Day 2- Facilitation Guide

Introduction to OOPs concepts & Class members

Index

For (1.5 hrs) ILT

In this session we will discuss OOP concepts and will teach you how to define a class and add class members. Java Class consists of Class Declaration, Class Members, Constructors and Methods. It also contains optional package declaration and import statements.We will also discuss Java operators. Java operators are symbols or special characters used to perform operations on variables, constants, and expressions.

Food for thought…

Trainer should ask the students if they remember what is OOPS? How did we move from procedural programming to object-oriented programming?

I. OOPs Concept

What is OOP?

OOP stands for Object-Oriented Programming, which is a programming paradigm that organizes and structures code around the concept of "objects."

Object-oriented programming is a core of Java Programming, which is used for designing a program using classes and objects. OOP can also be characterized as data controlling for accessing the code. In this approach, programmers define the data type of a data structure and the operations that are applied to the data structure.

Question: Do you know any other OOP language?

OOP in Java helps to improve code readability and reusability by defining a Java program efficiently. The main principles of object-oriented programming are abstraction, encapsulation, inheritance, and polymorphism. These concepts aim to implement real-world entities in programs.

Object-Oriented Programming (OOP) is a programming paradigm that brings a set of principles and concepts to organize and design software systems. It is not mandatory, but it is widely adopted and has become an essential approach in modern software development.

List of OOP concepts in Java

- Objects
- Classes
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Video: ***Get ready to immerse yourself in the concept as we play the video and embark on a journey of learning***

Java concept-OOP

II. What are Objects?

Any real-world entity that has some characteristic and behavior are considered as objects.

In Java, objects are instances of a class which are created from a class in Java. They have states and behavior.

These objects always correspond to things found in the real world, i.e., real entities. So, they are also called run-time entities of the world. These are self–contained which consists of methods and properties which make data useful. Objects can be both physical and logical data. It contains addresses and takes up some space in memory. Some examples of objects are a dog, chair, tree, etc.

When we treat an animal as an object, it has states like color, name, breed etc., and behavior such as eating, wagging the tail, making a sound, etc.

Let's take a look at a real time example of Object:

A smartphone can be modeled as an object. It has attributes such as the brand, model, and storage capacity. The smartphone object's behavior includes making calls, sending messages, taking photos, and installing apps.

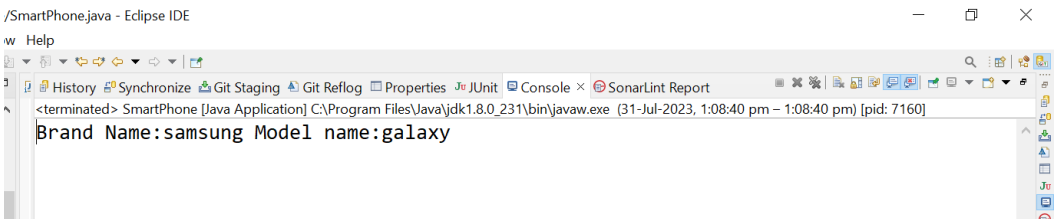
Let's create a class called SmartPhone, we specify the class name followed by the object name, and we use the **new** keyword.

Example:

```
public class SmartPhone{

String brandName=”samsung”;

String modelName=”Galaxy”;

public static void main (String[] args) {

SmartPhone smartPhone= new SmartPhone(); //we are creating object of SmartPhone Class

System.out.println("Brand Name:"+smartPhone.brandName+" Model name:"+smartPhone.modelName);


}

}
```

In the above example,We have declared a SmartPhone with some attribute brandName and modelName and assign some values into it.

Output:



III. What are Classes?

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even a simple class can cleanly model state and behavior.

Classes have members which can be fields, methods and constructors. A class has both **static** and **instance** initializers.

> **Food for thought:**
>
> Why do you think static keyword was introduced in Java?

A class declaration consists of:

1. Modifiers: These can be public or default access.
2. Class name: Initial letter.
3. Superclass: A class can only extend (subclass) one parent.
4. Interfaces: A class can implement more than one interface.
5. Body: Body surrounded by braces, { }.

A class keyword is used to create a class. A simplified general form of the class definition is given below:

Syntax to declare a class:

```
class <class_name>{
   field(s);
   method(s);
   Constructor(s);
}
```

In the above example,we have seen one example of Object.In that example,we have created a class named SmartPhone.A Smartphone class can define the common features of smartphones. It might have attributes like brand, model, and storage_capacity, as well as methods for making_calls(), sendingMessages(), takingPhotos(), and installingApps().

Up until now, we've come across terms like fields and methods. Let's delve deeper into these concepts and explore their intricacies.

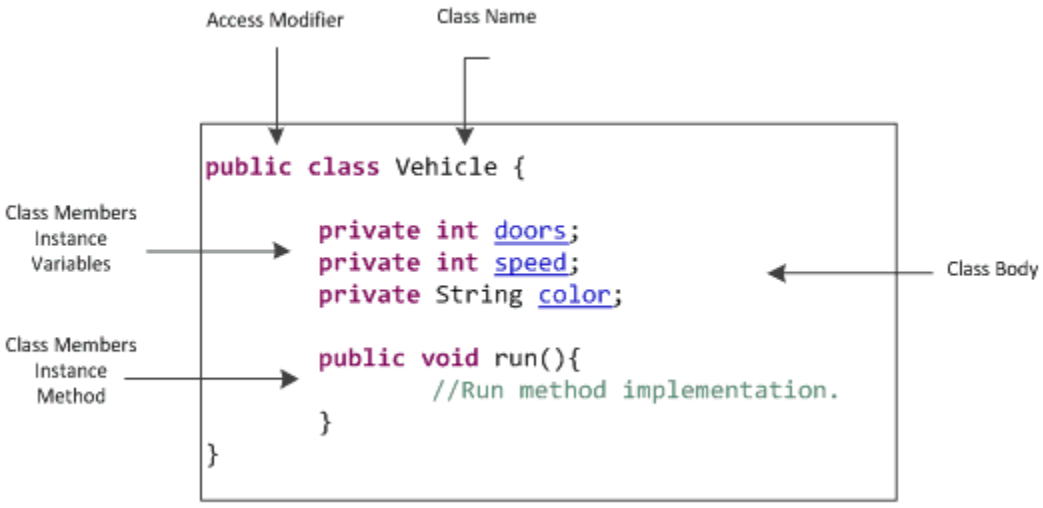
Class Members

In a Java program, class members refer to the fields (variables), methods, constructors, nested classes or interfaces, and static blocks.

Each member must be declared with an appropriate access modifier and should follow the syntax rules for the particular type of member.

Class members are used to define the structure and functionality of the class.

Here are the three main types of class members in Java:



Fields (Variables):

Fields, also known as variables or class variables, represent the data associated with a class. They store the state or characteristics of objects created from the class. Fields can be of various data types such as integers, strings, booleans, and custom types. They can have different access modifiers (e.g., public, private, protected, or default) to control their visibility and accessibility.

```
//Java Program to illustrate how to define a class and fields
//Defining an Employee class.

class Employee{

 //defining fields
 int id;  //field or data member or instance variable
```

```
 String name;
}
```

Here, Employee is a class that has characteristics such as id, name.Id and name are instance variables which will be created when an object is instantiated, and are accessible to all the constructors, methods, or blocks in the class. Access modifiers can be given to the instance variable.

Overall, variables are fundamental building blocks in programming, allowing developers to work with and manage data efficiently, make decisions, and create dynamic and flexible applications. Without variables, programming would be severely limited in its ability to store and manipulate data, and the complexity of software development would increase significantly.

Methods:
Methods define the behavior or actions that objects of a class can perform. They encapsulate the algorithms and operations associated with the class. Methods can have parameters (inputs) and a return type (output), which determines if and what value is returned from the method. Like fields, methods can have different access modifiers.

Example:

```
class Employee{

 //defining fields
 int id; //field or data member or instance variable
 String name;
//declaring method to initialize the data members
void insertRecord(int i, String n){

  id=r;

  name=n;

 }

}
```

Here, Employee is a class whose objects will have a behavior/action of insertRecord. This is indicated by the method named insertRecord().

Constructors:

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.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such cases, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

There are two 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 a "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

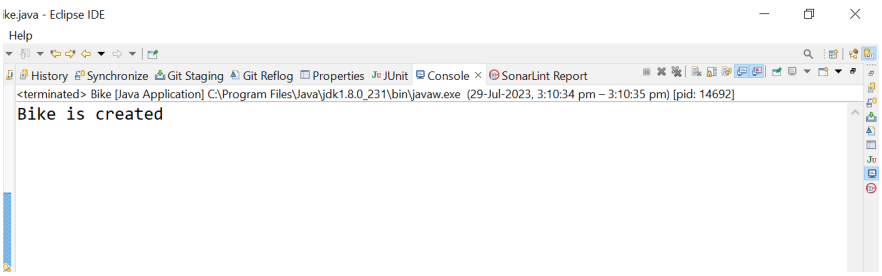
1. <class_name>(){}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{

//creating a default constructor
Bike1(){System.out.println("Bike is created");}

//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

When we run the program,the Output will be:



Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

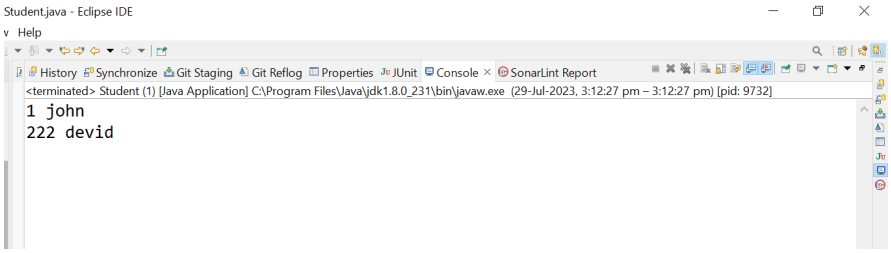
The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of the Student class that has two parameters. We can have any number of parameters in the constructor.

```
//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(1,"john");
    Student s2 = new Student(222,"devid");

    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

When we run the program,the Output will be:



In the example above, we have a Student class with a parameterized constructor that takes id and name as arguments. These constructors can be used to create Student objects with different initial values for the id and name fields.

Knowledge check…

Now that we've covered Class, Objects, and data members, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1.Which of the following statements is true about a Java class?

a) A class is an object.

b) A class is a blueprint for creating objects.

c) A class is a method in Java.

d) A class is a data type.

Ans-b

2.Which keyword is used to create an object of a class in Java?

a) new

b) create

c) instantiate

d) build

Ans-a

3.What is the purpose of a constructor in Java?

a) To define class fields.

b) To perform mathematical calculations.

c) To initialize an object's state or properties.

d) To create a new instance of a class.

Ans-c

4.What is the default access modifier for class members (fields and methods) in Java if no access modifier is specified?

a) protected

b) private

c) public

d) default or package-private

Ans-d

At the end of the quiz,the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.

Note: Before moving to the next concept,the trainer can increase curiosity in students for upcoming concepts so that it can stimulate interest and encourage exploration.

Here are some **Tips**:

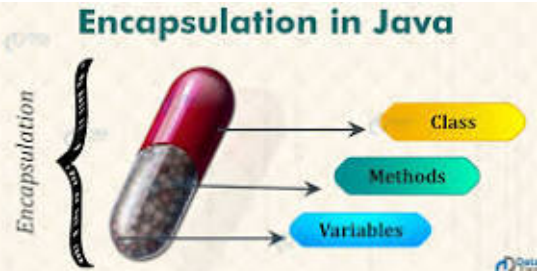
- Encourage group discussion on the concept.
- Relate the new concept to existing knowledge or experiences.
- Play the **video** and start group discussion

Java Concept-Encapsulation

IV. Encapsulation

Encapsulation describes the ability of an object to hide its data and methods from the rest of the world and is one of the fundamental principles of object-oriented programming.

Imagine a capsule that encloses the contents within it.

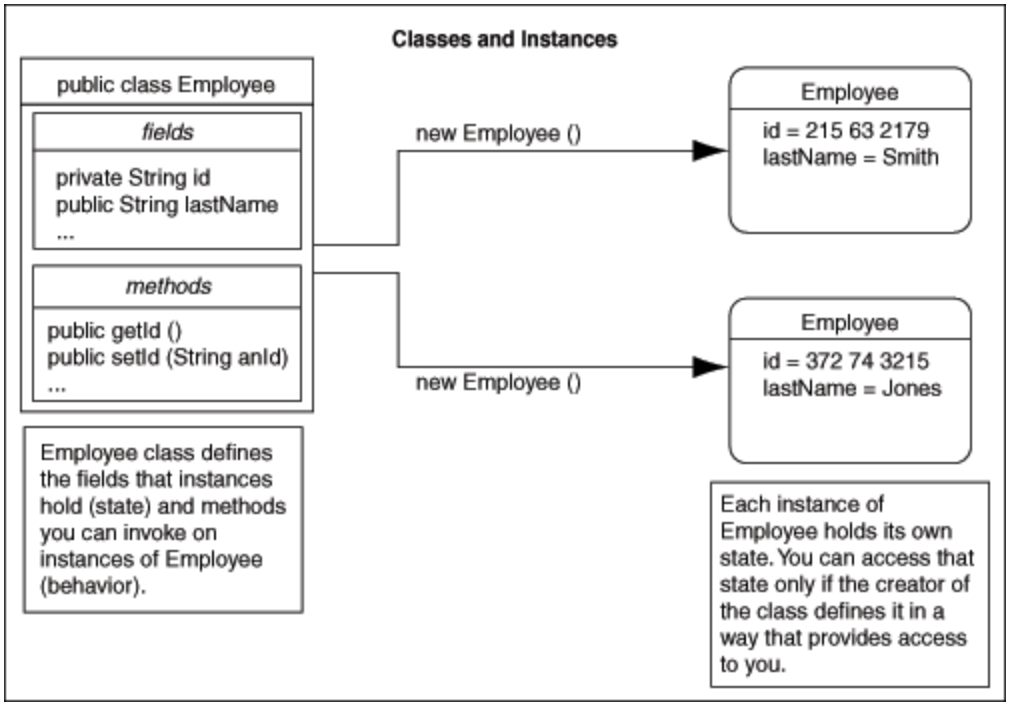


Similarly, we use classes in Java to encapsulate the data members (variables) and methods.

In Java, a class encapsulates the fields, which hold the state of an object, and the methods, which define the actions of the object.

***Note**: Encapsulation enables you to write reusable programs. It also enables you to restrict access only to those features of an object that are declared public. All other fields and methods that are private can be used for internal object processing.*

```
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
package com.example;
public class Employee {
private String id;
private String name;
        public String getId() {
        return id;
}
public void setId(String id) {
        this.id = id;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}
}
```



In the example illustrated in above Figure, the id field is private, and access to it is restricted to the object that defines it. Other objects can access this field using the getId() method. Using encapsulation, you can deny access to the id field either by declaring the getId() method as private or by not defining the getId() method.

Let us understand the importance of Encapsulation:

Data Hiding: Encapsulation allows you to hide the internal implementation details of a class from the outside world. By using access modifiers (e.g., private, protected), you can control the visibility of class members (fields and methods). This prevents direct access to sensitive data, reducing the risk of unauthorized modifications and ensuring that the data remains consistent.

Data Protection: By encapsulating data within a class, you can enforce constraints and validations on the data. This prevents the data from being set to invalid or inconsistent states. By providing public methods (getters and setters) to access and modify the data, you can maintain control over how the data is accessed and modified.

Security: Encapsulation enhances security by preventing unauthorized access to critical data. By encapsulating sensitive information, you reduce the risk of data manipulation and protect the integrity of the program.

Overall, encapsulation in Java is crucial for creating robust, maintainable, and secure software. It promotes best practices in designing classes, prevents data corruption, and allows for better control and understanding of your code.

Having learnt the importance of Encapsulation, let's now explore the consequences of not having Encapsulation in object-oriented programming (OOP).

Data Exposure: Without encapsulation, class members could be accessed and modified directly from any part of the program. This can lead to data being exposed to unintended or unauthorized changes, potentially causing data corruption and inconsistencies.

Lack of Data Validation: Encapsulation allows you to enforce constraints and validations on data within a class. Without it, there would be no centralized mechanism to ensure that data remains in a valid state, leading to potential bugs and errors due to inconsistent data.

Difficulty in Code Maintenance: In the absence of encapsulation, any changes to the internal representation of a class would impact the entire program that uses the class. This lack of isolation could make code maintenance challenging, as even minor changes may require modifications in many places.

Security Risks: Lack of encapsulation may lead to security vulnerabilities as sensitive data and critical methods would be exposed, making it easier for malicious users to manipulate the program in unintended ways.

In summary, encapsulation is a critical aspect of OOP that helps in data protection, code organization, code maintainability, and code reusability. Without encapsulation, software development would become more error-prone, less secure, and harder to maintain.

JavaBeans

JavaBeans are a convention for writing reusable software components in Java. They are Java classes that adhere to a specific naming convention and design pattern. The naming convention specifies that a JavaBean class must have a default constructor with no arguments and a set of getter and setter methods for each of its properties. The design pattern specifies that a JavaBean class must be serializable, have a consistent state, and provide a way to be notified of changes to its properties.

The properties of a JavaBean are private fields of the class, accessed through the getter and setter methods. The getter methods have names that start with "get" or "is" (for boolean properties), followed by the name of the property with the first letter capitalized. The setter methods have names that start with "set", followed by the name of the property with the first letter capitalized and an argument of the same type as the property.

Here's an example of a simple JavaBean class:

```
public class Person {

    private String name;

    private int age;


    public Person() {

        // default constructor

    }


    public String getName() {

        return name;

    }


    public void setName(String name) {

        this.name = name;
```

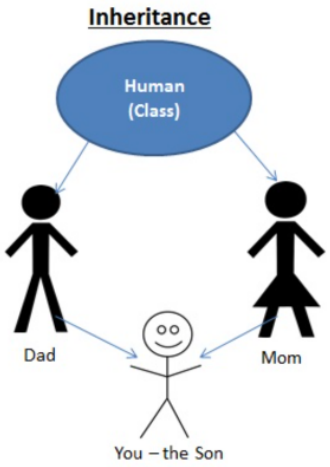
```
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Question:
What is the difference between a JavaBean and a Java class?

After explaining Encapsulation and Classes in detail, the trainer will introduce students to the other OOP concepts such as Inheritance, Polymorphism and Abstraction.

(PN: The scope of these topics is till introduction only. These will be covered in detail in succeeding sessions)

V. Inheritance



Consider a boy as shown above. He is a child of his parents (Dad and Mom). He has inherited certain characteristics and behaviors from them.
In short, they are related to each other.
Eg. He might look like his dad, his height might be like his mom, his nature might be like his dad.

But all of them are ultimately, humans with similar characteristics such as face, hands, legs, voice, etc.

> **Question:**
> Can you give examples such relationships between other real-world entities?

Similarly, we can create relationships between classes in Java.

Inheritance is the mechanism that allows one class to acquire all the properties from another class by inheriting the class. We call the inheriting class a child class and the inherited class as the superclass or parent class.

In Java, we do this by extending the parent class. Thus, the child class gets all the properties from the parent:

```
Unset


public class Car extends Vehicle {

    //...

}
```

When we extend a class, we form an IS-A relationship. The *Car* IS-A *Vehicle*. So, it has all the characteristics of a *Vehicle*.

We will learn about inheritance in detail in upcoming sessions.

VI. Abstraction

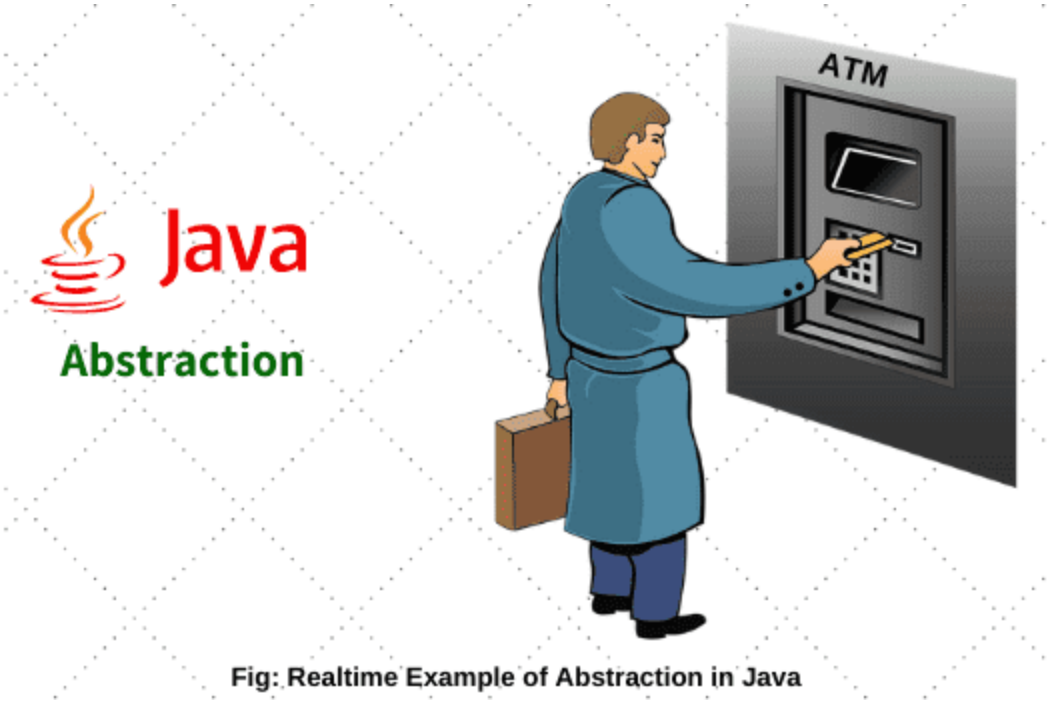


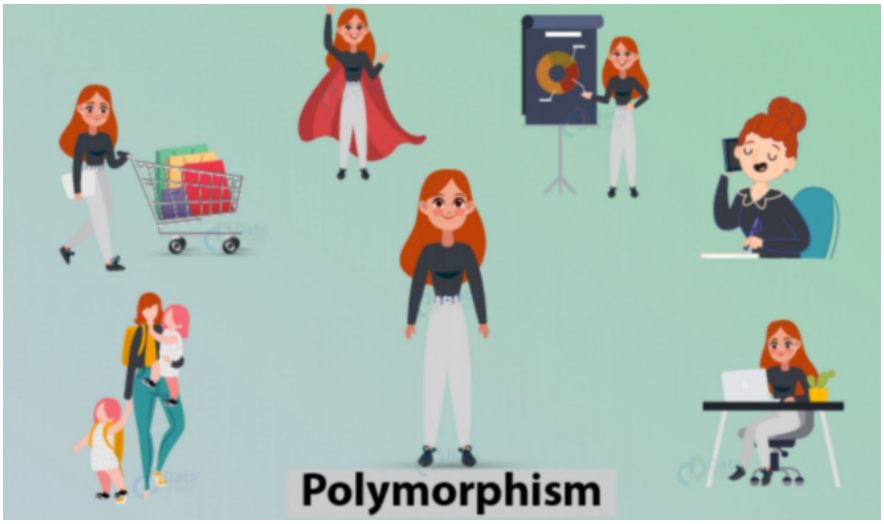
Fig: Realtime Example of Abstraction in Java

Consider an ATM. All the user has to do is insert the card, punch some buttons and the cash comes out. The user is not aware about the internal details or the process involved in account verification or counting the cash and dispensing it.

Abstraction in Java is a process of hiding the implementation details from the user and showing only the functionality to the user. It can be achieved by using abstract classes, methods, and interfaces. An abstract class is a class that cannot be instantiated on its own and is meant to be inherited by concrete classes.

We will learn about abstract classes in detail in upcoming sessions.

VII. Polymorphism



Consider a lady as shown above. She plays multiple roles of mother, sister, teacher, wife, etc and performs different functions based on that role. In short, the same person acts and behaves differently in different circumstances.

Polymorphism is the ability for different objects to respond differently to the same message. In object-oriented programming languages, you can define one or more methods with the same name. These methods can perform different actions and return different values.It will be discussed in upcoming sessions.

Knowledge check:

Now that we have learned the concept, let's put our knowledge to the test with a knowledge check!

Let's find the output of these given programs:

1.

```
package com.anudip.example;
class Circle {
private double radius;
public Circle(double radius) {
this.radius = radius;
}
public double getRadius() {
return radius;
}
```

```
public void setRadius(double radius) {
this.radius = radius;
}
}

public class Main {
public static void main(String[] args) {
        Circle circle = new Circle(5.0);
System.out.println("Radius of the circle: " + circle.getRadius());
circle.setRadius(7.0);
System.out.println("New radius of the circle: " + circle.getRadius());
}
}
```

What will be the output of the above Java program?

a) Radius of the circle: 5.0
New radius of the circle: 5.0

b) Radius of the circle: 5.0
New radius of the circle: 7.0

c) Radius of the circle: 7.0
New radius of the circle: 7.0

d) Radius of the circle: 7.0
New radius of the circle: 5.0

Ans-Radius of the circle: 5.0
New radius of the circle: 7.0

2.

```
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    public double getBalance() {
        return balance;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.0);
        System.out.println("Current balance: " + account.getBalance());

        account.balance = 1500.0;
        System.out.println("Updated balance: " + account.getBalance());
    }
}
```

What will be the output of the above Java program?

a) Current balance: 1000.0
Updated balance: 1500.0

b) Current balance: 1000.0
Updated balance: 1000.0

c) The program will not compile due to an error.

d) The program will throw a runtime exception.

Ans-The program will not compile due to an error.

VIII. Operators

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:

- Assignment Operator
- Arithmetic Operator
- Relational Operator
- Logical Operator
- Ternary Operator
- Unary Operator
- Bitwise Operator
- Shift Operator

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !

Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

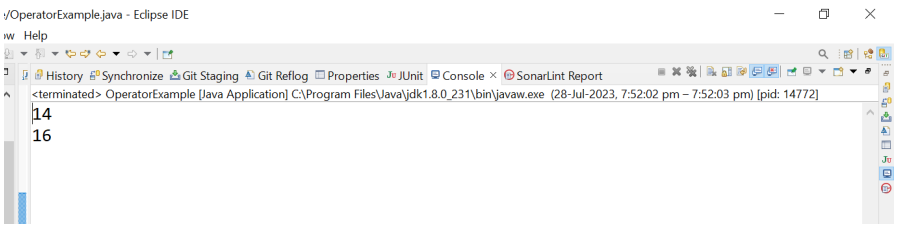
Assignment Operator

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.

Assignment Operator Example:

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

Output:



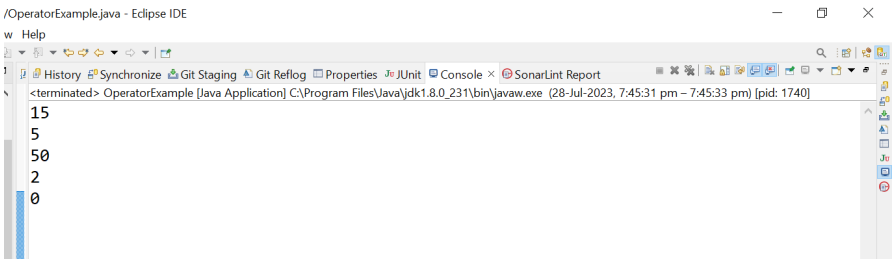
Arithmetic Operators

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

Arithmetic Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

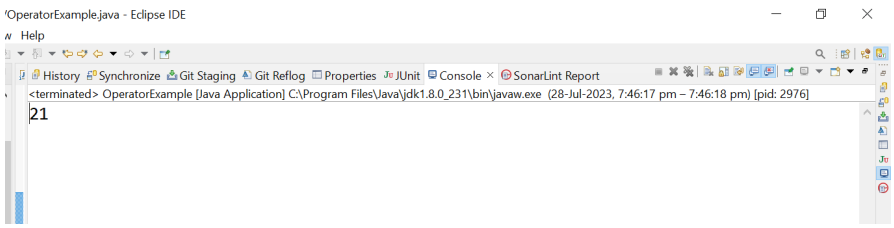
Output:



Arithmetic Operator Example: Expression

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:



Relational Operators

In Java, relational operators are used to compare values and determine the relationship between them. These operators return a boolean value, either true or false, based on the evaluation of the comparison. Java provides the following relational operators:

1.Equal to (==): Checks if two values are equal. For example:

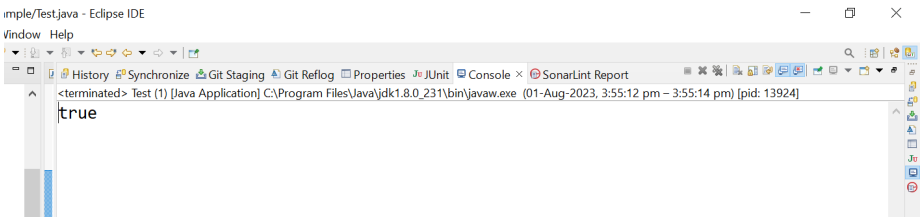
```
int x = 5;

int y = 10;

boolean result = x == y; // false
System.out.println(result);
```

2.Not equal to (!=): Checks if two values are not equal. For example:

```
int a = 7;

int b = 7;

boolean result = a != b; // false
System.out.println(result);
```

3.Greater than (>): Checks if the value on the left is greater than the value on the right. For example:

```
int num1 = 15;

int num2 = 10;

boolean result = num1 > num2; // true
System.out.println(result);
```



4.Less than (<): Checks if the value on the left is less than the value on the right. For example:

```
int num3 = 25;

int num4 = 30;

boolean result = num3 < num4; // true
System.out.println(result);
```

5.Greater than or equal to (>=): Checks if the value on the left is greater than or equal to the value on the right. For example:

```
int age = 18;
```

```
int requiredAge = 18;
boolean result = age >= requiredAge; // true
System.out.println(result);
```

6.Less than or equal to (<=): Checks if the value on the left is less than or equal to the value on the right. For example:

```
int marks = 85;
int passingMarks = 100;
boolean result = marks <= passingMarks; // false
System.out.println(result);
```

Relational operators are commonly used in conditional statements, such as "if" and "while" statements, to make decisions and control the flow of a Java program based on the results of the comparisons. They are essential for creating dynamic and responsive applications that can handle different input values effectively.

Logical Operators

In Java, logical operators are used to combine multiple conditions and perform logical operations on boolean expressions. Java provides three logical operators:

1.Logical AND (&&): The logical AND operator returns true if both operands are true, otherwise, it returns false. It performs short-circuit evaluation, which means if the left operand is false, the right operand is not evaluated.

```
boolean a = true;
boolean b = false;
boolean result = a && b; // false
System.out.println(result);
```

2.Logical OR (||): The logical OR operator returns true if at least one of the operands is true, otherwise, it returns false. Like the logical AND operator, it also performs short-circuit evaluation.

```
boolean x = true;
boolean y = false;
boolean result = x || y; // true
System.out.println(result);
```

3.Logical NOT (!): The logical NOT operator negates the value of a boolean expression. It returns true if the operand is false, and false if the operand is true.

```
boolean flag = true;
boolean result = !flag; // false
System.out.println(result);
```

Logical operators are commonly used in conditional statements, loops, and other control flow structures to create complex conditions and make decisions based on multiple criteria. They are also used for data validation, error handling, and various other scenarios where combining boolean expressions is necessary.

Ternary Operator

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

The meaning of **ternary** is composed of three parts. The **ternary operator (? :)** consists of three operands. It is used to evaluate Boolean expressions. The operator decides which value will be assigned to the variable. It can be used instead of the if-else statement. It makes the code much more easy, readable, and shorter.

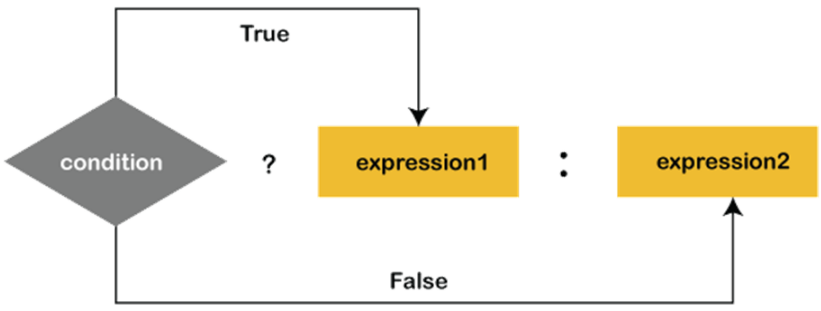
Note: Every code using an if-else statement cannot be replaced with a ternary operator.

Syntax:

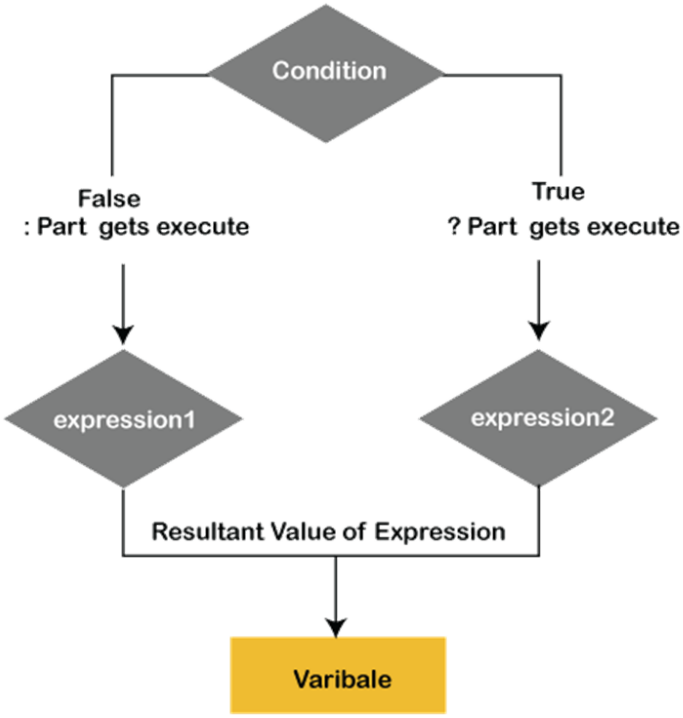
```
variable = (condition) ? expression1 : expression2
```

```
String result=(avg>=90)? “Grade A” :”fail”;
System.out.println(result);
```

The above statement states that if the condition returns **true, expression1** gets executed, else the **expression2** gets executed and the final result stored in a variable.



Let's understand the ternary operator through the flowchart.



Unary Operators

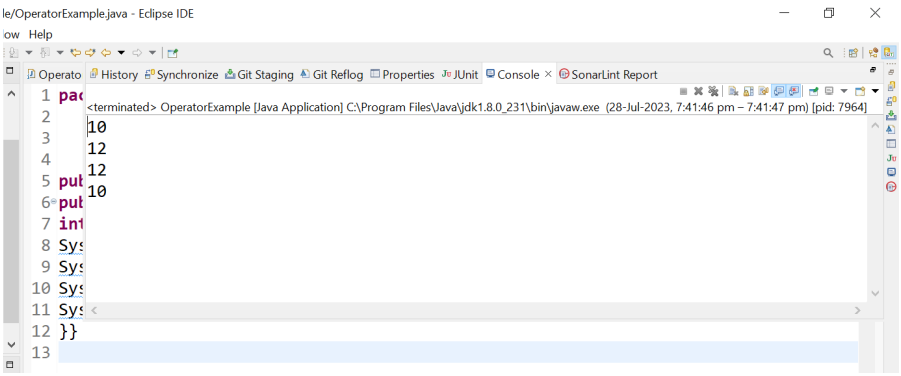
The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

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

Unary Operator Example 1: ++ and --

```
   public class OperatorExample{
public static void main(String args[]){
  int x=10;
System.out.println(x++);   //10 (11)  – first assignment then increment
  System.out.println(++x);   //12   – first increment then assignment
System.out.println(x--);   //12 (11)  – first assignment then decrement
 System.out.println(--x);  //10  –-first decrement then assignment
}}
```

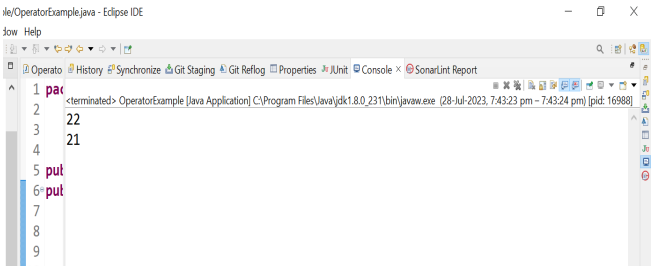
Output:



Unary Operator Example 2: ++

```
 public class OperatorExample{
 public static void main(String args[]){
  int a=10;
  int b=10;
System.out.println(a++ + ++a); //10+12=22
System.out.println(b++ + b++);  //10+11=21
}}
```

Output:



Unary Operator Example: ~ and !

'NOT' Operator(!)

This is used to convert true to false or vice versa. Basically, it reverses the logical state of an operand.

Syntax:

!(operand)

Illustration:

cond = !true;
// cond < false

Bitwise Complement(~)

This unary operator returns the one's complement representation of the input value or operand, i.e, with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

Syntax:

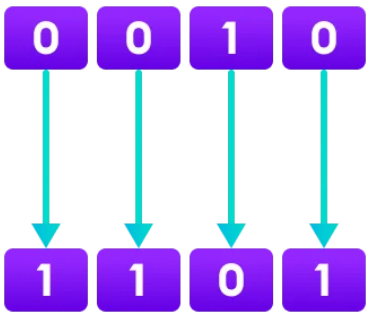
~(operand)

Illustration:

a = 5 [0101 in Binary]
result = ~5

This performs a bitwise complement of 5
~0101 = 1010 = 10 (in decimal)

Then the compiler will give 2's complement
of that number.
2's complement of 10 will be -6.
result = -6



Consider an integer 10. As per the rule, the bitwise complement of 10 should be -(10 + 1) = -11.

```
public class OperatorExample{
public static void main(String args[]){
int a= 10;
int b= -10;
boolean c=true;
boolean d=false;
System.out.println(a+"'s bitwise complement ="+~a);
System.out.println(b+"'s bitwise complement ="+~b);
System.out.println(!c);  //false (opposite of boolean value)
System.out.println(!d);  //true
}}
```

Output:

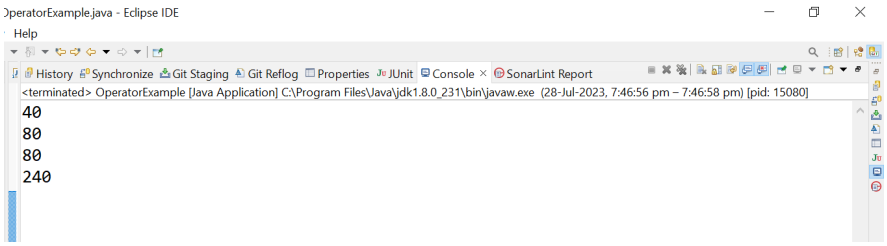
Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Left Shift Operator Example:

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:



Right Shift Operator

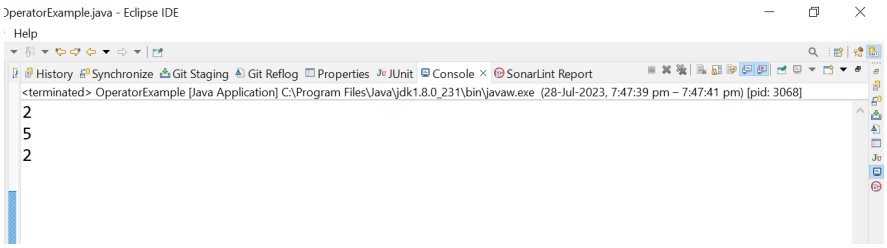
The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Right Shift Operator Example

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
```

```
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:



unsigned right shift operator

In Java, the >>> operator is known as the unsigned right shift operator. It is a bitwise operator used to perform a right shift on the binary representation of a number, treating the value as an unsigned quantity. The >>> operator is distinct from the regular right shift operator >> because it fills the vacant leftmost positions with zeros, irrespective of the sign bit.

The syntax of the unsigned right shift operator is:
int result = value >>> n;

Here, value is the number whose binary representation is to be shifted, and n is the number of positions to shift the bits to the right.
When using the >>> operator, the most significant bit (the leftmost bit) is not preserved, and the leftmost positions are filled with zeros.

```
package com.anudip.example;
public class OperatorExample {
        public static void main(String[] args) {
         int number = -16; // Binary representation: 11111111 11111111 11111111 11110000
         // Performing unsigned right shift by 2 positions
         int result = number >>> 2; // Result: 1073741820
         // Binary representation of result: 00111111 11111111 11111111 11111100
         System.out.println("Original number: " + number);
         System.out.println("Result after unsigned right shift: " + result);
        }
```

}

Output:

In the example above, we use the >>> operator to perform an unsigned right shift by 2 positions on the number -16. The result is 1073741820, which corresponds to the binary representation 00111111 11111111 11111111 11111100.

It's important to note that since Java does not have unsigned data types (except for char), the >>> operator is typically used when working with positive values or when you want to interpret negative values as positive unsigned values.

Bitwise AND (&)

The bitwise & operator always checks both conditions whether the first condition is true or false.

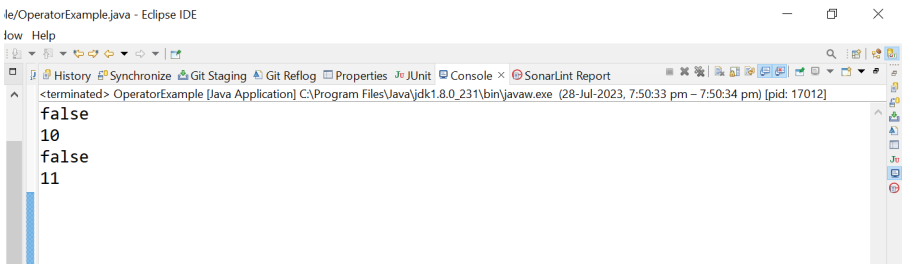
```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&a<c); //false & true = false
}}
```

Output:

Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c); //false && true = false
System.out.println(a);  //10 because second condition is not checked
System.out.println(a<b & a++<c);  //false && true = false
System.out.println(a); //11 because second condition is checked
}}
```

Output:



Bitwise OR (|)

In Java, the bitwise OR operator is represented by the | symbol. It performs a bitwise OR operation on the individual bits of two integer operands. The result is a new integer where each bit is set to 1 if at least one of the corresponding bits in the operands is 1; otherwise, the bit is set to 0.

```
int num1 = 12; // Binary: 1100
int num2 = 7;  // Binary: 0111

int result = num1 | num2; // Binary result: 1111 (Decimal: 15)
System.out.println(result);
```

output:

Explanation:

- The binary representation of num1 is 1100, and the binary representation of num2 is 0111.
- Performing the bitwise OR operation, we get 1111, which is the binary representation of the decimal number 15.

Logical || vs Bitwise |

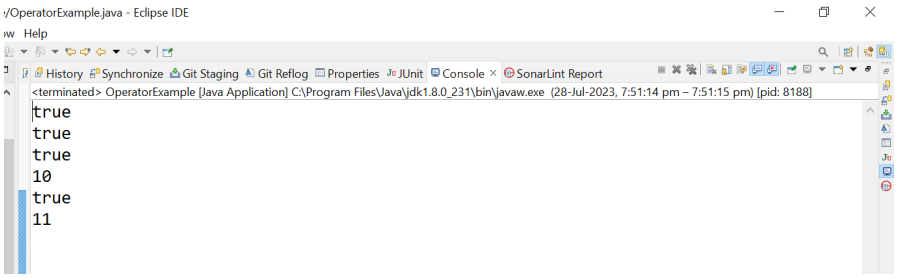
The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether the first condition is true or false.

```
public class OperatorExample{

public static void main(String args[]){

int a=10;

int b=5;

int c=20;

System.out.println(a>b||a<c);  //true || true = true

System.out.println(a>b|a<c);  //true | true = true
```

```
// || vs |
System.out.println(a>b||a++<c);  //true || true = true
System.out.println(a);  //10 because second condition is not checked
System.out.println(a>b|a++<c);  //true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:



After 1.5 hrs of ILT, the trainer needs to assign the below lab to the students in VPL.

Lab (30mins): (Attempt any 2)

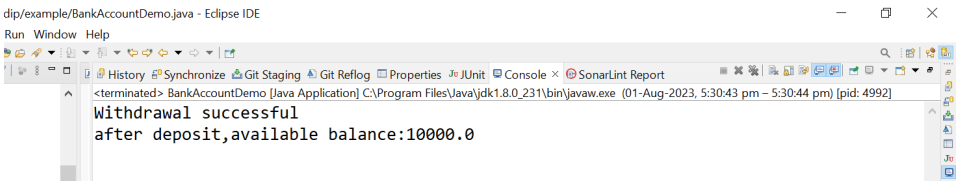
1. Create a Bank class and declare an instance variable named amount of type double.Create parameterized constructor to initialize variable “amount” with value 10000.Create two methods withdraw(double withdrawalAmount) and deposit(double depositAmount).Calculate withdrawal based on some condition (using ternary operator) like If amount is sufficient then “withdraw successful” message will be printed on the console and amount should be updated after withdraw. Later on, deposit 5000 in the account balance.At the end display total balance on the console.

String message = (withdrawalAmount <= amount) ? "Withdrawal successful" : "Insufficient balance";
[Hint: Use constructor, ternary operator]

Sample input:
Amount=10000

Withdrawal amount=5000
Deposit amount=5000

Expected output:



2. Write a program to input two numbers and find the maximum between two numbers using the conditional/ternary operator.

Sample Input:

num1 = 10
num2 = 30

Expected Output:

3. Write a program to declare two variables num and n and take an input during compilation time to check whether the nth bit of the given number is **set (1)** or **not (0)**.

Logic to get nth bit of a number

Step by step descriptive logic to get the nth bit of a number.

1. Take an input of any number and Store it in some variable, say **num**.
2. Take an Input the bit position and Store it in some variable, say **n.**
3. To get the nth bit of num right shift num, n times. Then perform bitwise AND with 1 i.e. bitStatus = (num >> n) & 1.

Sample Input:

Input number: num=12
Input nth bit number: n=2

Expected output: