FLOW CONTROL

Jump Statements
Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement
As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

```
public class BreakExample {


public static void main(String[] args) {

// TODO Auto-generated method stub

for(int i = 0; i<= 10; i++) {

System.out.println(i);

if(i==6) {

break;

}

}

}
}
```

Output:

```
0
1
2
3
4
5
6
```

```
public class Calculation {

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
}

}

}

}

}

}
```

Output:

```
0 1 2 3 4 5
```

Java continue statement
Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {


public static void main(String[] args) {

// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
```

```
continue;
}
System.out.print(j);
}
}
}
}
```

Output:

```
0 1 2 3 5 1 2 3 5 2 3 5
```

OOPS IN DETAIL

ABSTRACTION

Abstraction in Java is the process of hiding the implementation details and only showing the essential functionality or features to the user. This helps simplify the system by focusing on what an object does rather than how it does it. The unnecessary details or complexities are not displayed to the user.

```
// Demonstrating Abstraction in Java

abstract class Abc {

    abstract void turnOn();

    abstract void turnOff();

}


// Concrete class implementing the abstract methods

class TVRemote extends Abc {

    @Override

    void turnOn() {

        System.out.println("TV is turned ON.");

    }


    @Override

    void turnOff() {

        System.out.println("TV is turned OFF.");

    }
}


// Main class to demonstrate abstraction
public class Main {
    public static void main(String[] args) {
         Abc remote = new TVRemote();
        remote.turnOn();
        remote.turnOff();
    }
}
```

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces. Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Abstract Classes and Abstract Methods

- An abstract class is a class that is declared with an abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A abstract method must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

ENCAPSULATION

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

```
// Java program to demonstrate

// Java encapsulation


class Encapsulate {

    // private variables declared

    // these can only be accessed by

    // public methods of class

    private String geekName;

    private int geekRoll;

    private int geekAge;


    // get method for age to access

    // private variable geekAge

    public int getAge() { return geekAge; }


    // get method for name to access

    // private variable geekName

    public String getName() { return geekName; }
    {

    // get method for roll to access
    // private variable geekRoll
    public int getRoll() { return geekRoll; }

    // set method for age to access
    // private variable geekage
    public void setAge(int newAge) { geekAge = newAge; }

    // set method for name to access
    // private variable geekName
    public void setName(String newName)
    {
        geekName = newName;
    }

    // set method for roll to access
    // private variable geekRoll
    public void setRoll(int newRoll) { geekRoll = newRoll; }
}

public class TestEncapsulation {
    public static void main(String[] args)
    {
        Encapsulate o = new Encapsulate();

        // setting values of the variables
        o.setName("Harsh");
        o.setAge(19);
        o.setRoll(51);

        // Displaying values of the variables
        System.out.println("Geek's name: " + o.getName());
        System.out.println("Geek's age: " + o.getAge());
        System.out.println("Geek's roll: " + o.getRoll());

        // Direct access of geekRoll is not possible
        // due to encapsulation
        // System.out.println("Geek's roll: " +
        // obj.geekName);
    }
}
```

- In encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its own class.
- A private class can hide its members or methods from the end user, using abstraction to hide implementation details, by combining data hiding and abstraction.

- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

INHERITANCE

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well

Need of inheritance:

- Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- Abstraction: The concept of abstract where we do not have to provide all details, is achieved through inheritance. Abstraction only shows the functionality to the user.

Important Terminologies Used in Java Inheritance

Class: Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.

Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).

Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

The extends keyword is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

```
// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}

// derived class
class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
    {
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}

// driver class
public class Test {
    public static void main(String args[])
    {
        MountainBike mb = new MountainBike(3, 100, 25);

        System.out.println(mb.toString());

    }

}
```

Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, ‘A’ is a parent class and ‘B’ is a child class. The class ‘B’ inherits all the properties of the class ‘A’.

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent’s members.

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.

POLYMORPHISM

The word ‘polymorphism’ means ‘having many forms’. In Java, polymorphism refers to the ability of a message to be displayed in more than one form. This concept is a key feature of Object-Oriented Programming and it allows objects to behave differently based on their specific class type.

Types of Java Polymorphism

In Java Polymorphism is mainly divided into two types:

1.Compile-Time Polymorphism(Method Overloading)

Method overloading in Java means when there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

```
// Method overloading By using

// Different Types of Arguments


// Class 1

class Harman {


    // Method with 2 integer parameters

    static int Multiply(int a, int b)

    {
        // Returns product of integer numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class Harman
{
    // Main driver method
    public static void main(String[] args) {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

2. Runtime Polymorphism

Runtime Polymorphism in Java known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

MEMORY ALLOCATION OF ARRAY

In Java, arrays are objects and are stored in heap memory. However, the reference to the array is stored in stack memory

Java divides memory into two main areas:

Memory Type	Stores
Stack Memory	Local variables (e.g., array references)
Heap Memory	Actual array objects (data storage)

String:

Example: Allocating a String[] Array

```
public class ArrayMemoryTest {

    public static void main(String[] args) {

        String[] strArray = new String[5];  // Step 1: Create an array

        strArray[0] = "Hello";            // Step 2: Assign values

        strArray[1] = "World";

    }

}
```

Memory Breakdown

Step 1: Creating the String[] Array

String[] strArray = new String[5];

Stack Memory: Stores the reference strArray.

Heap Memory: Allocates space for the String[] array (5 references). Each reference is initially null.

```
Stack:                    Heap:

┌────────────────────────┐
┌─────────────────────────────────────────────┐
│ strArray Ref │ ───▶     │ [null, null, null, null, null] │
└────────────────────────┘
└─────────────────────────────────────────────┘
```

The array object itself (5 references) is stored in the heap.

Step 2: Assigning String Values

strArray[0] = "Hello";

strArray[1] = "World";

The "Hello" and "World" String objects are also stored in the heap.

The array references are updated to point to these objects.

Memory Structure (After Assignment)

```
Stack:                    Heap:

┌────────────────────────┐
┌──────────────────────────────────────────────────┐
│ strArray Ref │ ───▶     │ [Ref1, Ref2, null, null, null] │
└────────────────────────┘
└──────────────────────────────────────────────────┘

                ▲     ▲
                │     │
```

"Hello" "World" (Stored in String Pool)

The actual String objects ("Hello" and "World") are interned in the String Pool.

The strArray reference still resides in stack memory.

Component	Memory Location
String[] strArray reference	Stack Memory
String[] array object (holds references)	Heap Memory
Actual String objects ("Hello", "World")	Heap Memory (String Pool)

Integer:

If we allocate an int[] array:

int[] intArray = new int[5];

intArray reference → Stored in stack

int[] object (actual numbers) → Stored in heap

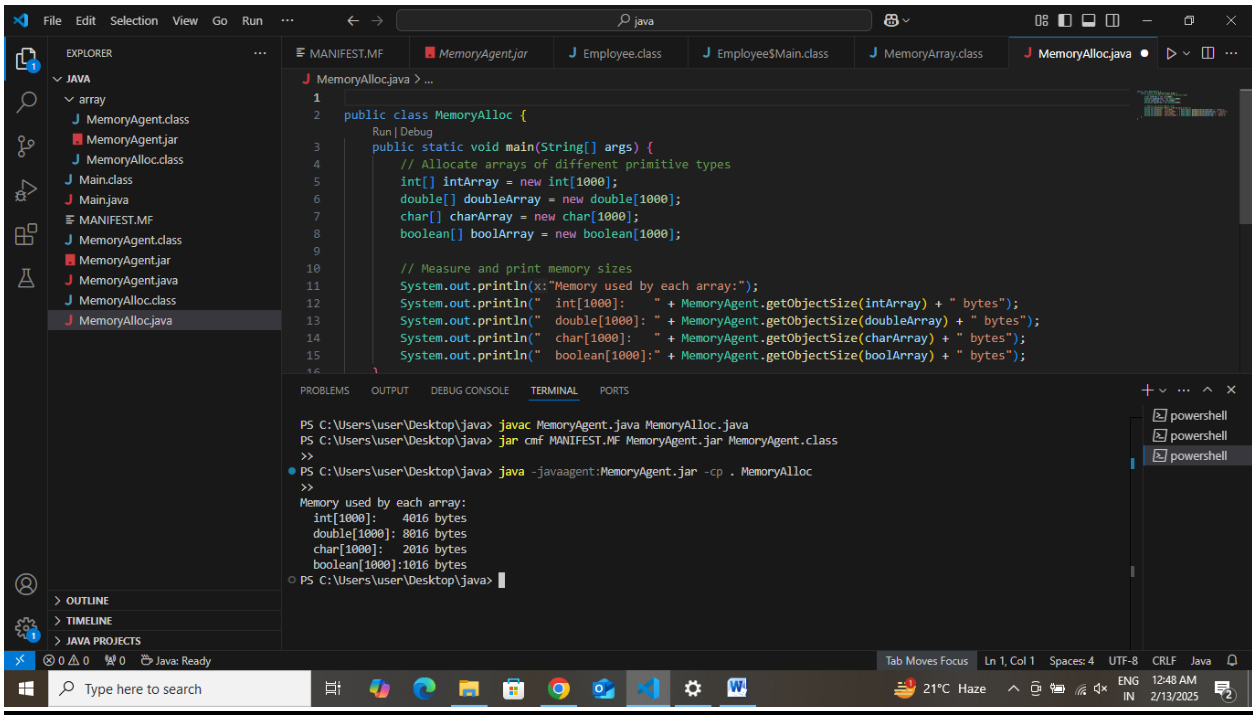
Each integer takes 4 bytes (JVM dependent)

Memory Layout

Stack: Heap:

```
┌───────────────────┐
┌────────────────────────────────────┐
│ intArray Ref │ ──► │ [0, 0, 0, 0, 0] (5 * 4B) │
└───────────────────┘
└────────────────────────────────────┘
```

HOW TO CHECK JAVA MEMORY ALLOCATION(ARRAY)

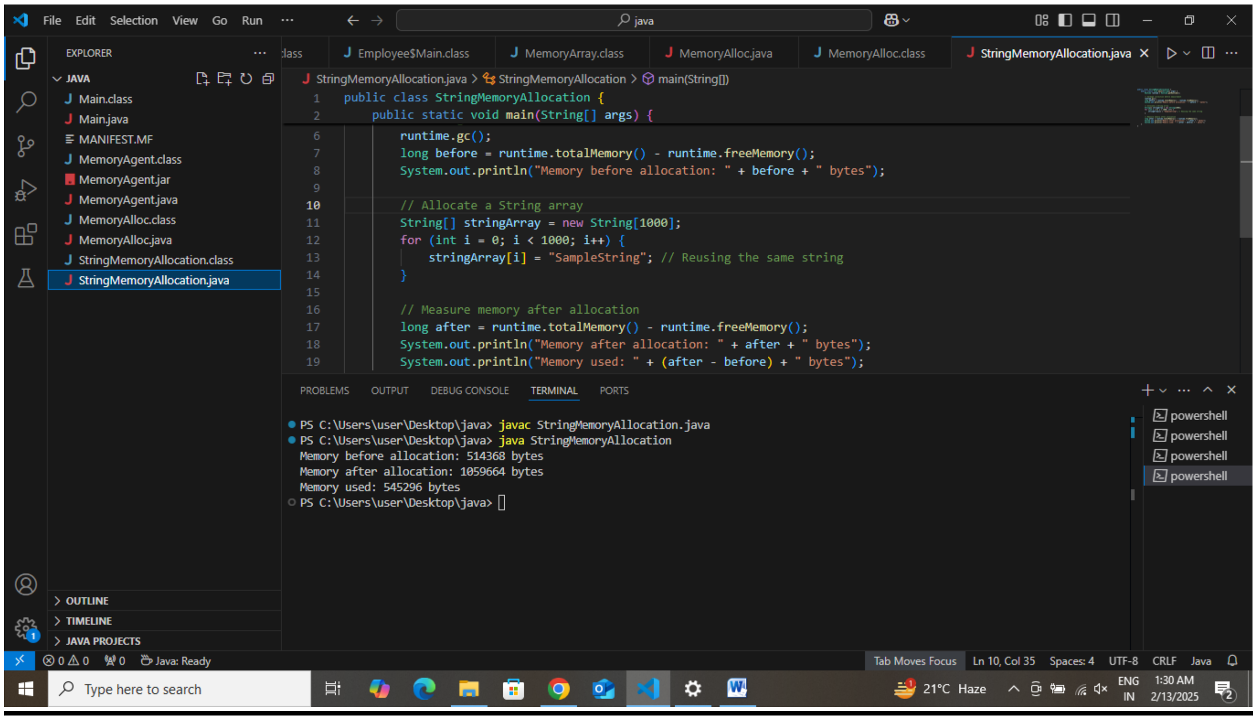


```java

public class MemoryAlloc {
    public static void main(String[] args) {
        // Allocate arrays of different primitive types
        int[] intArray = new int[1000];
        double[] doubleArray = new double[1000];
        char[] charArray = new char[1000];
        boolean[] boolArray = new boolean[1000];

        // Measure and print memory sizes
        System.out.println(x:"Memory used by each array:");
        System.out.println("  int[1000]:     " + MemoryAgent.getObjectSize(intArray) + " bytes");
        System.out.println("  double[1000]: " + MemoryAgent.getObjectSize(doubleArray) + " bytes");
        System.out.println("  char[1000]:    " + MemoryAgent.getObjectSize(charArray) + " bytes");
        System.out.println("  boolean[1000]:" + MemoryAgent.getObjectSize(boolArray) + " bytes");
```

```
PS C:\Users\user\Desktop\java> javac MemoryAgent.java MemoryAlloc.java
PS C:\Users\user\Desktop\java> jar cmf MANIFEST.MF MemoryAgent.jar MemoryAgent.class
>>
PS C:\Users\user\Desktop\java> java -javaagent:MemoryAgent.jar -cp . MemoryAlloc
>>
Memory used by each array:
  int[1000]:    4016 bytes
  double[1000]: 8016 bytes
  char[1000]:   2016 bytes
  boolean[1000]:1016 bytes
PS C:\Users\user\Desktop\java>
```



```java
public class StringMemoryAllocation {
    public static void main(String[] args) {
        runtime.gc();
        long before = runtime.totalMemory() - runtime.freeMemory();
        System.out.println("Memory before allocation: " + before + " bytes");

        // Allocate a String array
        String[] stringArray = new String[1000];
        for (int i = 0; i < 1000; i++) {
            stringArray[i] = "SampleString"; // Reusing the same string
        }

        // Measure memory after allocation
        long after = runtime.totalMemory() - runtime.freeMemory();
        System.out.println("Memory after allocation: " + after + " bytes");
        System.out.println("Memory used: " + (after - before) + " bytes");
```

```
PS C:\Users\user\Desktop\java> javac StringMemoryAllocation.java
PS C:\Users\user\Desktop\java> java StringMemoryAllocation
Memory before allocation: 514368 bytes
Memory after allocation: 1059664 bytes
Memory used: 545296 bytes
PS C:\Users\user\Desktop\java>
```