student(112,"sumit","java",6000f);
                s1.display();
                s2.display();
        }
}
```

Output:

Program 05:
Write a Java program to convert primitive data type into objects (Autoboxing)

```
public class AutoboxingDemo
{
        public static void main(String args[])
        {
                //Converting int into Integer
                int a=20;
                Integer j=a; //autoboxing
                System.out.println(a+"   "+"   "+j);
        }
}
```

Output:

Program 06:

Write a Java program to convert object into primitive data type (Unboxing)

```
public class UnboxingDemo
{
        public static void main(String args[])
        {
                //Converting Integer to int
                Integer a=new Integer(3);
                int j=a; //unboxing
                System.out.println(a+"   "+"   "+j);
        }
}
```

Output:

Program 07:

Write the output of the following programs

(i)

```
class op
{
        public static void main(String args[])
        {
                int a = 10, int i = 3, int j = 15;
                System.out.println(-3-a+j);
        }
}
```

Output:

(ii)

```
class Precedence
{
        public static void main(String[] args)
        {
                int a = 10, b = 5, c = 1, result;
                result = a-++c-++b;
                System.out.println(result);
        }
}
```

Output:

(iii)

```
class OperatorPrecedence
{
  public static void main (String[] args)
{
     int result = 0;
     result = 5 + 2 * 3 - 1;
     System.out.println("5 + 2 * 3 - 1 = " +result);

     result = 5 + 4 / 2 + 6;
     System.out.println("5 + 4 / 2 + 6 = " +result);

     result = 3 + 6 / 2 * 3 - 1 + 2;
     System.out.println("3 + 6 / 2 * 3 - 1 + 2 = " +result);

     result = 6 / 2 * 3 * 2 / 3;
     System.out.println("6 / 2 * 3 * 2 / 3 = " +result);

     int x = 2;
     result = x++ + x++ * --x / x++ - --x + 3 >> 1 | 2;
     System.out.println("result = " +result);
    }
 }
```

Output: