Programming in Java

U23CA404

Unit I

Introduction: Review of Object Oriented Concepts-History of Java-Java buzzwords-JVM Architecture- Data Types-Variables-Scope and Lifetime of Variables-arrays-Operators-Control Statements-Type Conversion and Casting-simple Java Program-Constructors-methods-Static block-Static Data-Static Method-String and String Buffer Classes

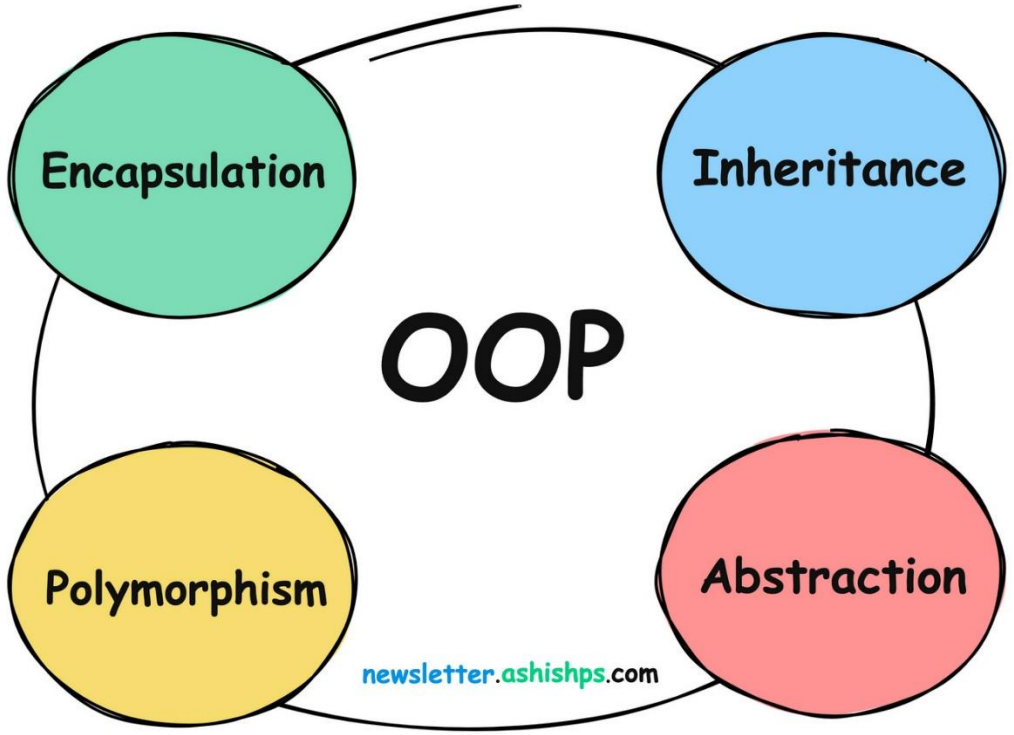
Introduction

- Java is a **programming language** and a **platform**.
- Java is a high level, robust, object-oriented and secure programming language.
- Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.
- The primary objective of Java programming language creation was to make it portable, simple and secure programming language.

Review of Object Oriented Concepts

Encapsulation:

Binding (or wrapping) code and data together into a single unit is known as encapsulation.



Inheritance:

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

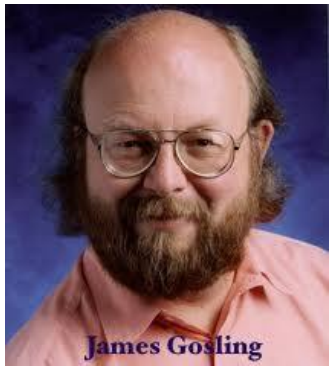
Polymorphism:

When one task is performed by different ways is known as polymorphism.

Abstraction:

Hiding internal details and showing functionality is known as abstraction.

History of Java



1978

C language –Structured language –By Dennis Ritchie in 1978 – Difficult to tackle complex problems

1979

C++ - Object oriented Programming- BY Bjarne Stroustrup in 1979.

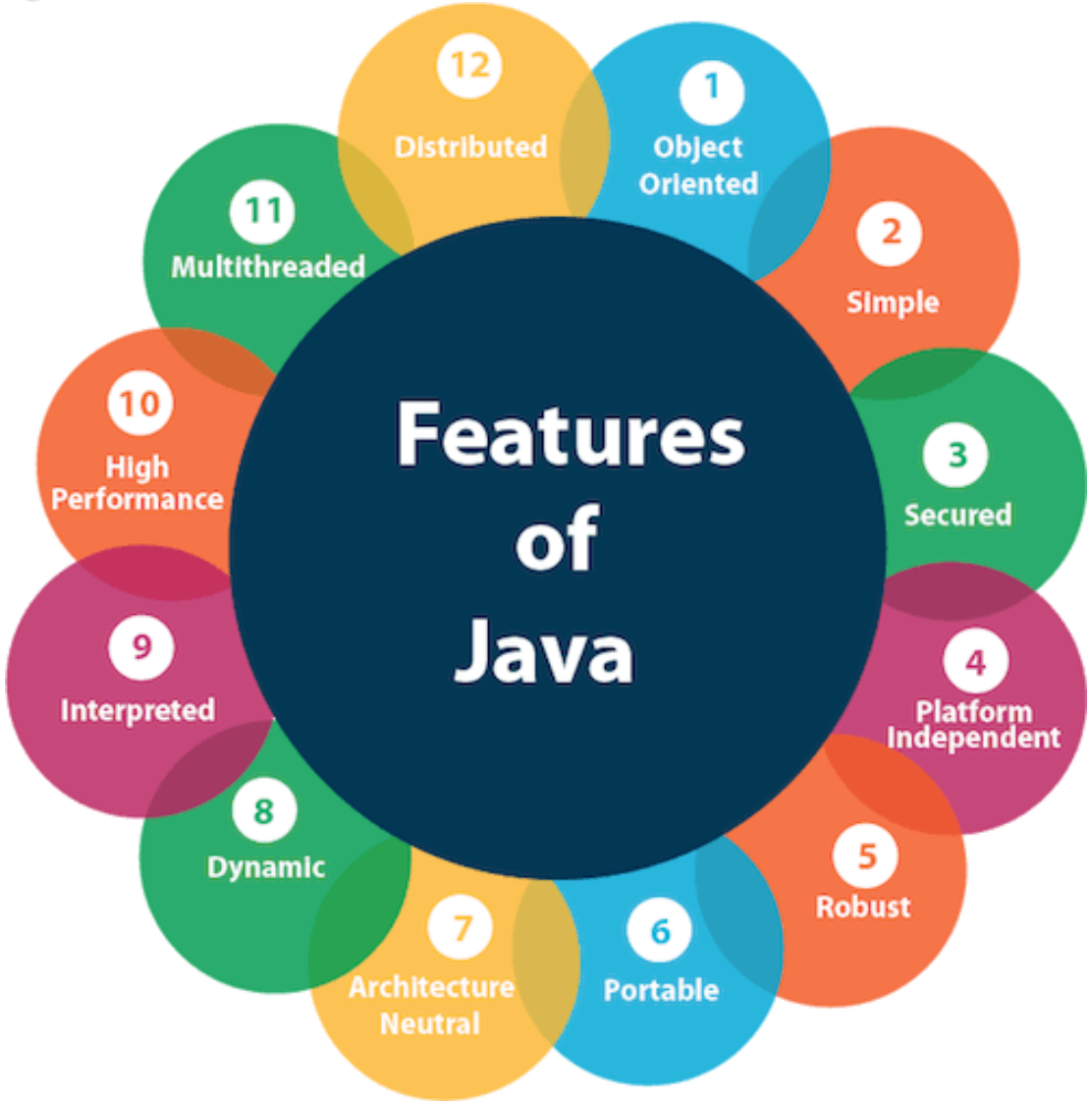
1991

OAK - James Gosling and Patrick Naughton in 1991 at Sun Microsystems - simple and platform independent.

1995

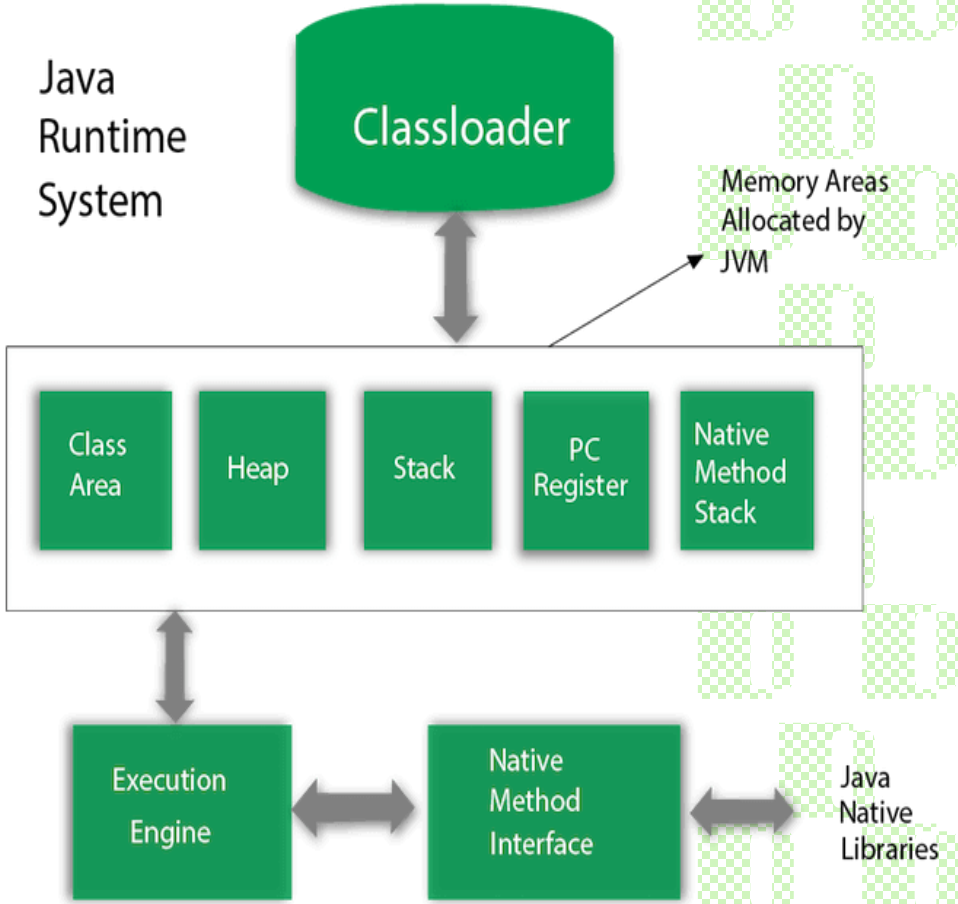
In 1995 renamed as JAVA

Java Buzzwords/Features



JVM Architecture

- JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.
- It is:
  - A specification where working of Java Virtual Machine is specified.
  - An implementation Its implementation is known as JRE (Java Runtime Environment).
  - Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.
- The JVM performs following operation:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment



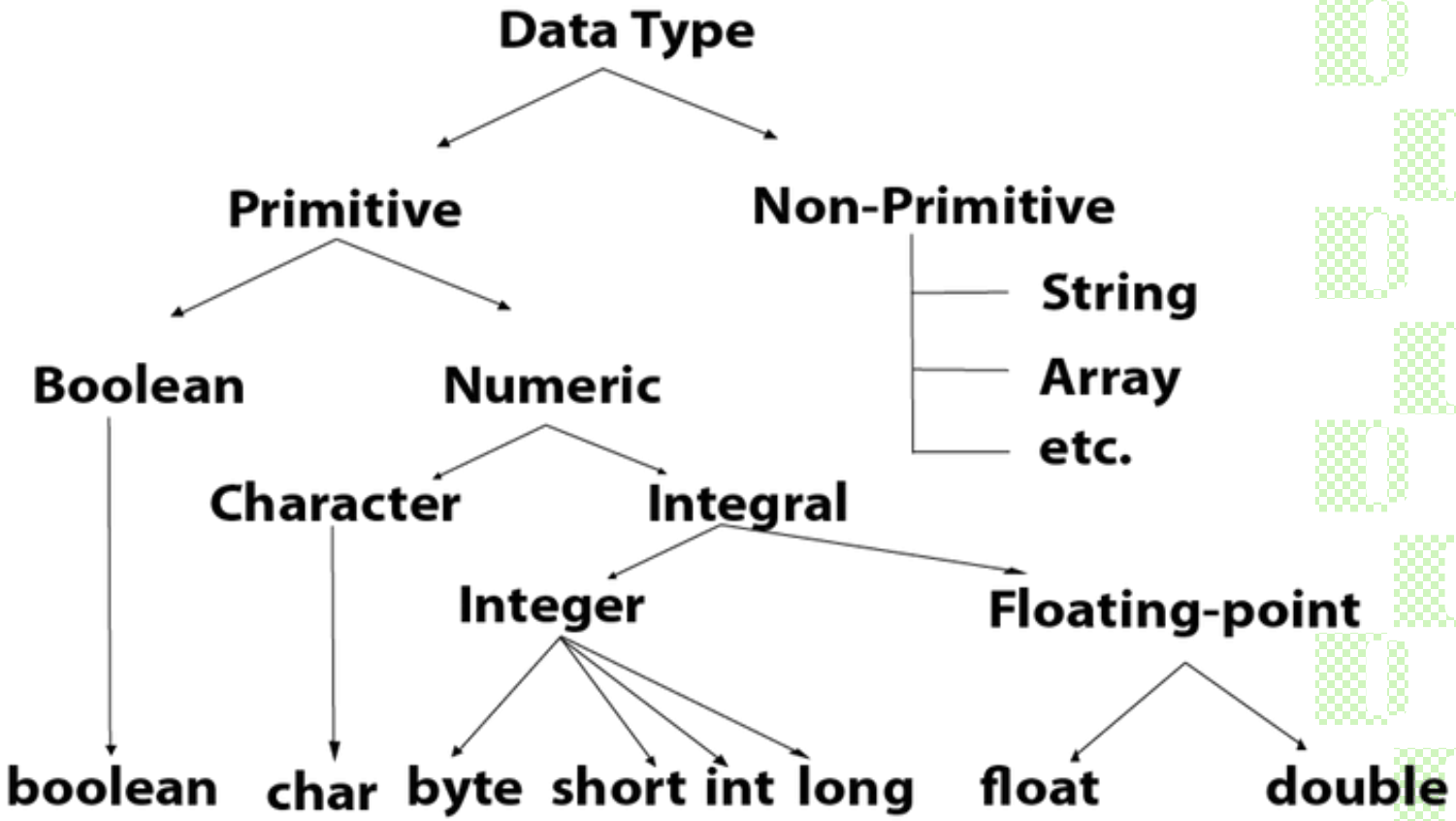
- Classloader:
  - Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.
    - Bootstrap ClassLoader:
    - Extension ClassLoader:
    - System/Application ClassLoader:
- Class Area:
  - Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
- Heap:
  - It is the runtime data area in which objects are allocated.
- Stack:
  - Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.
  - Each thread has a private JVM stack, created at the same time as thread.
  - A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

- PC Register:
  - PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.
- Native Method Stack:
  - It contains all the native methods used in the application.
- Execution Engine:
  - A virtual processor
  - Interpreter: Read bytecode stream then execute the instructions.
  - Just-In-Time(JIT) compiler: It is used to improve the performance.
- Java Native Libraries:
  - Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.
  - Java uses JNI framework to send output to the Console or interact with OS libraries.

Working of JVM

Data Types

- Data types represent the different values to be stored in the variable.
- In java, there are two types of data types:
  - Primitive datatypes
  - Non-primitive datatypes



Data Type	Size	DefaultValue	Description
byte	1 byte	0	Stores whole numbers from -128 to 127
short	2 bytes	0	Stores whole numbers from -32,768 to 32,767
int	4 bytes	0	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	0L	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	0.0f	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	0.0d	Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits
boolean	1 bit	False	Stores true or false values
char	2 bytes	‘\u000’	Stores a single character/letter or ASCII values

Variables

- Variable is a name of memory location. There are three types of variables in java:
  - local - A variable which is declared inside the method is called local variable.
  - instance - A variable which is declared inside the class but outside the method, is called instance variable. It is not declared as static.
  - static - A variable that is declared as static is called static variable. It cannot be local. We will have detailed learning of these variables in next chapters.
- **Example to understand the types of variables in java**

```
class A
{
int data=50;//instance variable
static int m=100;//static variable
void method()
{
int n=90;//local variable
}
}//end of class
```

Constants in Java

- A constant is a variable which cannot have its value changed after declaration. It uses the **'final' keyword.**
- **Syntax**
- modifier**final** dataType variableName = value; *//global constant*
- modifier**static final** dataType variableName = value; *//constant within a c*

Scope and Lifetime of Variables

- The scope of a variable defines the section of the code in which the variable is visible.
- The lifetime of a variable refers to how long the variable exists before it is destroyed.
- Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it.
- The three types of variables are:
  - Instance Variables
  - Class Variables
  - Local Variables

Instance Variables

- A variable which is declared inside a class and outside all the methods and blocks is an instance variable.
- General **scope of an instance variable** is throughout the class except in static methods.
- **Lifetime of an instance variable** is until the object stays in memory.

```
class Sample
{
    int x, y;   //instance variables
    static int result;
    void add(int a, int b)  //a and b are local variables
    {
            x = a;
            y = b;
            int sum = x+y;   //Sum
            System.out.println("Sum = "+sum);
    }
    public static void main(String[] args)
    {
            Sample obj = new Sample();
            obj.add(10,20);
    }
}
```

Scope of x and y

Class Variables

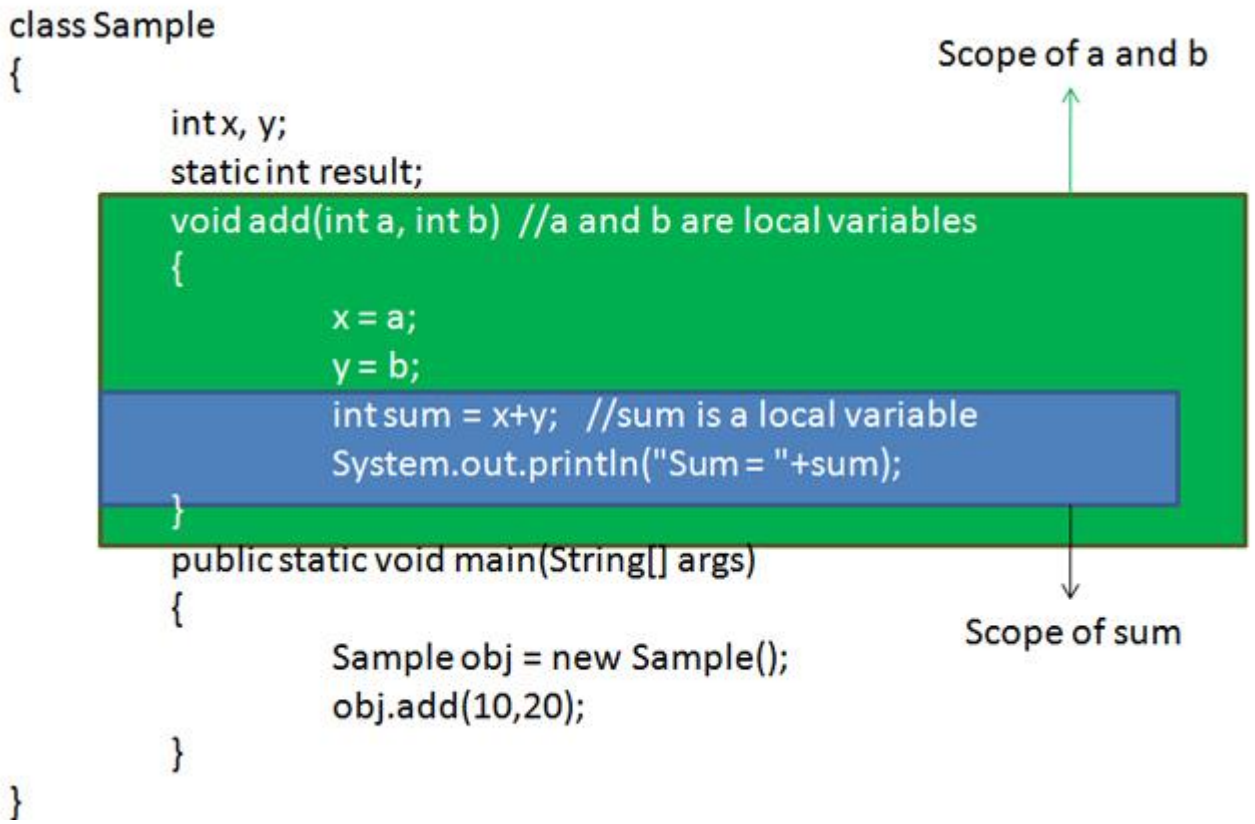
- A variable which is declared inside a class, outside all the blocks and is marked *static* is known as a class variable.
- General **scope of a class variable** is throughout the class.
- **Lifetime of a class variable** is until the end of the program or as long as the class is loaded in memory.

```
class Sample
{
        int x, y;
        static int result;  //Class variable
        void add(int a, int b)
        {
                x = a;
                y = b;
                int sum = x+y;
                System.out.println("Sum = "+sum);
        }
        public static void main(String[] args)
        {
                Sample obj = new Sample();
                obj.add(10,20);
        }
}
```

Scope of result

Local Variables

- All other variables which are not instance and class variables are treated as local variables including the parameters in a method.
- **Scope of a local variable** is within the block in which it is declared.
- **Lifetime of a local variable** is until the control leaves the block in which it is declared.



Summary of scope and lifetime of variables

Variable Type	Scope	Lifetime
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Arrays

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- To declare an array, define the variable type with square brackets:

```
String[] cars;
```

- To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

- To access an array element, we can refer to the index number.

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
```

Output: Volvo

- To change the value of a specific element, refer to the index number:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
```

Output: Opel

- To find out how many elements an array has, use the length property:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars.length);
```

Types of Arrays

Arrays

- Single Dimensional
  - Syntax:
    datatype[] arr;/
    datatype []arr;/
    datatype arr[];

```
class Testarray1
{
public static void main(String args[])
{
        int a[]={33,3,4,5};
        for(int i=0;i<a.length;i++)
        System.out.println(a[i]);
}
}
```

Output:
33
3
4
5

- Multi Dimensional
  - Syntax:
    dataType[][] arr;/
    dataType [][]arr;/
    dataType arr[][];/
    dataType []arr[];

```
class Testarray3
{
public static void main(String args[])
{
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}}}
```

Output:
1 2 3
2 4 5
4 4 5

```
class Testarray3
{
public static void main(String args[])
{
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
for(int i=0;i<3;i++)    // row iteration
{
        for(int j=0;j<3;j++)        //
       Column iteration
       {
         System.out.print(arr[i][j]+" ");
       }
 System.out.println();
}
}
}
```

Explanation:

	j=0	j=1	j=2
i=0	1	2	3
i= 1	2	4	5
i=2	4	4	5

i=0,0<3 –T → j=0, 0<3-T → arr[0][0] →1
j=1, 1<3-T → arr[0][1] → 2
j=2, 2<3-T → arr[0][2] →3
j=3, 3<3 –F exit the j Loop
i=1,1<3-T →j=0, 0<3-t → arr[1][0] → 2
j=1, 1<3-T → arr[1][1] → 4
j=2, 2<3-T → arr[1][2] →5
j=3, 3<3 –F exit the j Loop
i=2,2<3-T →j=0,0<3-T --. arr[2][0] →4
j=1, 1<3-T → arr[2][1] → 4
j=2, 2<3-T → arr[2][2] →5
j=3, 3<3 –F exit the j Loop
i=3,3<3-F

Operators in Java

- Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.
- There are many types of operators in Java which are given below:
  - Unary Operator
  - Arithmetic Operator
  - Shift Operator
  - Relational Operator
  - Bitwise Operator
  - Logical Operator
  - Ternary Operator
  - Assignment Operator

Operators in Java - Unary

Operator	Description	Example Program	Output

Unary Operator
++, --, ~, !
	It requires only one operand.
	Operations:
	1. incrementing/decrementing a value by one
	2. Negating an expression
	3. Inverting the value of a boolean
		public class UnaryOperator
		{
		public static void main(String args[])
		{
		int x=10;
		boolean y=true;
		System.out.println(x++); //prints 10, then the value is incremented to 11
		System.out.println(++x); //prints 12, as the value of x is 11
		System.out.println(x--); //prints 12 and decrements to 11
		System.out.println(--x); //prints 10, as the value of x is 11
		System.out.println(~x); //minus of total positive Value starts from 0
		Sysyem.out.println(!y); //opposite of Boolean value
		}
		}
			10
			12
			12
			10
			-11
			false

Operators in Java - Arithmetic

Operator	Description	Example Program	Output

Arithmetic Operator	1. Java arithmetic operators are used to perform addition, subtraction, multiplication, and division.
	2. They act as basic mathematical operations.
		public class ArithmeticOperator
		{
		public static void main(String args[])
		{
		int a=10;
		int b=5;
		System.out.println(a+b);
		System.out.println(a-b);
		System.out.println(a*b);
		System.out.println(a/b);
		System.out.println(a%b);
		System.out.println(10*10/5+3-1*4/2);
		}
		}
			15
			5
			50
			2
			0
			21

Operators in Java - Arithmetic

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Operators in Java - Shift

Operator	Description	Example Program	Output

Left Shift Operator <<	1. 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.	public class LeftShiftOperator
		{
		public static void main(String args[])
		{
		System.out.println(10<<2);
		System.out.println(10<<3);
		System.out.println(20<<2);
		}
		}
			40
			80
			80

Right Shift Operator >>	1. 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.	public class RightShiftOperator
		{
		public static void main(String args[])
		{
		System.out.println(10>>2);
		System.out.println(20>>2);
		System.out.println(20>>3);
		}
		}
			2
			5
			2

Operators in Java – Logical && and Bitwise &

Operator	Description	Example Program	Output

Logical AND &&	1. The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.	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

Bitwise AND &	1. The bitwise & operator always checks both conditions whether 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

Operators in Java – Logical || and Bitwise |

Operator	Description	Example Program	Output

Logical OR
||
	1. 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	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

Bitwise OR
|
	1. The bitwise | operator always checks both conditions whether 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

Operators in Java - Logical

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Operators in Java – Ternary and Assignment

Operator	Description	Example Program	Output

Ternary operator
?:
	1. Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.	public class OperatorExample
		{
		public static void main(String args[])
		{
		int a=2;
		int b=5;
		int min=(a<b)?a:b;
		System.out.println(min);
		}
		}
			2

Assignment operator
=
	1. 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.	public class OperatorExample
		{
		public static void main(String args[])
		{
		int a=10;
		int b=20;
		a+=4;
		b-=4;
		System.out.println(a);
		System.out.println(b);
		}
		}
			14
			16

Operators in Java - Assignment

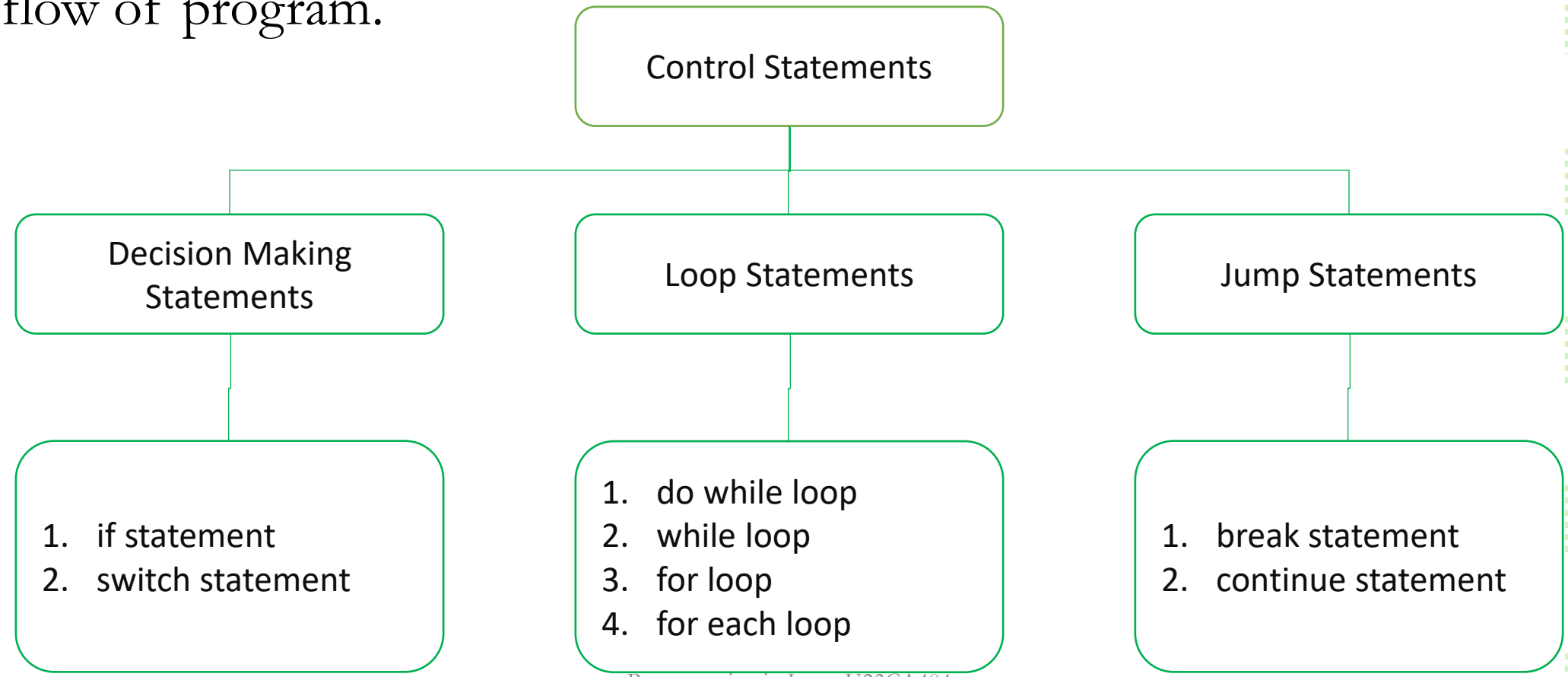
Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Operators in Java - Relational

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Control Statements

- Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements.
- It is one of the fundamental features of Java, which provides a smooth flow of program.



1. Decision Making Statements

- Decision-making statements decide which statement to execute and when.
- Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided.
- There are two types of decision-making statements in Java.

1. if statement
2. switch statement.

1.1 if Statement

- In Java, the "if" statement is used to evaluate a condition.
- The control of the program is diverted depending upon the specific condition.
- The condition of the If statement gives a Boolean value, either true or false.
- In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

1.1.a Simple if Statement

- It is the most basic statement among all control flow statements in Java.
- It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax	Sample Program	Output

if(condition)
{
statement 1; //executes when co
ndition is true
}
	public class Student
	{
	public static void main(String[] args)
	{
	int x = 10;
	int y = 12;
	if(x+y > 20)
	{
	System.out.println("x + y is greater than 20");
	}
	}
	}
		x+y is greater than 20

1.1.b if-else Statement

- The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block.
- The else block is executed if the condition of the if-block is evaluated as false.

Syntax	Sample Program	Output

if(condition)
{
statement 1; //executes when condition is true
}
else
{
statement 2; //executes when condition is false
}
	public **class** Student
	{
	public static void main(String[] args)
	{
	int x = 10;
	int y = 12;
	if(x+y < 10)
	{
	System.out.println("x + y is less than 10");
	}
	else
	{
	System.out.println("x + y is greater than 20");
	}
	}
	}
		x+y is greater than 20

1.1.c if-else-if ladder Statement

- The if-else-if statement contains the if-statement followed by multiple else-if statements.
- In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true.
- We can also define an else statement at the end of the chain.

Syntax	Sample Program	Output

if(condition 1)
{
statement 1; //executes when condition 1 is true
}
else if(condition 2)
{
statement 2; //executes when condition 2 is true
}
else
{
statement 2; //executes when all the conditions are false
}
	public class Student
	{
	public static void main(String[] args) {
	String city = "Delhi";
	if(city == "Meerut") {
	System.out.println("city is meerut");
	}else if (city == "Noida") {
	System.out.println("city is noida");
	}else if(city == "Agra") {
	System.out.println("city is agra");
	}else {
	System.out.println(city);
	}
	}
	}
		Delhi

1.1.d Nested if Statement

- In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax	Sample Program	Output

if(condition 1)
{
statement 1; //executes when condition 1 is true
if(condition 2)
{
statement 2; //executes when condition 2 is true
}
else
{
statement 2; //executes when condition 2 is false
}
}
	public class Student {
	public static void main(String[] args) {
	String address = "Delhi, India";
	if(address.endsWith("India")) {
	if(address.contains("Meerut")) {
	System.out.println("Your city is Meerut");
	}else if(address.contains("Noida")) {
	System.out.println("Your city is Noida");
	}else {
	System.out.println(address.split(",")[0]);
	}
	}else {
	System.out.println("You are not living in India");
	}
	}
	}
		Delhi

1.2 switch statement

- In Java, Switch statements are similar to if-else-if statements.
- The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.
- The switch statement is easier to use instead of if-else-if statements.
- It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

1.2 switch Statement

Syntax	Sample Program	Output

switch (expression)
{
case value1:
statement1;
break;
.
.
.
case valueN:
statementN;
break;
default:
default statement;
}
	public class Student implements Cloneable
	{
	public static void main(String[] args)
	{
	int num = 2;
	switch (num)
	{
	case 0:
	System.out.println("number is 0");
	break;
	case 1:
	System.out.println("number is 1");
	break;
	default:
	System.out.println(num);
	}
	}
	}
		2

Java User Input

- The Scanner class is used to get user input, and it is found in the java.util package.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.
- In our example, we will use the nextLine() method, which is used to read Strings.

```
import java.util.Scanner; // Import the Scanner class
class sample
{
public static void main(String[] args)
{
Scanner myObj = new Scanner(System.in); // Create a Scanner
object
System.out.println("Enter username");
String userName = myObj.nextLine(); // Read user input
System.out.println("Username is: " + userName); // Output
user input
}
}
```

Output:

Enter username
Ivan
Username is: Ivan

Input types

Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Example Program

```
import java.util.Scanner;
class sample
{
public static void main(String[] args)
{
        Scanner myObj = new Scanner(System.in);
        System.out.println("Enter name, age and salary:");
        String name = myObj.nextLine(); // String input
        int age = myObj.nextInt(); // Numerical input
        double salary = myObj.nextDouble();
        System.out.println("Name: " + name); // Output input by user
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
}
}
```

Output:

```
Enter name, age and salary:
Jagan
34
75000
Name: Jagan
Age: 34
Salary: 75000.00
```

Practical Programs: 1. Prime Number

```
import java.util.*;
public class PrimeNumbersUpToLimit
{
// Function to check if a number is prime
   public static boolean isPrime(int n) {
      if (n <= 1) {
         return false; // Not prime
      }
      for (int i = 2; i <= n/2 ; i++)
            {
            // Check divisibility up to √n
         if (n % i == 0) {
            return false;
         }
      }
      return true;
   }
```

```
public static void main(String[] args)
{
      Scanner scanner = new Scanner(System.in);
      // Input the limit
      System.out.print("Enter the limit: ");
      int limit = scanner.nextInt();
      System.out.println("Prime numbers up to " + limit + ":");
      // Loop through numbers from 2 to the limit
      for (int num = 2; num <= limit; num++) {
         if (isPrime(num)) {
            System.out.print(num + " ");
         }
      }
   }
}
```

Output:

Enter the limit:10

Prime numbers up to 10 :

2 3 5 7

Practical Programs: 2. Matrix Multiplication

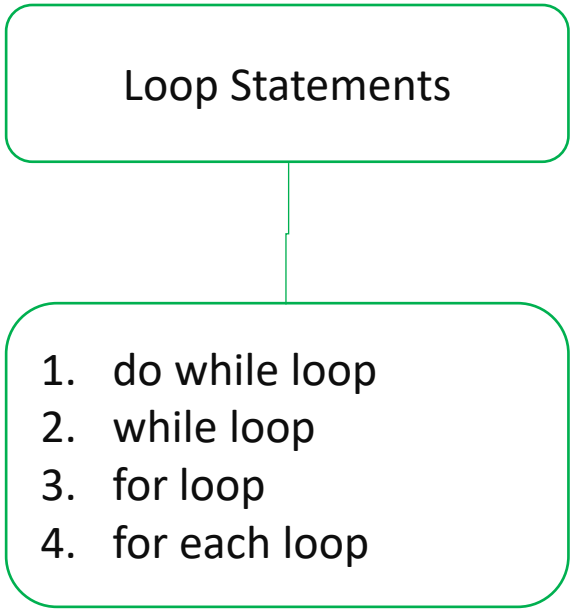
```
import java.util.Scanner;
public class MatrixMultiplication {
   public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
       System.out.print("Enter the number of rows in the first matrix: ");
       int rows1 = scanner.nextInt();
       System.out.print("Enter the number of columns in the first matrix: ");
       int cols1 = scanner.nextInt();
       System.out.print("Enter the number of rows in the second matrix: ");
       int rows2 = scanner.nextInt();
       System.out.print("Enter the number of columns in the second matrix: ");
       int cols2 = scanner.nextInt();
       if (cols1 != rows2) {
          System.out.println("Matrix multiplication not possible as the no. of columns
in the first matrix must equal the no. of rows in the second matrix.");
   return;
       }
System.out.println("Enter elements of the first matrix:");
       int[][] matrix1 = new int[rows1][cols1];
       for (int i = 0; i < rows1; i++) {
          for (int j = 0; j < cols1; j++) {
            matrix1[i][j] = scanner.nextInt();          }}
System.out.println("Enter elements of the second matrix:");
       int[][] matrix2 = new int[rows2][cols2];
       for (int i = 0; i < rows2; i++) {
          for (int j = 0; j < cols2; j++) {
            matrix2[i][j] = scanner.nextInt();
          }
}
int[][] result = new int[rows1][cols2];
       for (int i = 0; i < rows1; i++) {
          for (int j = 0; j < cols2; j++) {
             for (int k = 0; k < cols1; k++) {
result[i][j] += matrix1[i][k] * matrix2[k][j];
             }}}
System.out.println("The product of the matrices is:");
       for (int i = 0; i < rows1; i++) {
          for (int j = 0; j < cols2; j++) {
             System.out.print(result[i][j] + " ");
          }
          System.out.println();
  }
}
}
```

Practical Programs: 4. Random Numbers

```
import java.util.Scanner;
import java.util.Random;
public class RandomNumberGenerator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
System.out.print("Enter the lower limit: ");
        int lowerLimit = scanner.nextInt();
        System.out.print("Enter the upper limit: ");
        int upperLimit = scanner.nextInt();
int randomNumber = random.nextInt(upperLimit - lowerLimit + 1) +
lowerLimit;
        System.out.println("Generated Random Number: " +
randomNumber);
if (randomNumber < (lowerLimit + upperLimit) / 3) {
            System.out.println("The number is in the lower
range.");
        }
else if (randomNumber < 2 * (lowerLimit + upperLimit) /
3) {
            System.out.println("The number is in the middle
range.");
        }
else {
            System.out.println("The number is in the upper
range.");
        }
        scanner.close();
    }
}
```

2. Loop Statements

- Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.
- Java provides three ways for executing the loops.
- While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

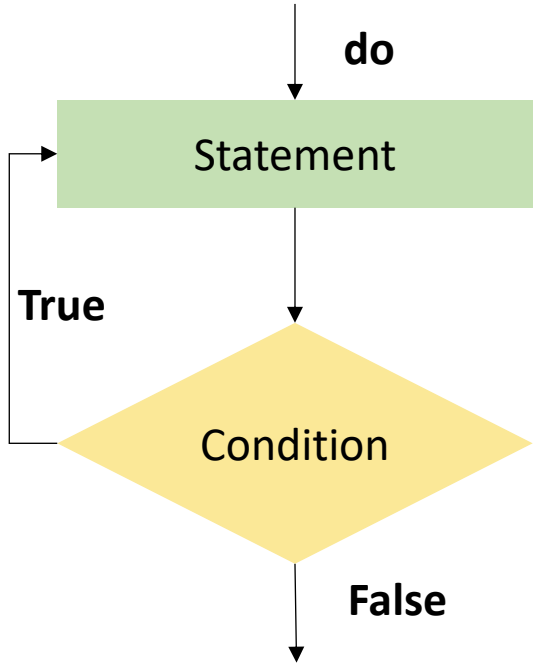


2.1 do while loop

- The Java do-while loop is used to iterate a part of the program repeatedly, until the specified condition is true.
- Java do-while loop is called an exit control loop.
- 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);
```



do while - program

```
public class DoWhileExample
{
public static void main(String[] args)
{
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Output:

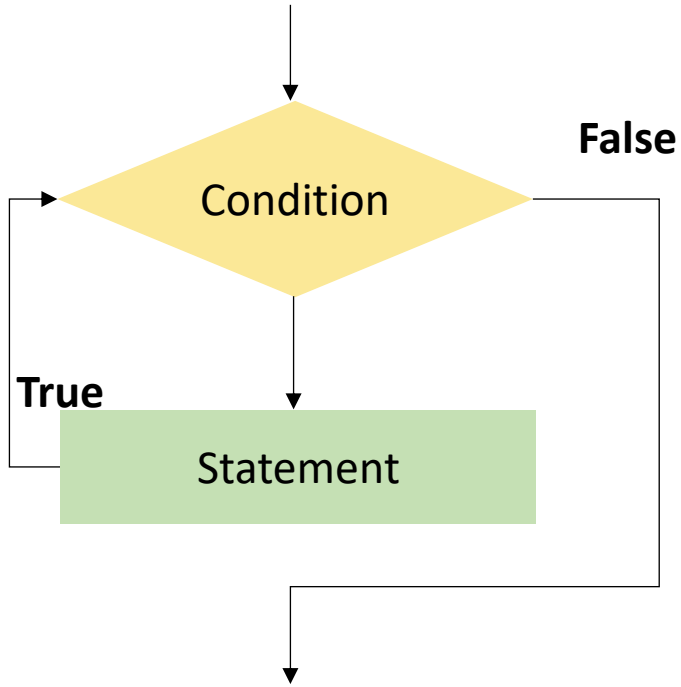
```
1
2
3
4
5
6
7
8
9
10
```

2.2 while loop

- 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.

Syntax:

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



while loop - program

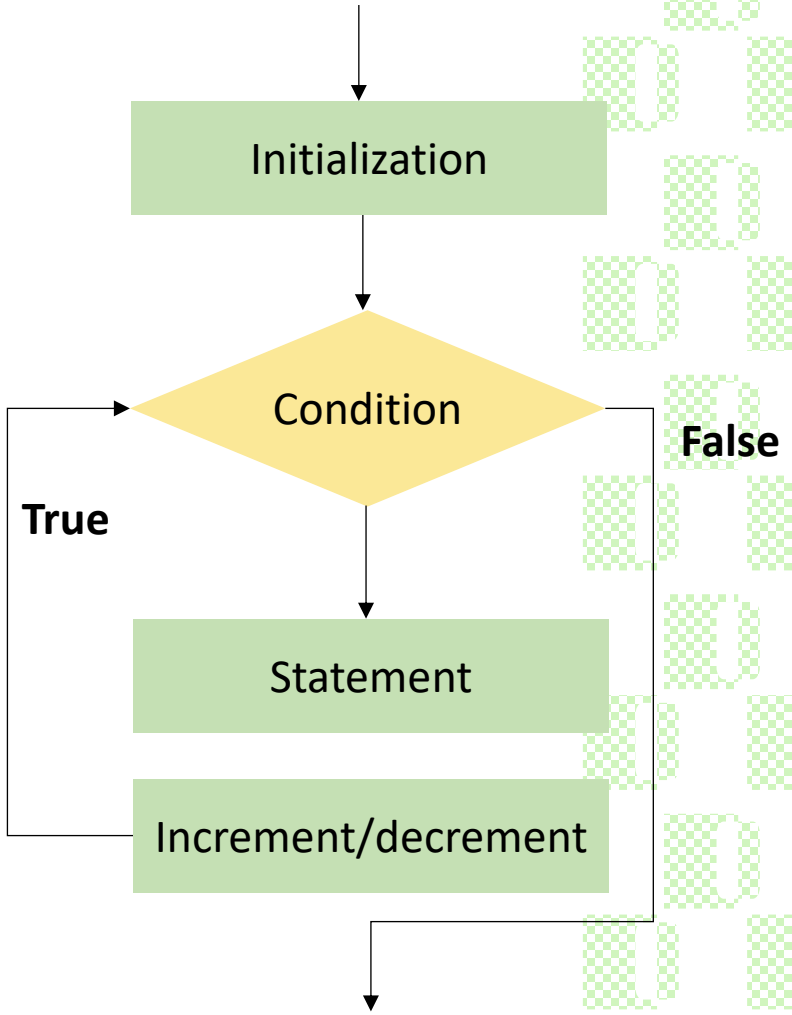
```
public class WhileExample
{
public static void main(String[] args)
{
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

2.3 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:
- **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.
- **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.
- **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.
- **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
}
```

for loop program

```
public class FactorialExample
{
        public static void main(String[] args)
        {
        int number = 5;
        int factorial = 1;
        for(int i=1;i<=number;i++)
        {
                factorial*=i;
        }
        System.out.println("Factorial of"+number+"is:"+factorial);
        }
}
```

Output:

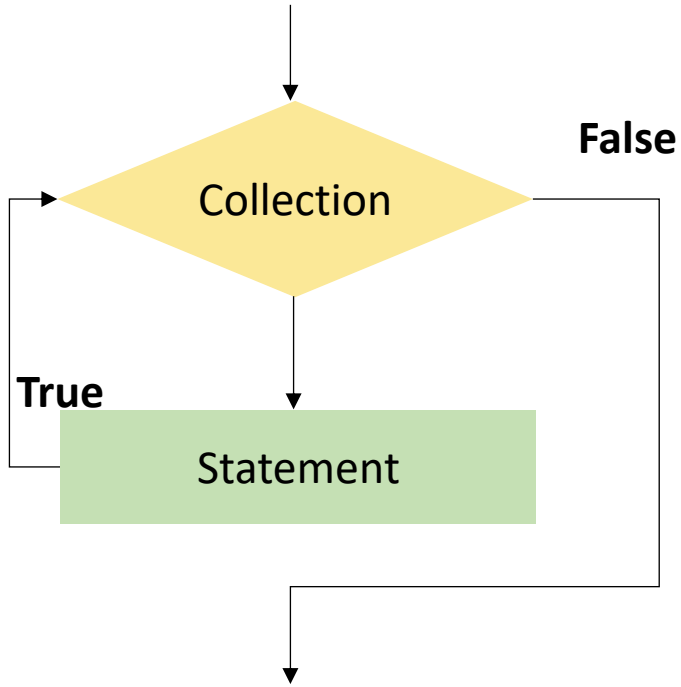
Factorial of 5 is: 120

2.4 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
}
```



for each program

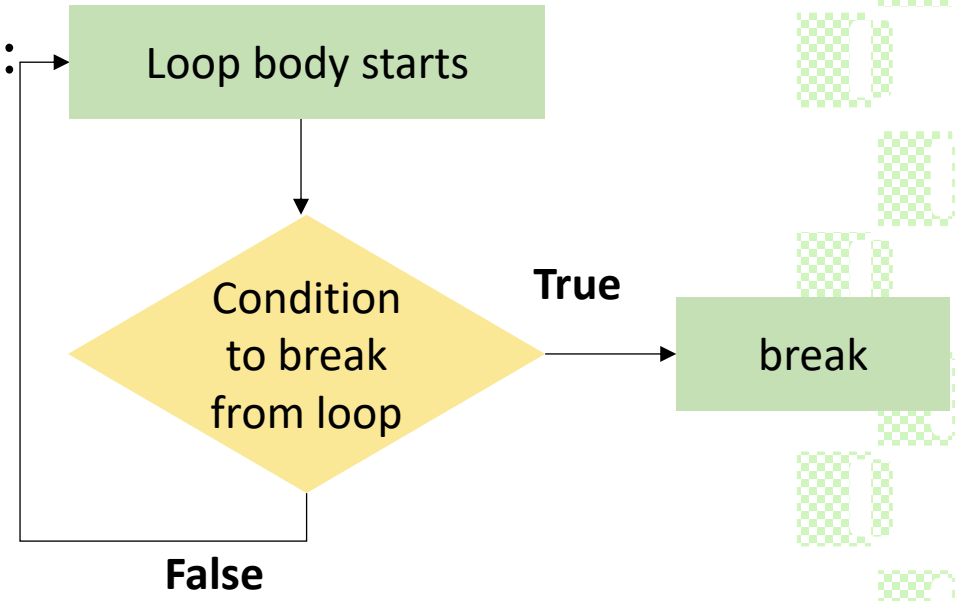
```
// Java Program to find maximum in an array Using for-each loop
public class Maximum
{
   public static void main(String[] args)
{
      int[] mark = {125, 132, 95, 116, 110};
      int max = findMax(mark);
      System.out.println("" + max);
   }
   public static int findMax(int[] n)
{
      int maximum = n[0];
for (int n1: n) {
         if (n1 > maximum) {
            maximum = n1;
         }
      }
      return maximum;
   }
}
```

Output:

132

3. Jump Statements - Break

- Break Statement is a loop control statement that is used to terminate the loop.
- As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.
- Break: In Java, the break is majorly used for:
- Terminate a sequence in a switch statement
- To exit a loop.
- Used as a "civilized" form of goto.



Syntax:
break;

Sample Program: Break

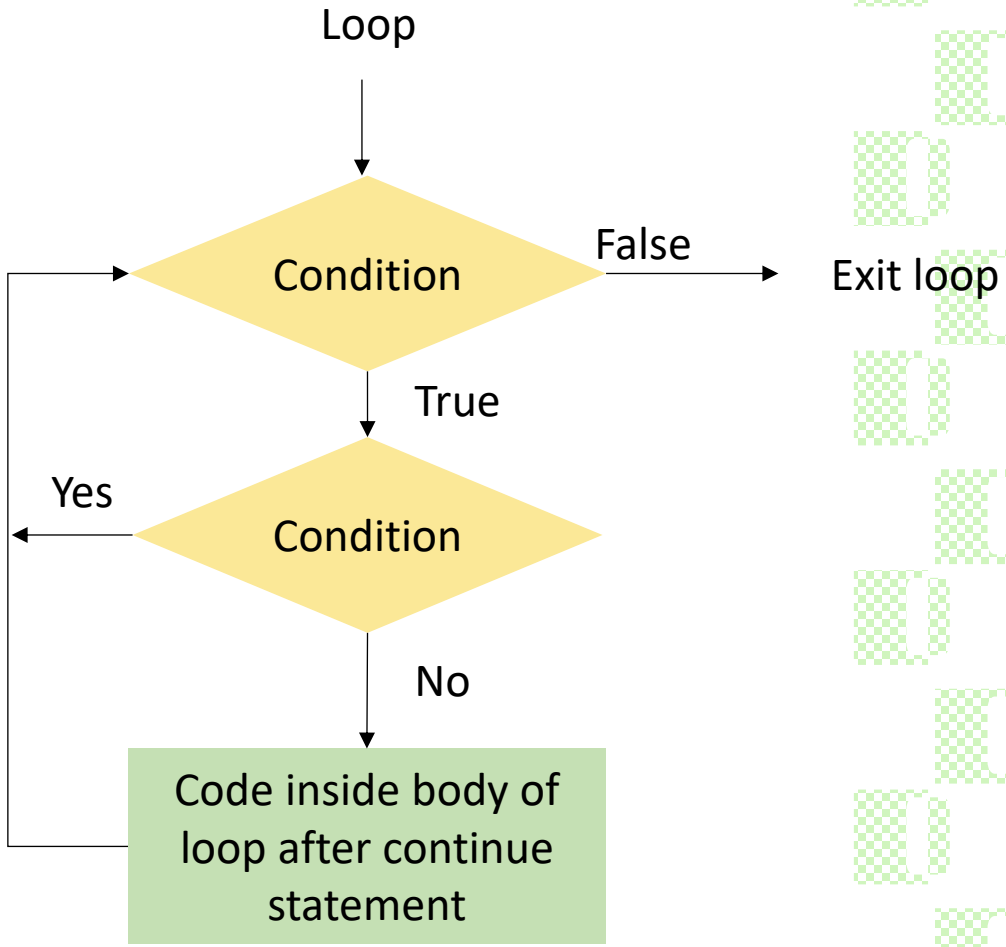
```
//Java Program to demonstrate the use of break statement

//inside the for loop.
public class BreakExample
{
public static void main(String[] args)
{
   //using for loop
   for(int i=1;i<=10;i++)
{
      if(i==5)
            {
         //breaking the loop
         break;
      }
      System.out.println(i);
   }
}
}
```

Output:

1
2
3
4

3. Jump Statements - continue

- The continue statement is used when we want to skip a particular condition and continue the rest execution.
- Java continue statement is used for all type of loops but it is generally used in for, while, and do-while loops.
- In the case of for loop, the continue keyword force control to jump immediately to the update statement.
- In the case of a while loop or do-while loop, control immediately jumps to the Boolean expression.



Syntax:
continue;

Sample Program : Continue

```
public class ContinueExample
{
public static void main(String[] args)
{
    //for loop
    for(int i=1;i<=10;i++)
    {
        if(i==5)
        {
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
}
```

Output:

1
2
3
4
6
7
8
9
10

Type Conversion/Widening

Automatic Type Conversion/ Widening:

- Widening takes place when two different datatypes are converted automatically if only it satisfies the following condition:
  - Two data types are compatible.
  - when we assign a value of a smaller data type to a bigger data type.

Byte → Short → Int → Long → Float → Double

```
// Java Program to Illustrate Automatic Type Conversion
class typeconversion
{
public static void main(String[] args)
        {
        int i = 100;
        long l = i;
        float f = l;
-       System.out.println("Int value " + i);
        System.out.println("Long value " + l);
        System.out.println("Float value " + f);
        }
}
```

Output:

Int value 100
Long value 100
Float value 100.0

Type Casting/Narrowing

Explicit Type Casting/Narrowing:

- If we want to assign a value of a larger data type to a smaller data type we perform explicit type casting or narrowing.
  - This is useful for incompatible data types where automatic conversion cannot be done.
  - Here, the target type specifies the desired type to convert the specified value to.

Double → Float → Long → Int → Short → Byte

```
// Java program to Illustrate Explicit Type
Conversion
public class typecasting
{
    public static void main(String[] args)
    {
            double d = 100.04;
            long l = (long)d;
            int i = (int)l;
            System.out.println("Double value " + d);
            System.out.println("Long value " + l);
            System.out.println("Int value " + i);
    }
}
```

Output:

Double value 100.04
Long value 100
Int value 100

Naming Conventions of Different Identifier

Identifiers Type	Naming Rules	Examples

Class	• It should start with the **uppercase letter.**
	• It should be a **noun** such as **Color, Button, System, Thread**, etc.
	• Use appropriate words, instead of acronyms.
		public class **Employee**
		{
		//code snippet
		}

Interface	• It should start with the **uppercase letter.**
	• It should be an **adjective** such as **Runnable, Remote, ActionListener**.
	• Use appropriate words, instead of acronyms.
		interface **Printable**
		{
		//code snippet
		}

Method	• It should start with **lowercase letter**.
	• It should be a **verb** such as **main(), print(), println().**
	• If the name contains multiple words, start it with a **lowercase letter followed by an uppercase letter such as actionPerformed().**
		class Employee
		{
		// method
		void **draw()**
		{
		//code snippet
		}
		}

Naming Conventions of Different Identifier

Identifiers Type	Naming Rules	Examples

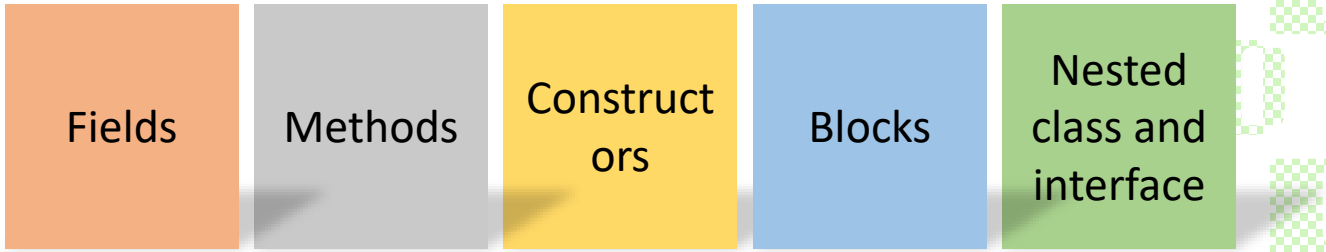
Variable	• It should start with a **lowercase letter** such as **id**, **name**.
	• It **should not start with the special characters** like **& (ampersand), $ (dollar), _ (underscore).**
	• If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as **firstName**, **lastName**.
	• Avoid using one-character variables such as x, y, z.
		class Employee
		{
		// variable
		int **id**;
		//code snippet
		}

Package	• It should be a **lowercase letter** such as **java**, **lang**.
	• If the name contains multiple words, it should be **separated by dots (.)** such as **java.util**, **java.lang**.
		//package
		package **com.javatpoint;**
		class Employee
		{
		//code snippet
		}

Constant	• It should be in **uppercase letters** such as **RED**, **YELLOW**.
	• If the name contains multiple words, it should be **separated by an underscore(_)** such as **MAX_PRIORITY.**
	• It may contain **digits** but **not as the first letter**.
		class Employee
		{
		//constant
		static final int **MIN_AGE** = 18;
		//code snippet
		}

Classes and Objects

- A class is a group of objects which have common properties.
- It is a template or blueprint from which objects are created.
- It is a logical entity. It can't be physical.
- A class in Java can contain:



Properties of Class

- Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Class does not occupy memory.
- Class is a group of variables of different data types and a group of methods.
- A Class in Java can contain:
  - Data member
  - Method
  - Constructor
  - Nested Class
  - Interface

A General form of Class

```
class classname
{
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list)
    {
        // body of method
    }
    type methodname2(parameter-list)
    {
        // body of method
    }
    // ...
    type methodnameN(parameter-list)
    {
        // body of method
    }
}
```

- The data, or variables, defined within a class are called instance variables.
- The code is contained within methods.
- Collectively, the methods and variables defined within a class are called members of the class.

Objects and its declaration

Objects

- An object is an instance of a class.
- A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- An object consists of :
  - **State** : It is represented by attributes of an object. It also reflects the properties of an object.
  - **Behavior** : It is represented by the methods of an object. It also reflects the response of an object with other objects.
  - **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

new keyword in Java

- The new keyword is used to allocate memory at runtime. All objects get memory in the Heap memory area.

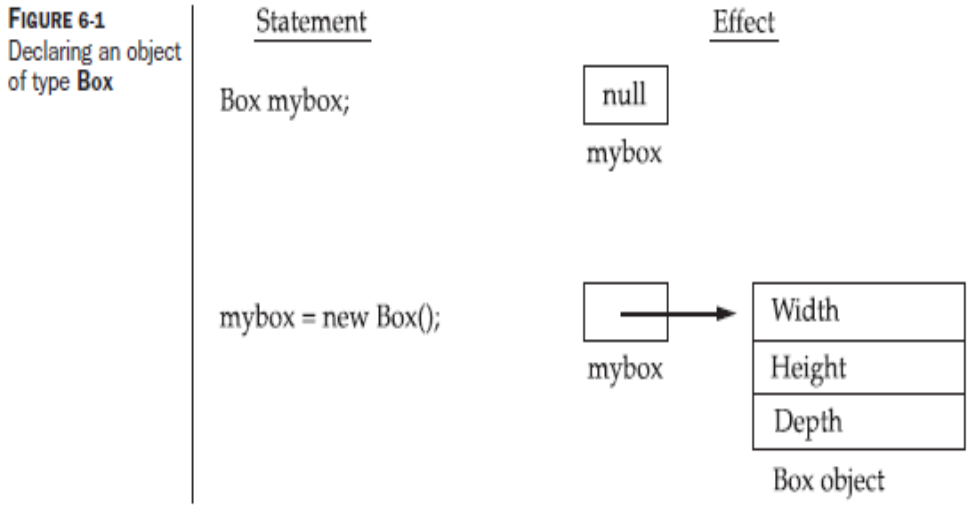


FIGURE 6-1
Declaring an object of type **Box**

- class-var is a variable of the class type being created.
- The classname is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the constructor for the class.
- A constructor defines what occurs when an object of a class is created.
- Constructors are an important part of all classes and have many significant attributes.

Difference between Class and Object

Class	Object
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a car.	Objects of the class car can be BMW, Mercedes, Ferrari, etc.

Example Program:

```
class Employee{
   int id;
   String name;
   float salary;
   void insert(int i, String n, float s)
   {
      id=i;
      name=n;
      salary=s;
   }
   void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
   Employee e1=new Employee();
   Employee e2=new Employee();
   Employee e3=new Employee();
   e1.insert(101,“Ajeet",45000);
   e2.insert(102,“Irfan",25000);
   e3.insert(103,“Nakul",55000);
   e1.display();
   e2.display();
   e3.display();
}
}
```

Output:

101 Ajeet 45000
102 Irfan 25000
103 Nakul 55000

Constructors

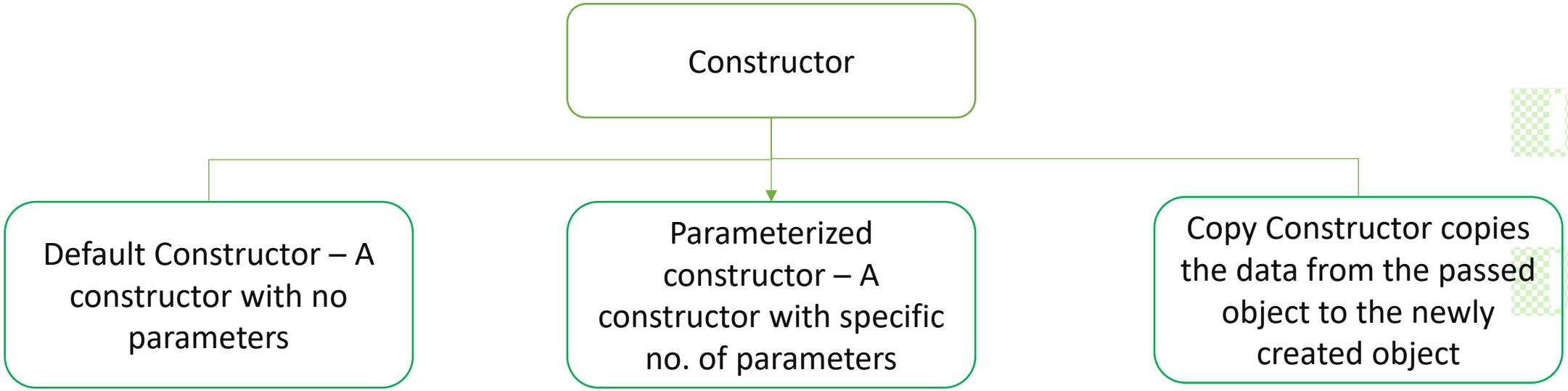
- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object 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.
- It is called constructor because it constructs the values at the time of object creation.

Rules for creating Constructor

- There are following rules for defining a 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 Constructor



Default Constructor

- A constructor that has no parameters is known as default the constructor.
- The default constructor can be implicit or explicit.
- If we don't define explicitly, we get an implicit default constructor.
- If we manually write a constructor, the implicit one is over-ridded.

Parameterized Constructor

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Copy Constructor

- Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

Example Program

```
public class Main
{
int x,y;
public Main()
{
x = 5;
}
public Main(int y1)
{
y=y1;
}
public static void main(String[] args)
{
Main myObj = new Main();
Main myObj1 = new Main(15);
System.out.println(myObj.x);
System.out.println(myObj1.y);
}
}
```

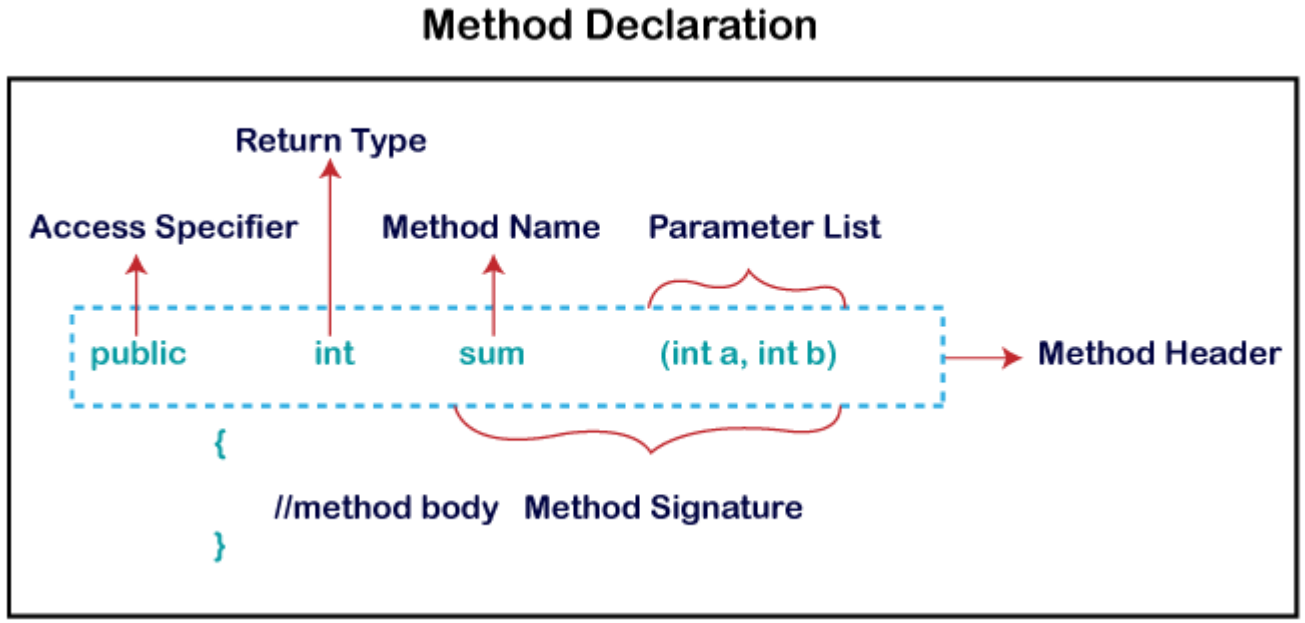
Output:
5
15

```
import java.io.*;
class Example1 {
   String name;
   int id;
 Example1(String name, int id)
   {
      this.name = name;
      this.id = id;
   }
 Example1(Example1 obj2)
   {
      this.name = obj2.name;
      this.id = obj2.id;
   }}
class paraexample      {
   public static void main(String[] args)
   {
      System.out.println("First Object");
 Example1 ex1 = new Example1("Jagan", 34);
      System.out.println("Name :" + ex1.name + " and Id :" + ex1.id);
 Example1 ex2 = new Example1(ex1);
      System.out.println("Copy Constructor used Second Object");
      System.out.println("Name :" + ex2.name+ " and Id :" + ex2.id);
   }
}
```

Output:
First Object
Name :Jagan and Id :34
Copy Constructor used Second Object
Name :Jagan and Id :34

Methods

- The method in Java is a collection of instructions that performs a specific task.
- It provides the reusability of code.
- It also provides the easy modification and readability of code, just by adding or removing a chunk of code.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

- Public: The method is accessible by all classes when we use public specifier in our application.
- Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Static Block

- When a block is decorated or associated with the word static, it is called a static block. Static Block is known as the static clause.
- A static block can be used for the static initialization of a class.
- The code that is written inside the static block run once, when the class is getting loaded into the memory.

```
public class StaticBlock
{
StaticBlock()
{
System.out.println("Inside the constructor of the class.");
}
public static void print()
{
System.out.println("Inside the print method.");
}
static
{
System.out.println("Inside the static block.");
}
public static void main(String[] args)
{
StaticBlock sbObj = new StaticBlock();
sbObj.print();
new StaticBlock();
}
}
```

Output:

```
Inside the static block.
Inside the constructor of the class.
Inside the print method.
Inside the constructor of the class.
```

Static data and Static method

- The static keyword in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static variable can be used to refer to the common property of all objects.
- The static variable gets memory only once in the class area at the time of class loading.
- Static variables in Java are also initialized to default values if not explicitly initialized by the programmer.
- Static variables are shared among all instances of the class, meaning if the value of a static variable is changed in one instance, it will reflect the change in all other instances as well.

Static variable - Program

```
class Student{
  int rollno;
  String name;
  static String college ="ITS";//static variable
   Student(int r, String n){
  rollno = r;
  name = n;
  }
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 s1.display();
 s2.display();
 }
}
```

Output:
111 Karan ITS
222 Aryan ITS

Static method

- If we apply a static keyword with any method, it is known as a static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data members and can change their value of it.

Static method - Program

```
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    static void change(){
    college = "BBDIT";
    }
   Student(int r, String n){
    rollno = r;
    name = n;
    }
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Output:

111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

String

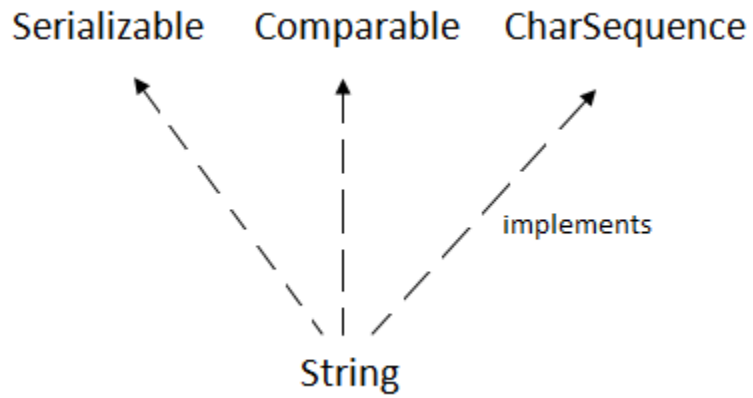
- In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

- In Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.
- Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
- The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.



String Declaration

- There are two ways to create String object:

1. By string literal
   - Java String literal is created by using double quotes.

     String s= “Welcome”;

2. By new keyword
   - String s=new String("Welcome");
   - JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String Program

```
import java.util.Scanner;
public class StringOperations {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the first string: ");
        String str1 = scanner.nextLine();
        System.out.print("Enter the second string: ");
        String str2 = scanner.nextLine();
        // a. String Concatenation
        String concatenatedString = str1.concat(str2);
        System.out.println("\nConcatenated String: " + concatenatedString);
        // b. Search a substring
        System.out.print("\nEnter the substring to search: ");
        String substring = scanner.nextLine();
        if (str1.contains(substring)) {
            System.out.println("The substring \"" + substring + "\" is found in the first string at index: " + str1.indexOf(substring));
        } else {
            System.out.println("The substring \"" + substring + "\" is not found in the first string.");
        }
```

```
// c. Extract substring
        System.out.print("\nEnter the starting index to extract substring from the first string: ");
        int startIndex = scanner.nextInt();
        System.out.print("Enter the ending index: ");
        int endIndex = scanner.nextInt();
        scanner.nextLine();
        if (startIndex >= 0 && endIndex <= str1.length() && startIndex < endIndex) {
            String extractedSubstring = str1.substring(startIndex, endIndex);
            System.out.println("Extracted Substring: " + extractedSubstring);
        } else {
            System.out.println("Invalid indices for substring extraction.");
        }
        scanner.close();
    }
}
```

Output:

```
Enter the first string: Bishop Heber
Enter the second string: College

Concatenated String: Bishop HeberCollege

Enter the substring to search: Heber
The substring "Heber" is found in the first string at index: 7

Enter the starting index to extract substring from the first string: 1
Enter the ending index: 7
Extracted Substring: ishop
```

StringBuffer class

- StringBuffer represents a mutable sequence of characters that ensures thread safety, making it suitable for scenarios involving multiple threads that modify a character sequence.
- It includes various string manipulation capabilities, including the ability to insert, delete, and append characters.
- Syntax: StringBuffer sb = **new** StringBuffer();

Constructor	Description
StringBuffer()	It creates an empty String buffer with the initial capacity of 16.
StringBuffer(String str)	It creates a String buffer with the specified string..
StringBuffer(int capacity)	It creates an empty String buffer with the specified capacity as length.

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.
public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

```
public class StringBufferExample {
    public static void main(String[] args) {

        StringBuffer sb = new StringBuffer("Hello");

        sb.append(", World!");

        sb.insert(5, " Java");

        sb.delete(5, 10);
        System.out.println(sb);
    }
}
```

Output:
Hello, World!

CharSequence interface

- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.
- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.