Object Oriented Programming using Java

Notes DECEMBER 23

Dr. Chetan KR
Professor and Head
Dept. of AIML

Module-1

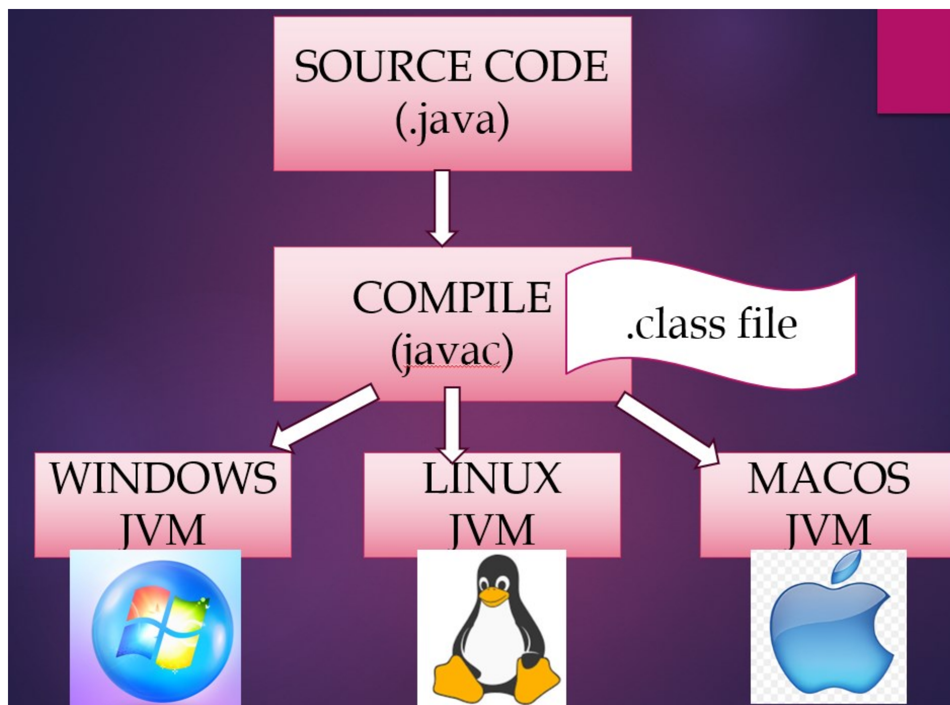
Overview of Java

Java is a widely used object-oriented programming language and software platform that runs on billions of devices, including notebook computers, mobile devices, gaming consoles, medical devices and many others. The rules and syntax of Java are based on the C and C++ languages. One major advantage of developing software with Java is its portability. Once you have written code for a Java program on a notebook computer, it is very easy to move the code to a mobile device. When the language was invented in 1991 by James Gosling of Sun Microsystems (later acquired by Oracle), the primary goal was to be able to "write once, run anywhere."

It's also important to understand that Java is much different from JavaScript. Javascript does not need to be compiled, while Java code does need to be compiled. Also, Javascript only runs on web browsers while Java can be run anywhere.

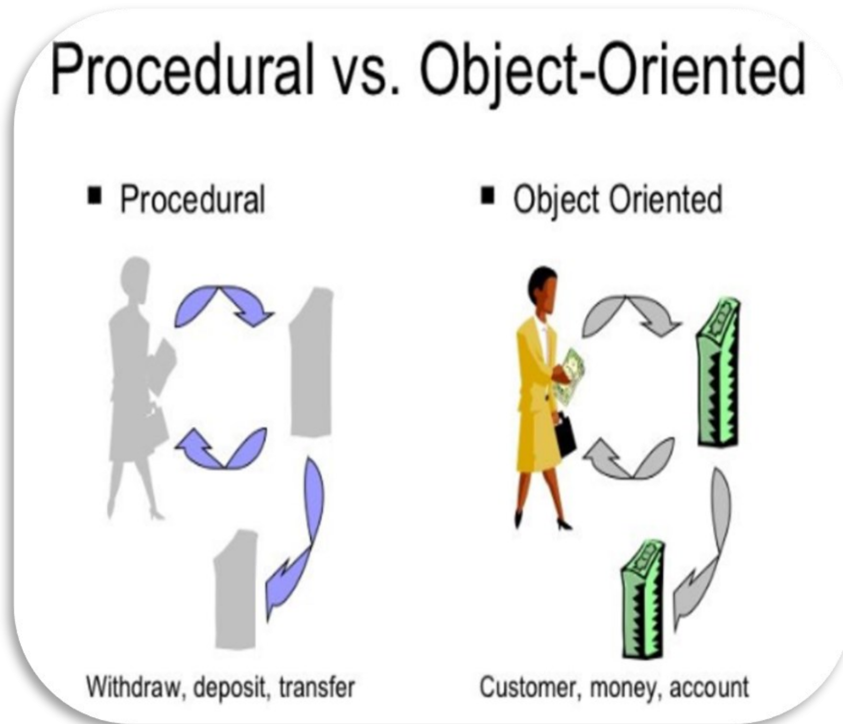
New and improved software development tools are coming to market at a remarkable pace, displacing incumbent products once thought to be indispensable. In light of this continual turnover, Java's longevity is impressive; more than two decades after its creation, Java is still the most popular language for application software development—developers continue to choose it over languages such as Python, Ruby, PHP, Swift, C++, and others. As a result, Java remains an important requirement for competing in the job market.

Working of Java



Java programs are written and saved in a file with extension .java. The program is compiled using javac (Java Compiler). A class file is generated and it is independent of any platform. To execute class file on a specific machine, Java Virtual Machine (JVM) of that machine is used.

Two paradigms of programming



Differences between 2 paradigms

STRUCTURED/ PROCEDURAL	OBJECT ORIENTED
Code divided into procedures/functions	Code divided into classes
Top down approach	Bottom up approach
Communication via function calls	Objects communicate with message passing
Less reusable (function)	More resuable (class)
Less flexible (function)	More flexible (class)
No data security	Provides data security

Abstraction

Abstraction is the process of hiding the internal details of an application from the outer world. Abstraction is used to describe things in simple terms. It's used to create a boundary between the application and the client programs. It shows only the necessary information and hides the other irrelevant information. Abstraction is implemented using Abstraction classes and interfaces . The problems in Abstraction are solved at design or interface level.

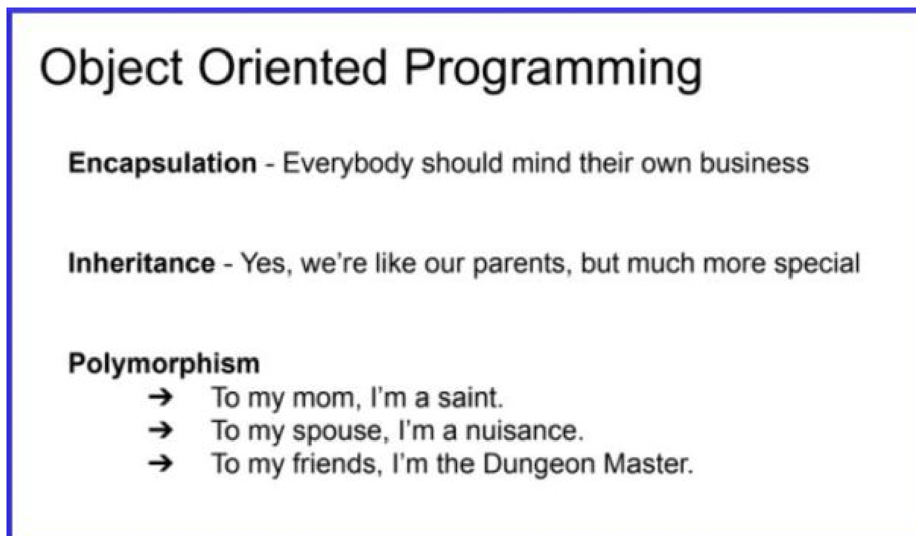
Objects are the building blocks of Object-Oriented Programming. An object contains some properties and methods. We can hide them from the outer world through access modifiers. We can provide access only for required functions and properties to the other programs. This is the general procedure to implement abstraction in OOPS.

Abstraction can be found in practically all real-world machines.

1. Automobile is an excellent illustration of abstraction. Turning the key or pressing the start button is how you start a car. You don't need to know how your car's engine starts or what components it contains. The user is completely unaware of the car's internal implementation and complicated logic.
2. Heating the food in Microwave. We press some buttons to set the timer and type of food. Finally, we get a hot and delicious meal. The microwave internal details are hidden from us. We have been given access to the functionality in a very simple manner.

Three OOP principles

There are three major pillars on which object-oriented programming relies: encapsulation, inheritance, and polymorphism.



1. Encapsulation: This is the idea of wrapping everything up about a particular thing, whether a Checking Account or Armadillo, into a defined object with features and behaviors. Once we do, we can ask the object itself to do what it is supposed to do, whether that is Deposit Money or Defend Yourself. But nobody outside the object needs to worry about how it does its jobs. We just tell it to do it and go about our day. If every object, simply minds its own business and stays out of the business of other objects, all is good with the world.
2. Inheritance: This is the idea that we don't have to define absolutely everything about an object over and over again if it shares features and behaviors with other objects. We can

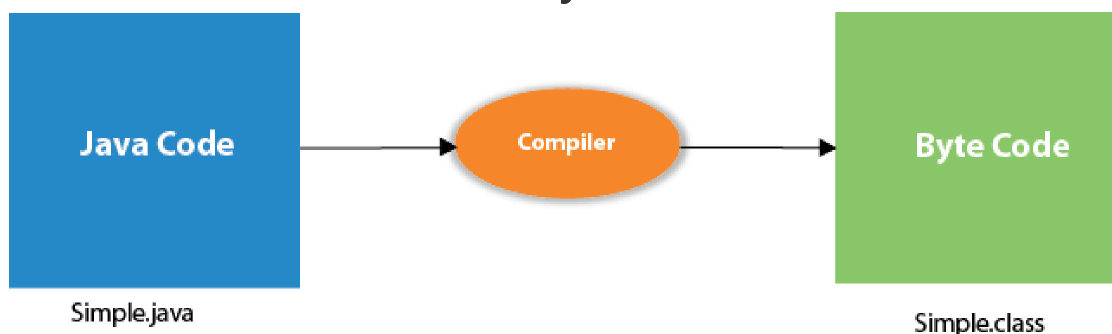
define a class for Accounts and then let our Checking Account or Savings Account inherit all the stuff in common. Likewise, we can define a class for Animals, and let our Armadillo inherit features like Number Of Legs and Weight as well as behaviors such as Breathe and Sleep. We call these overarching classes parent classes, and the ones that inherit from them, child classes. We can then inherit from the child classes and so on. But our Checking Account is more specialized than our Accounts because we can Write A Check, which we can't do with a Savings Account. Our Armadillo can Roll Into A Ball, but other animals such as a Giraffe don't have that behavior. Since we go from more general to more specialized, I like to say that a child is like its parents, but much more special.

3. Polymorphism: This fancy name just means that we can treat the same object as different things depending on how we need it at different times, and we can treat groups of different objects that share an ancestor or trait as if they were that ancestor or trait. So, we could have a set of different Checking, Savings, and Credit Accounts and ask each to Get Balance so we can figure out how much we have to spend on vacation this year. Or we could ask a queue of animals to Move Quickly, and not care how the Porpoise or Eagle or Armadillo would handle that shared behavior. I like to think that we are different things to different people, so even if not, every Dungeon Master has a spouse to think him or her a nuisance, we can ask any of them to organize a game for Saturday night.

Basic Java program

```
class Simple{
   public static void main(String args[]){
    System.out.println("Hello Java");
   }
}
```

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



- **class** keyword is used to declare a class in Java.

- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

Other Example basic Java programs

Greeting the user with user name accepted through keyboard

```
import java.util.*;
class Greet
{
   public static void main(String args[])
   {
          System.out.println("Enter your name");
         Scanner kb=new Scanner(System.in);
       String   name=kb.nextLine();
       System.out.println("Welcome "+ name);
   }
}
```

Finding taller of 2 person based on their height accepted through keyboard

```
import java.util.*;
class Taller
{
    public static void main(String aiml[])
    {
           System.out.println("Enter name1");
          Scanner kb=new Scanner(System.in);
           String name1=kb.nextLine();
           System.out.println("Enter the height");
            double height1=kb.nextDouble();
            kb.nextLine();
           System.out.println("Enter name2");
           String name2=kb.nextLine();
           System.out.println("Enter the height");
```

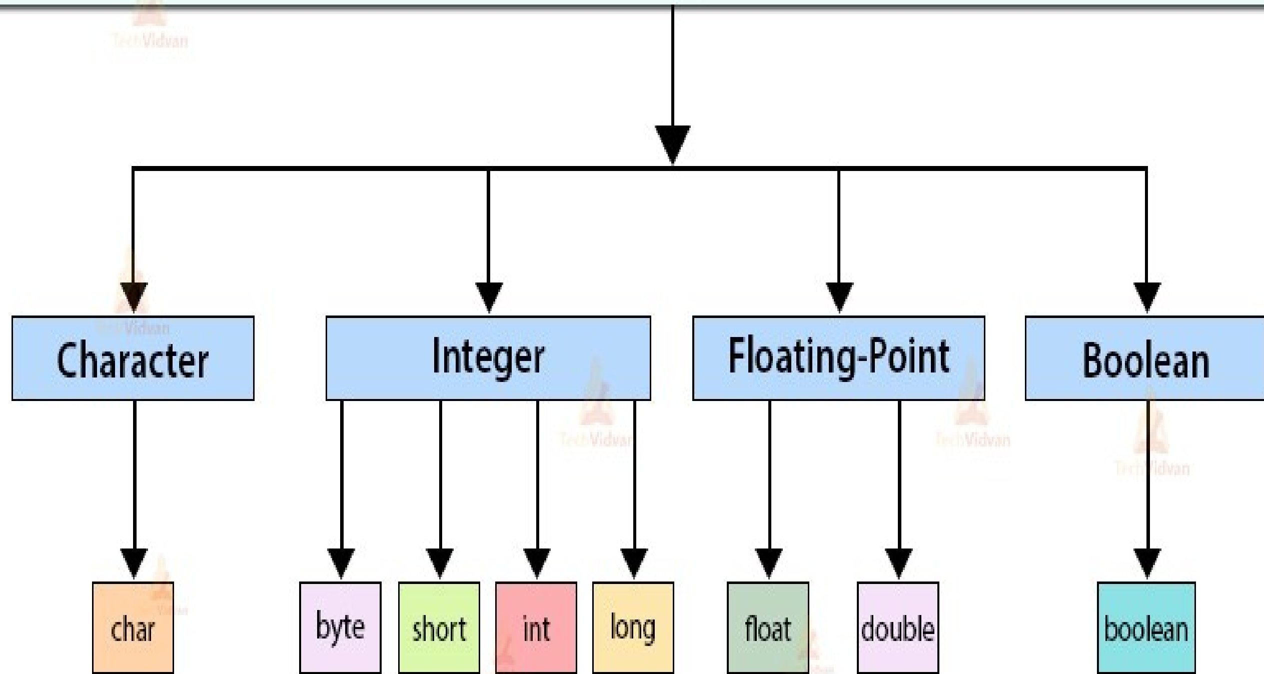
```
            double height2=kb.nextDouble();
            if(height1>height2)
                    System.out.println(name1+" is taller");
            else
                  System.out.println(name2+" is taller");
     }

}
```

Finding the tallest person among a set of people

```
import java.util.*;
class Tallest
{
   public static void main(String anything[])
   {
           String refname="";
           double refheight=0.0;
           for(int i=0;i<5;i++)
           {
                //Scan the name and height of ith person
               System.out.println("Enter the name");
               Scanner kb=new Scanner(System.in);
               String name=kb.nextLine();
               System.out.println("Enter the height");
               double height=kb.nextDouble();
               if(height>refheight)
               {
                       refheight=height;
                       refname=name;
               }
           }
           System.out.println(refname+" is tallest");
   }
}
```

Primitive Data Types in Java



In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte

double	0.0d	8 byte

Write a Java program to compute the distance travelled on 'n' days, given speed of light=18600 miles

```
import java.util.*;
class LightDistance
{
       public static void main(String anything[])
       {
           long speed=186000;
          //Read n from the keyboard
          short n;
          System.out.println("Enter the number of days(n)");
          Scanner kb=new Scanner(System.in);
          n=kb.nextShort();
          //Compute distance
          //distance=speed*n*24*60*60
          long distance=speed*n*24*60*60;
          System.out.println(distance);

       }
}
```

Write a Java program to compute the area of the Circle

```
import java.util.*;
class AC
{
  public static void main(String[] aids)
  {
      System.out.println("Enter the radius");
      Scanner kb=new Scanner(System.in);
      float radius=kb.nextFloat();
      float area=(float)Math.PI*radius*radius;
      System.out.println(area);
  }

}
```

Characters in Java

The data type char comes under the characters group that represents symbols i.e. alphabets and numbers in a character set. The Size of a Java char is 16-bit and the range is between 0 to 65,535. Also, the standard ASCII characters range from 0 to 127.

Syntax:
char variable_name = ‘variable_value’;

Given below are the major characteristics of a char.

1. As mentioned above, the range is between 0 to 65,535.
2. The default value is ‘\u0000’ and that is the lowest range of Unicode.
3. The default size (as mentioned above) is 2 bytes because Java uses the Unicode system and not the ASCII code system.

Write a Java program to compute all the alphabets

```
class PrintAlphabets
{
     public static void main(String[] icantwrite)
     {
          for(char c='A';c<='Z';c++)
       {
                System.out.print(c);
          }
       for(char c='a';c<='z';c++)
       {
              System.out.print(c);
          }
     }
}
```

Boolean keyword in Java

In Java, the boolean keyword is a primitive data type. It is used to store only two possible values, either true or false. It specifies 1-bit of information and its "size" can't be defined precisely. The boolean keyword is used with variables and methods. Its default value is false. It is generally associated with conditional statements.

Write a Java program to check student passed or failed in a subject

```
import java.util.*;
class PF
{
      public static void main(String[] passfail)
      {
              System.out.println("Enter the marks");
           Scanner kb=new Scanner(System.in);
           int marks=kb.nextInt();
           if(marks<40)
          System.out.println("Failed");
           else
          System.out.println("Passed");
```

```
        }
}
```

Literals in Java

Literals in Java are a synthetic representation of boolean, character, numeric, or string data. They are a means of expressing particular values within a program. They are constant values that directly appear in a program and can be assigned now to a variable. For example, here is an integer variable named ‘’/count assigned as an integer value in this statement:

int count = 0;
“int count” is the integer variable, and a literal ‘0’ represents the value of zero.
Therefore, a constant value that is assigned to the variable can be called a literal.

Types of Literals in Java

Literals in Java are typically classified into six types and then into various sub-types. The primary literal types are:

1. Integral Literals

Integral literals consist of digit sequences and are broken down into these sub-types:

- Decimal Integer: Decimal integers use a base ten and digits ranging from 0 to 9. They can have a negative (-) or a positive (+), but non-digit characters or commas aren’t allowed between characters. Example: 2022, +42, -68.
- Octal Integer: Octal integers use a base eight and digits ranging from 0 to 7. Octal integers always begin with a “0.” Example: 007, 0295.
- Hexa-Decimal: Hexa-decimal integers work with a base 16 and use digits from 0 to 9 and the characters of A through F. The characters are case-sensitive and represent a 10 to 15 numerical range. Example: 0xf, 0xe.
- Binary Integer: Binary integers uses a base two, consisting of the digits “0” and “1.” The prefix “0b” represents the Binary system. Example: 0b11011.

Floating-Point Literals

Floating-point literals are expressed as exponential notations or as decimal fractions. They can represent either a positive or negative value, but if it’s not specified, the value defaults to positive. Floating-point literals come in these formats:

- Floating: Floating format single precision (4 bytes) end with an “f” or “F.” Example: 4f. Floating format double precision (8 bytes) end with a “d” or “D.” Example: 3.14d.
- Decimal: This format uses 0 through 9 and can have either a suffix or an exponent. Example: 99638.440.
- Decimal in Exponent form: The exponent form may use an optional sign, such as a "-," and an exponent indicator, such as "e" or "E." Example: 456.5f.

3. Char Literals

Character (Char) literals are expressed as an escape sequence or a character, enclosed in single quote marks, and always a type of character in Java. Char literals are sixteen-bit Unicode characters ranging from 0 to 65535. Example: char ch = 077.

4. String Literals

String literals are sequences of characters enclosed between double quote ("") marks. These characters can be alphanumeric, special characters, blank spaces, etc.
Examples: "John", "2468", "\n", etc.

5. Boolean Literals

Boolean literals have only two values and so are divided into two literals:

- True represents a real boolean value
- False represents a false boolean value

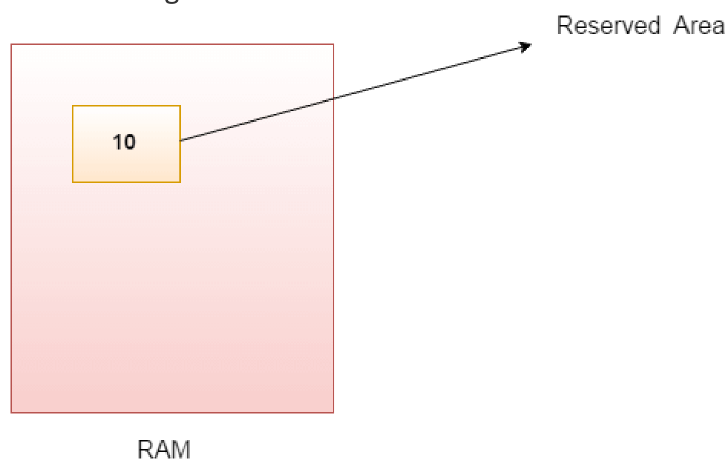
So, Boolean literals represent the logical value of either true or false. These values aren't case-sensitive and are equally valid if rendered in uppercase or lowercase mode. Boolean literals can also use the values of “0” and “1.”
Examples:
boolean b = true;
boolean d = false;

6. Null Literals

Null literals represent a null value and refer to no object. Nulls are typically used as a marker to indicate that a reference type object isn’t available. They often describe an uninitialized state in the program. It is a mistake to try to dereference a null value. Example: Patient age = NULL;
Remember, not everyone divides literals in Java into these six types. Alternative classifications split literals into as few as four types (Integer, Character, Boolean, and String) or as many as ten (Integer, Real, Backslash, Character, String, Floating-Point, Boolean, NULL, Class, and Invalid).

Variables in Java

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. There are three types of variables in java: local, instance and static. A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.




```
int data=10;//Here data is variable
```

Declaration, Initialization and Assignment in Java

Declaration: Declaration is when you declare a variable with a name, and a variable can be declared only once.

Example: int x;, String myName;, Boolean myCondition;

Initialization: Initialization is when we put a value in a variable, this happens while we declare a variable.

Example: int x = 7;, String myName = "Emi";, Boolean myCondition = false;

Assignment: Assignment is when we already declared or initialized a variable, and we are changing the value. You can change value of the variable as many time you want or you need.

Example:

int x = 7; x = 12; ....We just changed the value.

String myName = "Emi"; myName = "John" ....We just changed the value.

Boolean myCondition = false; myCondition = true; ....We just changed the value.

Note: In memory will be saved the last value that we put.

Dynamic Initialization of variables

The initialization of variables is done during runtime.

```
class RH
{
        public static void main(String any[])
        {
             double a=5;
            double b=3;
            double c=Math.sqrt(a*a+b*b); //Dynamic initialization
            System.out.println(c);

        }
}
```

Scope and Lifetime of variables

The *scope* of a variable refers to the areas or the sections of the program in which the variable can be accessed, and the *lifetime* of a variable indicates how long the variable stays alive in the memory. A joint statement defining the scope and lifetime of a variable is *"how and where the variable is defined."* Let me simplify it further. The comprehensive practice for the *scope* of a variable is that it is accessible only inside the block it is declared.

Types of Variables and its Scope

There are three types of variables.

1. Block Variables
2. Method Variables
3. Class Variables

Block Scope
If a variable is defined/declared in some block of code, then it exists until the end of that block of code. Typically, such variables exist between the curly braces in which they are defined. Very often block scope could be a loop variable. A variable that is declared in a for loop condition is not accessible outside the loop, unless you defined it beforehand.

Method Level Scope
Any variable declared in a method, including arguments, is not accessible outside of that method. All variables declared inside methods are visible from the beginning of their declaration to the end of the method.

Class Level Scope
Class-Level Scope (Instance Variables) — any variable declared in a class is available for all methods of that class. Depending on its access modifier (i.e. public or private), it can sometimes be accessed outside of the class. So if a variable is a class variable, then it is bound to a specific object and exists as long as there is an object of this class. If there is no object, then there is no copy of the variable. The variable is visible from all methods of the class, regardless of whether they are declared before or after it. Each object has its own variable independent of other objects. Access to a variable from static methods is not possible.

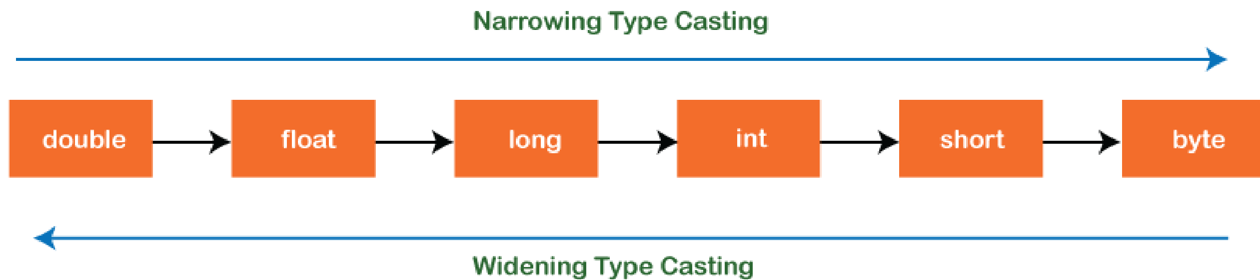
Java program demonstrating scope and lifetime of variables

```
class ScopeLife
{
	//static int a=2;
	public static void main(String[] p)
	{
		//System.out.println(a);
		int a=3;
		System.out.println(a);
		for(int a=0;a<4;a++)
		 {
			System.out.println(a);
		}
	}
	//System.out.println(a);
}
```

Type conversion and casting in Java

In Java, type conversion and casting is a method that converts a data type into another data type in both ways manually or automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.
Type casting converts a value from one data type to another data type. There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting



Type Casting in Java

Widening Type Casting

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

byte -> short -> char -> int -> long -> float -> double

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other.

Narrowing Type Casting

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

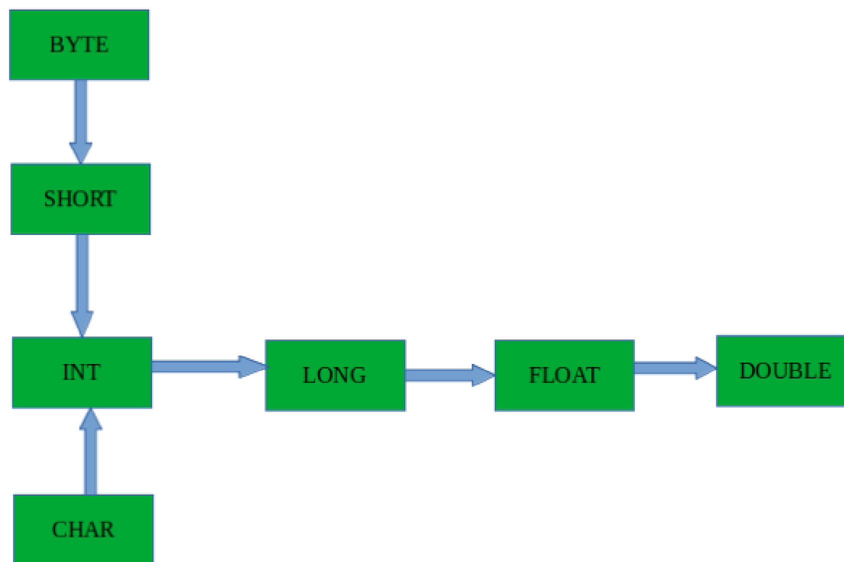
Java program demonstrating type casting

```
class BC
{
	public static void main(String[] abc)
	{
		int i=-140;
		byte b=(byte)i;
		System.out.println(b);
		double d=343.65;
		int x=(int)d;
		System.out.println(x);


	}
}
```

Automatic Type promotions in Java

Type promotion is a common occurrence in Java programming, which can be achieved automatically with primitive data types through the use of autotype promotion. It is also referred to as automatic data type promotion. Below is a diagrammatic illustration of possible type promotions:



In general, the largest data type in an expression is the one that determines the size of the result of that expression; if you multiply a float and a double, the result will be double; if you add an int and a long, the result will be long. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a*b** easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a*b** is performed using integers — not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the

result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.
In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

Rules for automatic type promotion
The following rules for type promotion must be followed when executing expressions in Java to achieve correct results:

- All variables of the types **byte**, **short**, and **char** must be auto type promoted to **int**.
- If any variable taking part in an operation is **long**, the operation result must be **long**.
- If any variable taking part in an operation is **float**, the operation result must **float**.

in sample words all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

Java program to demonstrate Type promotions

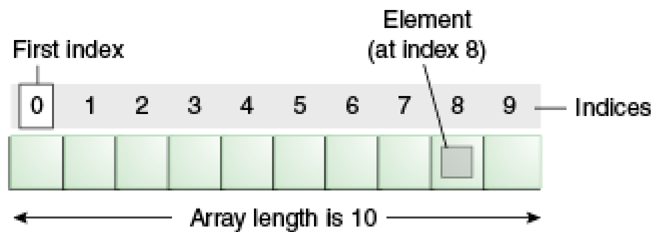
```
class TP
{
        public static void main(String[] any)
        {
                byte b=5;
                short s=10;
                char c=3;
                int i=10;
                long l=20;
                float f=3.1f;
                double dd=11.4;
                double ans=(i/s)+(b*c)+(dd/f);
                System.out.println(ans);
        }
}
```

Arrays in Java

An array is a collection of similar type of elements which has contiguous memory location. Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on. Unlike C/C++, we can get the length of the array using the length member.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or

objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java. Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- Random access: We can get any data located at an index position.

Disadvantages

- Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];

Instantiation of an Array in Java

arrayRefVar=new datatype[size];

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

Java program to demonstrate one dimensional array by computing average

```
class Darr
{
        public static void main(String[] any)
        {
                double d[]={4.1,3.7,1.6,9.2,2.6,1.2};
                double sum=0.0;
                for(int i=0;i<d.length;i++)
                        sum=sum+d[i];
                double avg=sum/d.length;
                System.out.println(avg);
        }
}
```

Multidimensional Arrays in Java

A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
int[][] a = new int[3][4];
```

Here, we have created a multidimensional array named a. It is a 2-dimensional array, that can hold a maximum of 12 elements. Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1.

Let's take another example of the multidimensional array. This time we will be creating a 3-dimensional array. For example,

```
String[][][] data = new String[3][4][2];
```

Here, data is a 3d array that can hold a maximum of 24 (3*4*2) elements of type String.

Here is how we can initialize a 2-dimensional array in Java.

```
int[][] a = {
    {1, 2, 3},
    {4, 5, 6, 9},
    {7},
```

Let's see how we can use a 3d array in Java. We can initialize a 3d array similar to the 2d array. For example,

```
// test is a 3d array
int[][][] test = {
        {
          {1, -2, 3},
          {2, 3, 4}
        },
        {
          {-4, -5, 6, 9},
          {1},
          {2, 3}
        }
};
```

Basically, a 3d array is an array of 2d arrays. The rows of a 3d array can also vary in length just like in a 2d array.

Java program for the construction of 3D Arrays

```
class ThreeD
{
```

```
        public static void main(String[] any)
        {
                int[][][] t=new int[3][4][5];
                for(int i=0;i<3;i++)
                {
                        for(int j=0;j<4;j++)
                        {
                                for(int k=0;k<5;k++)
                                {
                                        t[i][j][k]=i*j*k;
                                        System.out.print(t[i][j][k]+" ");
                                }
                                System.out.println();
                        }
                        System.out.println();
                }
        }
}
```

Jagged Arrays in Java

A jagged array in Java is a collection of arrays where each array may contain a varied number of elements. A two-dimensional array, in contrast, requires all rows and columns to have the same length. Jagged arrays are also known as "ragged arrays" or "irregular arrays". They can be created by specifying the size of each array in the declaration. For example, a jagged array with three rows can have the first row with three elements, the second with two elements, and the third with four elements.

Syntax:

```
datatype[][] arrayName = new datatype[numRows][];
arrayName[0] = new datatype[numColumns1];
arrayName[1] = new datatype[numColumns2];
...
arrayName[numRows-1] = new datatype[numColumnsN];
```

Here, the datatype is the data type of the elements in the array, numRows is the number of rows in the jagged array, and numColumns1, numColumns2, ..., numColumnsN are the number of columns in each row. Users should be aware that the number of columns in each row is flexible. The first line creates an array of arrays with numRows rows. Each row is initialized to null by default. The subsequent lines initialize each row of the jagged array. You create a new one-dimensional array of numColumns elements for each row and assign it to the corresponding row in the jagged array.

Java program to demonstrate Jagged Arrays

```
class Jagged
{
        public static void main(String[] any)
        {
                int[][] arr=new int[4][];
                arr[0]=new int[3];
                arr[0][0]=9;
                arr[0][1]=6;
                arr[0][2]=5;

                arr[1]=new int[2];
```

```
            arr[1][0]=0;
            arr[1][1]=1;

            arr[2]=new int[5];
            arr[2][0]=1;
            arr[2][1]=5;
            arr[2][2]=8;
            arr[2][3]=7;
            arr[2][4]=6;

            arr[3]=new int[1];
            arr[3][0]=3;

        }
}
```

Local variable type inferencing in Java

Local Variable Type Inference is one of the most evident change to language available from Java 10 onwards. It allows to define a variable using var and without specifying the type of it. The compiler infers the type of the variable using the value provided. This type inference is restricted to local variables.

Old way of declaring local variable.

String name = "Welcome to tutorialspoint.com";

New Way of declaring local variable.

var name = "Welcome to tutorialspoint.com";

Now compiler infers the type of name variable as String by inspecting the value provided.

Noteworthy points

- No type inference in case of member variable, method parameters, return values.
- Local variable should be initialized at time of declaration otherwise compiler will not be infer and will throw error.
- Local variable inference is available inside initialization block of loop statements.
- No runtime overhead. As compiler infers the type based on value provided, there is no performance loss.
- No dynamic type change. Once type of local variable is inferred it cannot be changed.
- Complex boilerplate code can be reduced using local variable type inference.

Program to demonstrate type inferencing in Java

```
class Infer
{
        public static void main(String[] any)
        {
                int x=10;
                var y=10.5;
                y='A';
                int var=10;
                //int int=10;
                System.out.println(x+y);
        }
}
```

Operators in Java

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. In this article, we will learn about Java Operators and learn all their types. Operators in Java are the symbols used for performing specific operations in Java. Operators make tasks like addition, multiplication, etc which look easy although the implementation of these tasks is quite complex.

Types of Operators in Java

There are multiple types of operators in Java all are mentioned below:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators
9. instance of operator

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

- * : Multiplication
- / : Division
- % : Modulo
- + : Addition
- – : Subtraction

Example:

```
// Java Program to implement Arithmetic Operators
import java.io.*;

// Drive Class
class GFG {
    // Main Function
   public static void main (String[] args) {

      // Arithmetic operators
      int a = 10;
      int b = 3;

      System.out.println("a + b = " + (a + b));
      System.out.println("a - b = " + (a - b));
      System.out.println("a * b = " + (a * b));
      System.out.println("a / b = " + (a / b));
      System.out.println("a % b = " + (a % b));
```

```
    }
}
```

Output
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1

2. Unary Operators

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- – : Unary minus, used for negating the values.
- + : Unary plus indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- ++ : Increment operator, used for incrementing the value by 1. There are two varieties of increment operators.
    - Post-Increment: Value is first used for computing the result and then incremented.
    - Pre-Increment: Value is incremented first, and then the result is computed.
- – – : Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operators.
    - Post-decrement: Value is first used for computing the result and then decremented.
    - Pre-Decrement: The value is decremented first, and then the result is computed.
- ! : Logical not operator, used for inverting a boolean value.

Example:

```
// Java Program to implement Unary Operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        // Interger declared
        int a = 10;
        int b = 10;

        // Using uniary operators
        System.out.println("Postincrement : " + (a++));
```

```
        System.out.println("Preincrement : " + (++a));

        System.out.println("Postdecrement : " + (b--));
        System.out.println("Predecrement : " + (--b));
    }
}
```

Output
Postincrement : 10
Preincrement : 12
Postdecrement : 10
Predecrement : 8

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.
The general format of the assignment operator is:
variable = value;

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a Compound Statement. For example, instead of a = a+5, we can write a += 5.

- +=, for adding the left operand with the right operand and then assigning it to the variable on the left.
- -=, for subtracting the right operand from the left operand and then assigning it to the variable on the left.
- *=, for multiplying the left operand with the right operand and then assigning it to the variable on the left.
- /=, for dividing the left operand by the right operand and then assigning it to the variable on the left.
- %=, for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:

```
// Java Program to implement Assignment Operators
import java.io.*;

// Driver Class
class GFG {
    // Main Function
    public static void main(String[] args)
    {
```

```
        // Assignment operators
        int f = 7;
        System.out.println("f += 3: " + (f += 3));
        System.out.println("f -= 2: " + (f -= 2));
        System.out.println("f *= 4: " + (f *= 4));
        System.out.println("f /= 3: " + (f /= 3));
        System.out.println("f %= 2: " + (f %= 2));
        System.out.println("f &= 0b1010: " + (f &= 0b1010));
        System.out.println("f |= 0b1100: " + (f |= 0b1100));
        System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
        System.out.println("f <<= 2: " + (f <<= 2));
        System.out.println("f >>= 1: " + (f >>= 1));
        System.out.println("f >>>= 1: " + (f >>>= 1));
    }
}
```

Output

```
f += 3: 10
f -= 2: 8
f *= 4: 32
f /= 3: 10
f %= 2: 0
f &= 0b1010: 0
f |= 0b1100: 12
f ^= 0b1010: 6
f <<= 2: 24
f >>= 1: 12
f >>>= 1: 6
```

4. Relational Operators

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,
variable relation_operator value

Some of the relational operators are-

- ==, Equal to returns true if the left-hand side is equal to the right-hand side.
- !=, Not Equal to returns true if the left-hand side is not equal to the right-hand side.
- <, less than: returns true if the left-hand side is less than the right-hand side.
- <=, less than or equal to returns true if the left-hand side is less than or equal to the right-hand side.
- >, Greater than: returns true if the left-hand side is greater than the right-hand side.
- >=, Greater than or equal to returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

```
// Java Program to implement Relational Operators
import java.io.*;

// Driver Class
class GFG {
  // main function
  public static void main(String[] args)
  {
    // Comparison operators
    int a = 10;
    int b = 3;
    int c = 5;

    System.out.println("a > b: " + (a > b));
    System.out.println("a < b: " + (a < b));
    System.out.println("a >= b: " + (a >= b));
    System.out.println("a <= b: " + (a <= b));
    System.out.println("a == c: " + (a == c));
    System.out.println("a != c: " + (a != c));
  }
}
```

Output
a > b: true
a < b: false
a >= b: true
a <= b: false
a == c: false
a != c: true

5. Logical Operators

These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

Conditional operators are:

- &&, Logical AND: returns true when both conditions are true.
- ||, Logical OR: returns true if at least one condition is true.
- !, Logical NOT: returns true when a condition is false and vice-versa

Example:

```
// Java Program to implement Logical operators
```

```
import java.io.*;

// Driver Class
class GFG {
    // Main Function
   public static void main (String[] args) {
      // Logical operators
      boolean x = true;
      boolean y = false;

      System.out.println("x && y: " + (x && y));
      System.out.println("x || y: " + (x || y));
      System.out.println("!x: " + (!x));
   }
}
```

Output
x && y: false
x || y: true
!x: false

6. Ternary operator

The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.
The general format is:
condition ? if true : if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.
Example:

```
// Java program to illustrate max of three numbers using ternary operator.
public class operators {
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 30, result;
       result  = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
      System.out.println("Max of three numbers = "  + result);
   }
}
```

Output
Max of three numbers = 30

7. Bitwise Operators

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

- &, Bitwise AND operator: returns bit by bit AND of input values.
- |, Bitwise OR operator: returns bit by bit OR of input values.
- ^, Bitwise XOR operator: returns bit-by-bit XOR of input values.
- ~, Bitwise Complement Operator: This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

```
// Java Program to implement bitwise operators
import java.io.*;

// Driver class
class GFG {
   // main function
   public static void main(String[] args)
   {
      // Bitwise operators
      int d = 0b1010;
      int e = 0b1100;
      System.out.println("d & e: " + (d & e));
      System.out.println("d | e: " + (d | e));
      System.out.println("d ^ e: " + (d ^ e));
      System.out.println("~d: " + (~d));
      System.out.println("d << 2: " + (d << 2));
      System.out.println("e >> 1: " + (e >> 1));
      System.out.println("e >>> 1: " + (e >>> 1));
   }
}
```

Output

```
d & e: 8
d | e: 14
d ^ e: 6
~d: -11
d << 2: 40
e >> 1: 6
e >>> 1: 6
```

8. Shift Operators

These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. General format-

number shift_op number_of_places_to_shift;

- <<, Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
- >>, Signed Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of the initial number. Similar effect to dividing the number with some power of two.
- >>>, Unsigned Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

```
// Java Program to implement shift operators
import java.io.*;

// Driver Class
class GFG {
  // main function
  public static void main(String[] args)
  {
    int a = 10;

     // using left shift
    System.out.println("a<<1 : " + (a << 1));

    // using right shift
    System.out.println("a>>1 : " + (a >> 1));
  }
}
```

Output
a<<1 : 20
a>>1 : 5

Precedence and Associativity of Java Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first, as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude, with the top representing the highest precedence and the bottom showing the lowest precedence.

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Some more example programs on Java operators:

Java program to compute modulus

```
class modl
{
        public static void main(String[] any)
        {
                int i=31;
                double d=31.2;
                System.out.println(i%15);
                System.out.println(d%15);
                System.out.println(-7%4);
        }
}
```

Java program to demonstrate Bitwise operators

```
class Bits
{
        public static void main(String[] any)
        {
                int a=3,b=6;
                int c=a|b;
                int d=a&b;
                int e=a^b;
                int f=(~a&b) |(a&~b);
                int g=~a&0x0f;
                System.out.println(c);
                System.out.println(d);
```

```
            System.out.println(e);
            System.out.println(f);
            System.out.println(g);
        }
}
```

Java program to perform left shifting

```
class BitLeft
{

        public static void main(String[] any)
        {
                byte b=8;
                b=(byte)(b<<2);

                System.out.println(b);
        }
}
```

Java program to perform right shifting

```
class BithShift
{
        public static void main(String[] any)
        {

                int i=-1;
                int res1=i>>24;
                int res2=i>>>24;
                System.out.println(res1);
                System.out.println(res2);
        }
}
```

Java Short Circuit operators

In Java, short circuit operators are used to enhance the efficiency and performance of logical operations. These operators consist of". They are also known as "conditional logical operators" due to the fact they use conditional statements to determine whether to evaluate a selected expression or now not. The "&&" operator is used for "AND" operations, whilst" operator is used for "OR" operations. When used in a logical operation, these operators can appreciably improve the efficiency of the code.

A short circuit operator is a logical operator that lets in a programmer to improve the performance of a program. When a short circuit operator is used, the evaluation of an expression stops as quickly because the result of the operation is understood. For instance, think we have an expression like this: "A && B". In this situation, if "A" is fake, then the complete expression is fake, irrespective of the value of "B". Therefore, there's no want to evaluate the cost of "B" due to the fact the very last result of the expression is already acknowledged. This is in which the quick circuit operator comes into play. It stops comparing the expression as soon as it reveals out the result. Similarly, don't forget an expression like "A this example, if "A" is actual, then the whole expression is true, irrespective of the value of "B". Therefore, there may be no want to evaluate the fee of "B" due to the fact the final end result of the expression is already regarded.

Java's "&&" Operator

The "&&" operator is used for "AND" operations. It tests if each the left-hand side and the proper-hand side of the operator are authentic. If both operands are authentic, then the expression is genuine. Otherwise, the expression is fake. One of the primary blessings of the use of the "&&" operator is that it is able to prevent the assessment of pointless code. For example, if we've an expression like this: "A && B", and "A" is false, then there's no need to assess "B". This can improve the efficiency of the program and keep execution time.

Java's "||" Operator

The "||" operator is used for "OR" operations. It checks if either the left-hand side or the right-hand side of the operator is true. If either operand is true, then the expression is true. Otherwise, the expression is false. Like the "&&" operator, the "||" operator can also prevent the evaluation of unnecessary code. For example, if we have an expression like this: "A || B", and "A" is true, then there is no need to evaluate "B". This can improve the efficiency of the program and save execution time.

Decision making in Java

Decision making means the action of taking decisions and choosing the action plan accordingly. Consider a real life example, you are driving a car on a road to a concert, and you see your fuel light flashing, you immediately decide that you will stop at the next petrol pump. But if the light would not have blinked at all, you would have continued driving. Right? That is where you took a decision and planned your action.

Decision making in Java executes a particular segment of code based on the result of a boolean condition.It is important because conditions define the flow of programs and the output of a particular program.The decision making principles in Java chiefly consist of if else statements, continue, break and switch statements. It decides the flow of the program control during the execution of the program.

There are the 6 ways of exercising decision making in Java:

1. if
2. if-else
3. nested-if
4. if-else-if
5. switch-case
6. jump-break,continue,return

1. If Statement in Java

Java if statement is the simplest decision making statement. It encompasses a boolean condition followed by a scope of code which is executed only when the condition evaluates to true. However if there are no curly braces to limit the scope of sentences to be executed if the condition evaluates to true, then only the first line is executed.

Syntax:

```
if(condition)
{
//code to be executed
}
```

2. if else statement in Java

This pair of keywords is used to divide a program to be executed into two parts, one being the code to be executed if the condition evaluates to true and the other one to be executed if the value is false. However if no curly braces are given the first statement after the if or else keyword is executed.

```
if(condition)
{
//code to be executed if the condition is true
}
else
{
//code to be executed if the condition is false
}
```

3. Nested if Statements in Java

If the condition of the outer if statement evaluates to true then the inner if statement is evaluated. Nested if's are important if we have to declare extended conditions to a previous condition

Syntax:

```
if(condition)
{
//code to be executed
if(condition)
{
//code to be executed
}
}
```

4. if-else-if Statements in Java

These statements are similar to the if else statements. The only difference lies in the fact that each of the else statements can be paired with a different if condition statement. This renders the ladder as a multiple-choice option for the user. As soon as one of the if conditions evaluates to true the equivalent code is executed and the rest of the ladder is ignored.

Syntax:

```
if
{
//code to be executed
}
else if(condition)
{
//code to be executed
}
else if(condition)
{
//code to be executed
}
else
```

```
{
//code to be executed
}
```

5. Switch Statement in Java

Java switch statement is a different form of the if else if ladder statements.

- This saves the hassle of writing else if for every different option.
- It branches the flow of the program to multiple points as and when required or specified by the conditions.
- It bases the flow of the program based on the output of the expression.
- As soon as the expression is evaluated, the result is matched with each and every case listed in the scope. If the output matches with any of the cases mentioned, then the particular block is executed. A break statement is written after every end of the case block so that the remaining statements are not executed.
- The default case is written which is executed if none of the cases are the result of the expression. This is generally the block where error statements are written.

```
switch(expression)
{
case <value1>:
//code to be executed
break;
case <value2>:
//code to be executed
break;
default:
//code to be defaultly executed
}
```

6. Jump Statements in Java

The jump statements are the keywords in Java which have specific meaning and specific action of disrupting the normal flow of the program. Some of them are:

a. Java break statement

The break statement , as the name signifies, is useful to break out of the normal workflow of a program. We can use it to terminate loops, end cases of switch statements, and as a goto statement. Break statements, if used in a nested loop, terminates the innermost loop. The outermost loop still functions. break statement as a goto statement. If we mention a label after the break keyword, the control shifts to the end of the particular label scope. Java prohibits the use of goto statements because it encourages branched behaviour of programming. However, break labels function like goto statements.

b. Continue Statement in Java

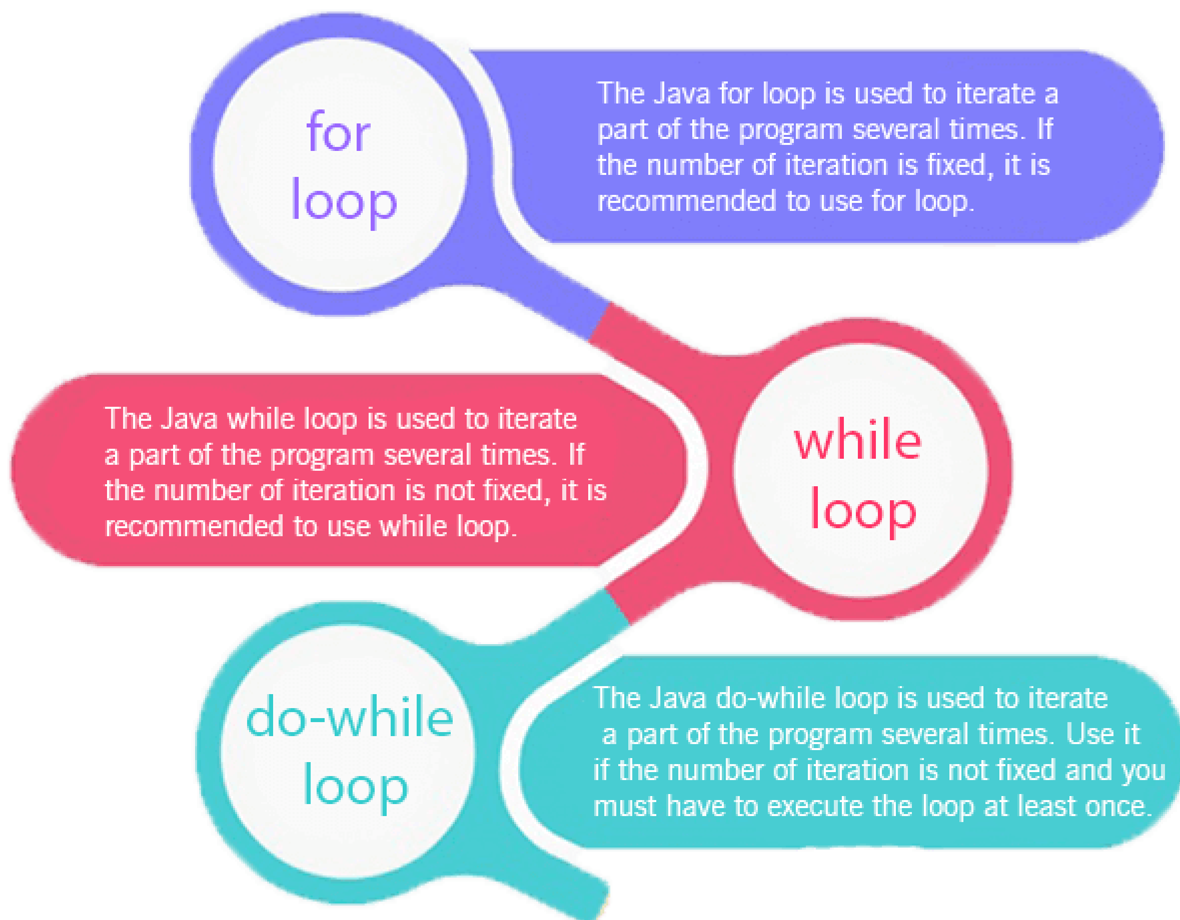
The continue statement is useful for early iteration of a particular loop. Sometimes rather than breaking out of a loop and halting it for good we want to skip some iterations based on our requirements of a program. This results in the usage of continue statements.

c. Return Statement in Java

This keyword, when executed by the compiler, returns the control back to the method it was called from. This is generally used in methods which are not of the void type and return some values. When the statement is executed, the return function returns to the caller method along with whatever variable is mentioned in the definition.

for loop in Java

The Java *for loop* is used to iterate a part of the program several times. If the number of iterations is fixed, it is recommended to use for loop. There are three types of for loops in Java.



- Simple for Loop
- For-each or Enhanced for Loop
- Labeled for Loop
-

Java Simple for Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. Increment/Decrement: It increments or decrements the variable value. It is an optional condition.
4. Statement: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

Java Nested for Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Java for-each Loop

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on the basis of elements and not the index. It returns element one by one in the defined variable.

Syntax:

```
for(data_type variable : array_name){
//code to be executed
}
```

Java Infinitive for Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Java for Loop vs while Loop vs do-while Loop

Comparison	for loop	while loop	do-while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.

When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	for(init;condition;incr/decr){ // code to be executed }	while(condition){ //code to be executed }	do{ //code to be executed }while(condition);
Example	//for loop for(int i=1;i<=10;i++){ System.out.println(i); }	//while loop int i=1; while(i<=10){ System.out.println(i); i++; }	//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);
Syntax for infinitive loop	for(;;){ //code to be executed }	while(true){ //code to be executed }	do{ //code to be executed }while(true);

while loop in Java

The Java *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops. The while loop is considered as a repeating if statement. If the number of iterations is not fixed, it is recommended to use the while loop.

Syntax:

```
while (condition){
//code to be executed
Increment / decrement statement
}
```

do-while loop in Java

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop. Java do-while loop is called an exit control loop. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

Labelled break and continue in Java

In Java, we can label the loops and give them names. These named or labeled loops help in the case of *nested loops* when we want to break or continue a specific loop out of those multiple nested loops. The labeled blocks in Java are *logically* similar to goto statements in C/C++.

Syntax

A label is any valid identifier followed by a colon. For example, in the following code, we are creating two labeled statements:

```
outer_loop:
for (int i = 0; i < array.length; i++) {

 inner_loop:
 for (int j = 0; j < array.length; j++) {

  //...
 }

 //...
}
```

In the above example, we have two loops, and we have labeled them as *outer_loop* and *inner_loop*. This is helpful when we want to terminate the outer loop from a condition written in the inner loop.

Difference between Simple *break* and Labeled *break*

The simple *break* statement in Java terminates only the immediate loop in which it is specified. So even if we break from the inner loop, it will still continue to execute the current iteration of the outer loop. We must use the labeled break statement to terminate a specific loop, as the *outer_loop* in the above example. In the same way, we can use the labeled *continue* statements to jump to the next iteration of any specific loop in the hierarchy of nested loops.

```
continue outer_loop;
```

Some example Java programs illustrating loop behavior

Java program to count down from 10 to 1

```
class DownCounter
{
	public static void main(String args[])
	{
		for(int i=10;i>=1;i--)
			System.out.println(i);
		System.out.println("Happy New Year");
	}
}
```

Java program to illustrate pyramid patterns

```
class Pattern
{
```

```
        public static void main(String args[])
        {
                int n=5;
                for(int i=0;i<n;i++)
                {
                        for(int j=0;j<=i;j++)
                        {
                                System.out.print("* ");
                        }
                        System.out.println();
                }
        }
}
```

Java program for array traversal using enhanced for without indexing

```
class ForArrays
{

        public static void main(String args[])
        {

             int[] a={12,11,9,10,14};
            for(int num:a)
                    System.out.println(num);

        }

}
```

Java program to illustrate labelled break and continue

```
class BreakCont
{
        public static void main(String[] aiml)
        {
            l1:for(int j=0;j<4;j++)
                {
                for(int i=0;i<10;i++)
                {
                        if(i%2==1)
                           break l1;
                   System.out.println("AIML003");
                 }
                }
        }
}
```

Java program to illustrate loops with local variable type inferencing

```
class LoopVar
{
        public static void main(String any[])
        {
                for(var v=2.5;v<10;v++)
                        System.out.println(v*2);
```

```
       }
}
```

Java program to check whether number is prime or not

```
import java.util.*;
class Prime
{
     public static void main(String any[])
     {
            System.out.println("Enter a number");
         Scanner kb=new Scanner(System.in);
         int num=kb.nextInt();
         for(int i=2;i<=num-1;i++)
         {
               if(num%i==0)
            {
                 System.out.println("Non prime");
                 System.exit(0);
          }
            }
         System.out.println("Prime");
     }

}
```

LAB COMPONENT

1. Develop a JAVA program to add TWO matrices of suitable order N (The value of N should be read from command line arguments).

```
import java.util.*;
class Matrix
{
     public static void readMatrix(int[][] A,int N)
     {
          Scanner kb=new Scanner(System.in);
           for(int i=0;i<=N-1;i++)
        {
            for(int j=0;j<=N-1;j++)
            {
                      A[i][j]=kb.nextInt();
               }
          }
     }
     public static void addMatrix(int[][] A, int[][] B, int[][] C,int
N)
     {
          for(int i=0;i<=N-1;i++)
          {
```

```
                for(int j=0;j<=N-1;j++)
                {
                     C[i][j]=A[i][j]+B[i][j];
                }
           }
      }
      public static void printMatrix(int[][] A,int N)
      {
           for(int i=0;i<=N-1;i++)
           {
                for(int j=0;j<=N-1;j++)
                {
                     System.out.print(A[i][j]+" ");
                }
                System.out.println();
           }
      }

      public static void main(String[] any)
      {
            int N=Integer.parseInt(any[0]);
         int[][] A=new int[N][N];
         int B[][]=new int[N][N];
         int C[][]=new int[N][N];
         System.out.println("Enter Matrix A");
         readMatrix(A,N);
         System.out.println("Enter Matrix B");
         readMatrix(B,N);
         addMatrix(A,B,C,N);
         System.out.println("Sum Matrix C");
         printMatrix(C,N);
      }
}
```