A SKILL ORIENTED COURSE ON

"Java Programming Language"

Submitted in partial fulfillment of the requirements for award of the degree of

BACHELOR OF TECHNOLOGY

In

ELECTRICAL AND ELECTRONICS ENGINEERING

By

YESWANTH GUNISETTY

(21A81A0276)

Under the Esteemed Coordination of

Mr. M. T. V. L. Ravi Kumar M.Tech

Assistant Professor

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

SRI VASAVI ENGINEERING COLLEGE (AUTONOMOUS)

Sponsored by Sri Vasavi Educational Society)

(Approved by AICTE, New Delhi and Permanently Affiliated to JNTUK, Kakinada)

(Accredited by NBA & NAAC with 'A' Grade)

Pedatadepalli, **T**ADEPALLIGUDEM**,** W.G. Dist., A.P– 534 101

Academic Year : 2023 – 2024

SRI VASAVI ENGINEERING COLLEGE (AUTONOMOUS)

(Sponsored by Sri Vasavi Educational Society)

(Approved by AICTE, New Delhi and Permanently Affiliated to JNTUK, Kakinada)

(Accredited by NBA & NAAC with 'A' Grade)

Pedatadepalli, **T**ADEPALLIGUDEM**,** W.G. Dist., A.P– 534 101

DEPARTMENT OF

ELECTRICAL & ELECTRONICS ENGINEERING

CERTIFICATE

This is to certify that the Skill Oriented Course report entitled **"JAVA PROGRAMMING LANGUAGE"** is a bonafide Work done by **YESWANTH GUNISETTY (21A81A0276)** submitted in partial fulfillment of the requirements for the award of the Degree in Electrical and Electronics Engineering during the academic year 2023-2024. The results of investigation enclosed in this report have been verified and found satisfactory.

SOC COORDINATOR **HEAD OF THEDEPARTMENT**

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We would like to thank our Skill Oriented Course Coordinator **Mr. M.T.V.L.Ravi Kumar** **M. Tech** for the guidance and help throughout the development of this Skill Oriented Course on “**JAVA PROGRAMMING LANGUAGE”** by providing us the required information. Without his supervision, support, and encouragement. We would not gain awareness of many new things during this Course.

We would like to thank **Dr. SUDHA RANI DONEPUDI,** **ME, Ph.D**. Head of the department of Electrical & Electronics Engineering for supporting us to do our Skill Oriented Course Successfully.

We would like to sincerely thank Principal of our college, **Dr.G.V.N.S.R.RATNAKARARAO ,** **BE, ME, Ph.D.** for his constant support throughout the Skill Oriented Course .

An Endeavour over long period can also be successful by constant effort and encouragement. We wish to take this opportunity to express our deep gratitude to all the people who have extended their cooperation in various ways during our Skill Oriented Course. It is our pleasure to acknowledge the help of all those respected individuals.

YESWANTH GUNISETTY
(21A81A0276)

SRI VASAVI ENGINEERING COLLEGE

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

Skill Oriented Course for the Academic Year : (2022-2023)

SKILL ORIENTED COURSE

Name of the Student	:	YESWANTH GUNISETTY
Roll Number	:	21A81A0276
Program	:	B. Tech
Branch	:	Electrical and Electronics Engineering
Topic	:	JAVA PROGRAMMING LANGUAGE
Coordinator	:	Mr. M. T. V. L. Ravi Kumar M. Tech

TABLE OF CONTENTS

CHAPETR-1: INTRODUCTION TO JAVA

Java is a class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is intended to let application developers write once, and run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was first released in 1995 and is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

JAVA was developed by James Gosling at Sun Microsystems Inc in the year 1995 and later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is like C/C++.

History: Java's history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the Green team initiated the Java language in 1991. Sun Microsystems released its first public implementation in 1996 as Java 1.0. It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

In 1997, Sun Microsystems approached the ISO standards body and later formalized Java, but it soon withdrew from the process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

CHAPETR-2: JAVA VIRTUAL MACHINE

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e., JVM is platform dependent).

WHAT IS JVM?

It is:

1. A specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. An implementation Its implementation is known as JRE (Java Runtime Environment).
3. Runtime Instance Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

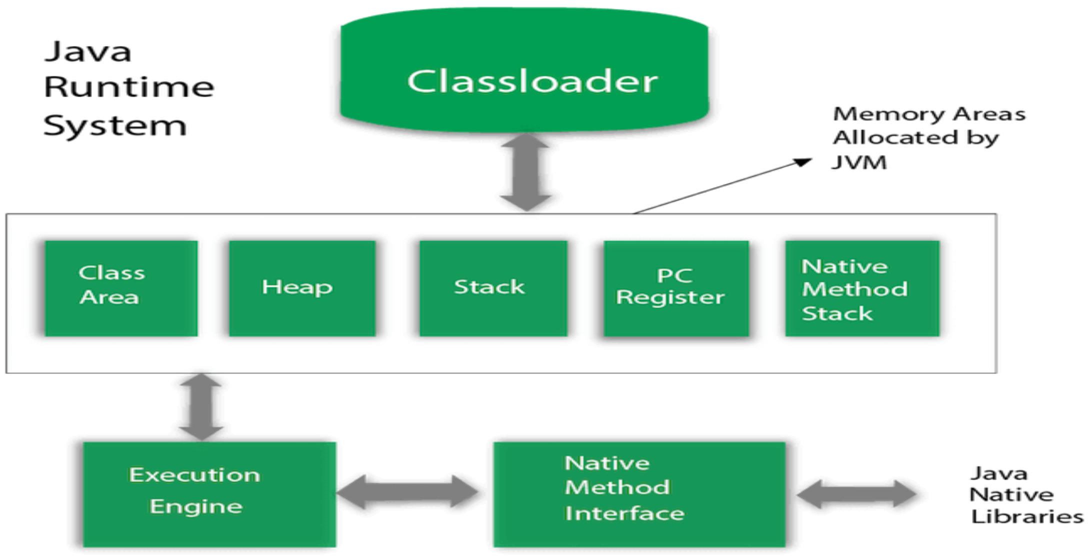
WHAT ITS DOES

The JVM performs following operation:

- Loads code.
- Verifies code.
- Executes code.
- Provides runtime environment.

JVM provides definitions for the:

- Memory area
- Class file format
- Register set.
- Garbage-collected heap
- Fatal error reporting etc.



CHAPETR-3: JAVA INDENTIFIERS

In Java, identifiers are used for identification purposes. Java Identifiers can be a class name, method name, variable name, or label.

Example of Java Identifiers

```
public class Test

{

   public static void main(String[] args)

   {

      int a = 20;

   }

}
```

In the above Java code, we have 5 identifiers namely :

- test : class name.
- main : method name.
- string : predefined class name.
- args : variable name.
- a : variable name.

Rules For Defining Java Identifiers:

There are certain rules for defining a valid Java identifier. These rules must be followed, otherwise, we get a compile-time error. These rules are also valid for other languages like C, and C++.

- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore).For example "geek@" is not a valid Java identifier as it contains a '@' a special character.
- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore).For example "geek@" is not a valid Java identifier as it contains a '@' a special character.
- Java identifiers are case-sensitive
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- Reserved Words can't be used as an identifier. For example "int while = 20;" is an invalid statement as a while is a reserved word. There are 53 reserved words in Java.

Examples of valid identifiers :

MyVariable
MYVARIABLE
myvariable
_myvariable

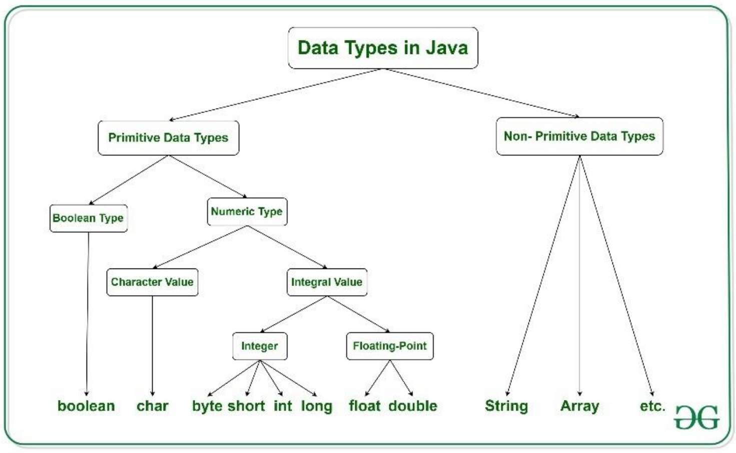
CHAPETR-4: JAVA DATA TYPES

Java is statically typed and a strongly typed language because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

What are Data Types in Java?

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. Primitive Data Type: such as boolean, char, int, short, byte, long, float, and double.
2. Non-Primitive Data :Type or Object Data type: such as String, Array, etc.



Primitive Data Types in Java

Primitive data are only single values and have no special capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows:

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

CHAPETR-5: AUTOBOXING AND UNBOXING

In Java, primitive data types are treated differently so do there comes the introduction of wrapper classes where two components play a role namely Autoboxing and Unboxing.

Autoboxing:

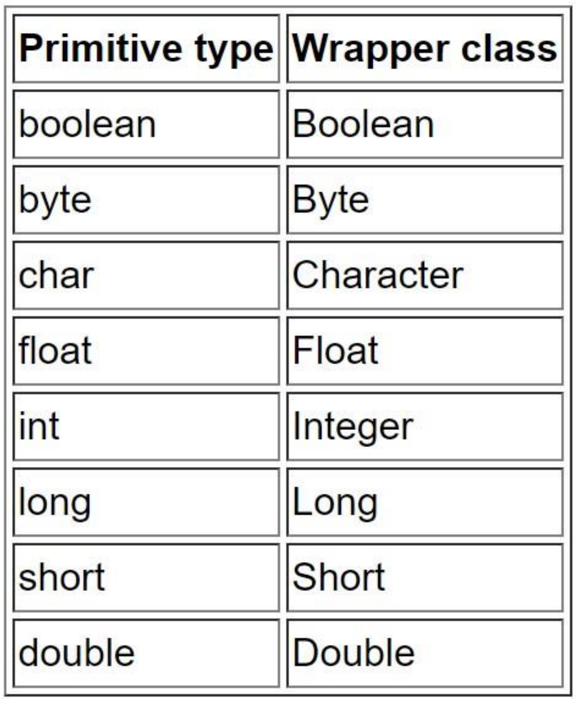
refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Unboxing:

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.



Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing. Now let us discuss a few advantages of autoboxing and unboxing to get why we are using it.

- Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
- The technique lets us use primitive types and Wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

CHAPETR-6: TYPE CASTING

Type casting is when you assign a value of one primitive data type to another type.
In Java, there are two types of casting:

- **Widening Casting** (automatically) - converting a smaller type to a larger type size
  `byte -> short -> char -> int -> long -> float -> double.`
- **Narrowing Casting** (manually) - converting a larger type to a smaller size type
  `double -> float -> long -> int -> char -> short -> byte.`

Widening Casting :

Widening casting is done automatically when passing a smaller size type to a larger size type:
Example :

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);
    System.out.println(myDouble);
  }
}
```

Output :

```
9
9.0
```

Narrowing Casting :

Narrowing casting must be done manually by placing the type in parentheses in front of the value:
Example :

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

Output

```
9.78
9
```

CHAPETR-7: INPUT AND OUTPUT

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:

Example

```java
import java.util.Scanner; // import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    String userName;

    // Enter username and press Enter
    System.out.println("Enter username");
    userName = myObj.nextLine();

    System.out.println("Username is: " + userName);
  }
}
```

Output

```
Enter username
Sri Vasavi

Username is: Sri Vasavi
```

Input Types

In the example above, we used the nextLine() method, which is used to read Strings. To read other types, look at the table below:

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

In the example below, we use different methods to read data of various types:

```
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");

    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

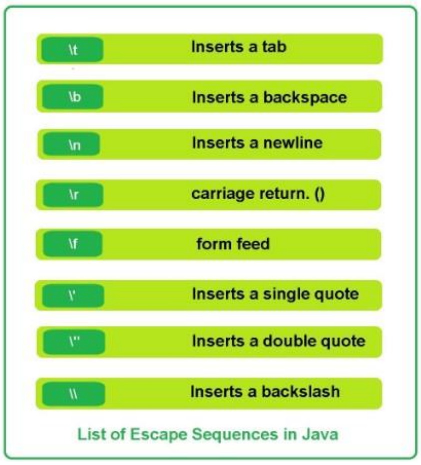
<u>Output</u>

```
Enter name, age and salary:
Sri Vasavi
26
100000

Name: Sri Vasavi
Age: 26
Salary: 100000
```

CHAPETR-8:. ESCAPE SEQUENCES IN JAVA

A character with a backslash (\) just before it is an escape sequence or escape character. We use escape characters to perform some specific task. The total number of escape sequences or escape characters in Java is 8. Each escape character is a valid character literal.

The list of Java escape sequences:



\t	Inserts a tab
\b	Inserts a backspace
\n	Inserts a newline
\r	carriage return. ()
\f	form feed
\'	Inserts a single quote
\"	Inserts a double quote
\\	Inserts a backslash

List of Escape Sequences in Java

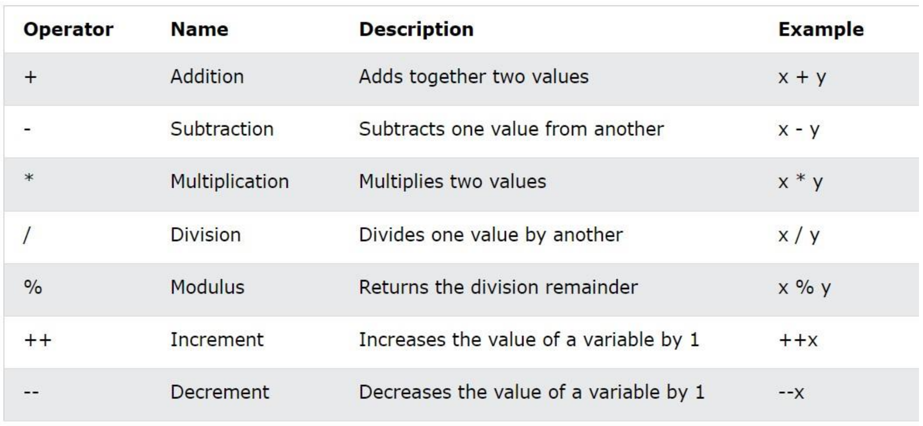
CHAPETR-9: OPERATORS

Operators are used to perform operations on variables and values.
Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

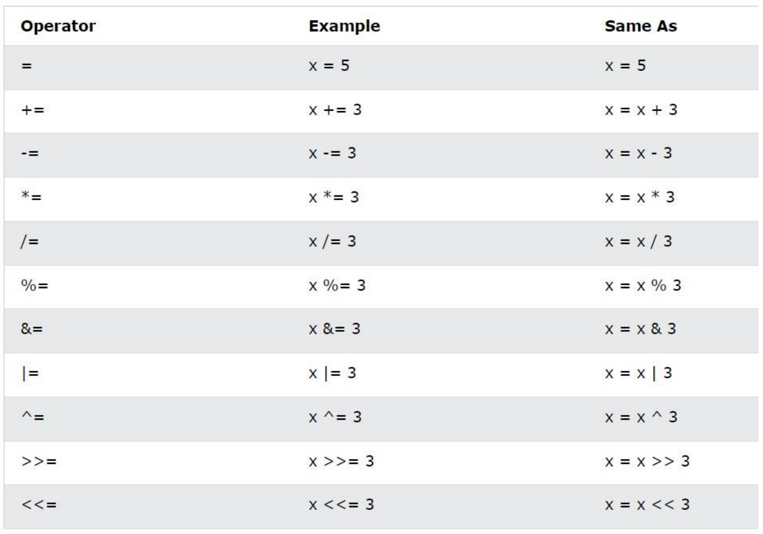
Arithmetic operators are used to perform common mathematical operations.



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

Assignment Operators

A list of all assignment operators:



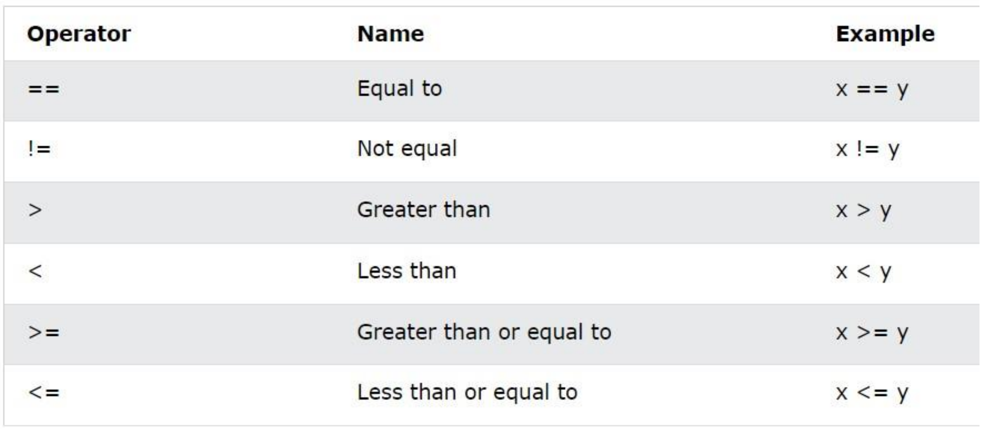
Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as Boolean values, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the greater than operator (>) to find out if 5 is greater than 3:

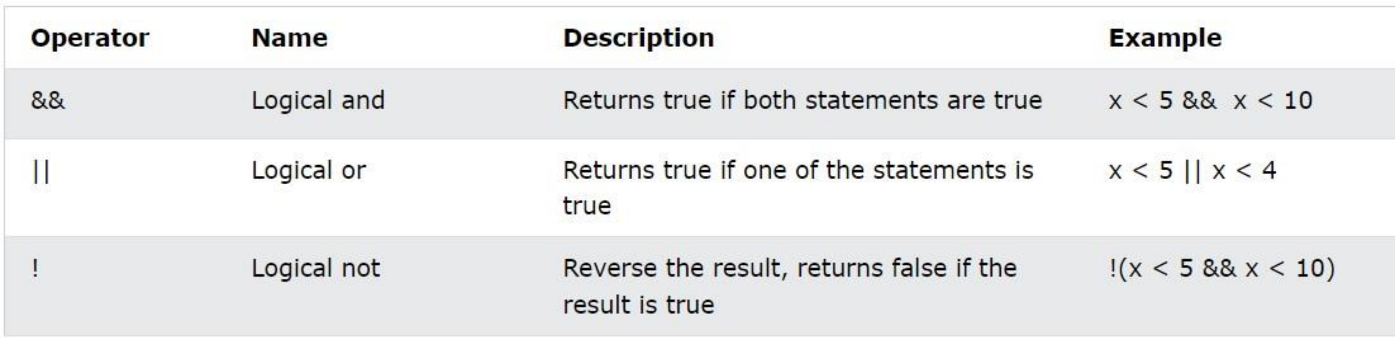


Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values:

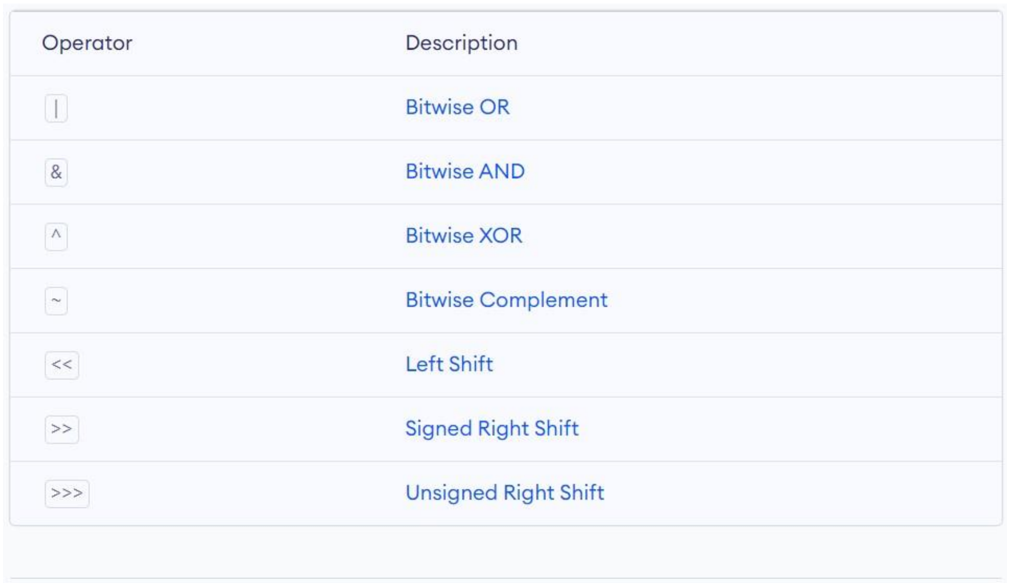


Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Bitwise Operators

Bitwise operators are used to performing the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.). They are used when performing update and query operations of the Binary indexed trees.

Now let’s look at each one of the bitwise operators in Java:



Operator	Description
|	Bitwise OR
&	Bitwise AND
^	Bitwise XOR
~	Bitwise Complement
<<	Left Shift
>>	Signed Right Shift
>>>	Unsigned Right Shift

Example Programs :

```
public class swap
{
    public static void main(String[] args)
    {
        int a,b;
        a=10;
        b=20;
        a=a^b;
        b=a^b;
        a=a^b;
        System.out.println("a= "+a+"\t b= "+b);
    }
}
```

Output

```
a= 20    b= 10
```

```
public class imc
{
    public static void main(String[] args){
        int x=10,y;
        System.out.println("Pre Increment");
        y=++x;
        System.out.println("y= "+y+"\t x= "+x);
        System.out.println("Post Increment");
        y=x++;
        System.out.println("y= "+y+"\t x= "+x);
        System.out.println("Pre Decrement");
        y=--x;
        System.out.println("y= "+y+"\t x= "+x);
        System.out.println("Post Decrement");
        y=x--;
        System.out.println("y= "+y+"\t x= "+x);

    }
}
```

Output

```
Pre-Increment
y= 11    x= 11
Post Increment
y= 11    x= 12
Pre-Decrement
y= 11    x= 11
Post Decrement
y= 11    x= 10
```

CHAPETR-10: CONDITIONAL STATEMENTS

Conditional statements in Java are utilized to make decisions in a program based on some specific conditions. The main constraints are ‘if,’ ‘else,’ and ‘else if.' These statements allow the program to execute different lines of code depending on the scenario, whether a particular condition is true or false. These are vital for controlling the flow of a program that enables the developers to create the logic that is able to respond dynamically to the data or the input, which further helps in making the program more powerful and flexible.

Types of Java Conditional Statements

1. Java If Statement

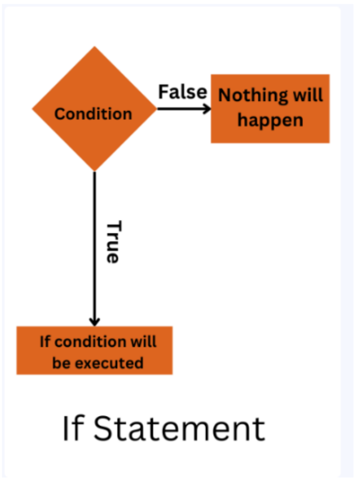
2. Java If-Else Statement

3. Java If-Else-If Ladder Statement

4. Java Nested If Statement

5. Java Switch Statement

1. Java If Statement



Suppose a condition is true if a statement is used to run the program. It is also known as a one-way selection statement. If a condition is used, an argument is passed, and if it is satisfied, the corresponding code is executed; otherwise, nothing happens.

Syntax

```
if (expression) {
  // You can enter the code here
}
```

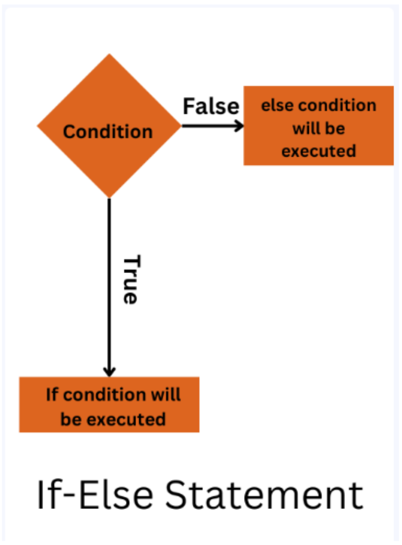
Example

```
import java.util.*;
class sample {
    public static void main(String[] args) {
        int a = -3;
        if (a < 0)   {
            System.out.println(a + " is a Negative Number.");
        }
    }
}
```

Output

-3 is a Negative Number.

2. Java If-Else Statement


Condition
False
else condition will be executed
True
If condition will be executed
If-Else Statement

An if-else statement is a control structure that determines which statements to choose based on a set of conditions.

Syntax

```
if (condition) {
  // Statements that will be carried out if the condition is satisfied
} else {
  // Statements that will be carried out if the condition is not met
}
```

Example

```java
import java.util.*;
class sample {
    public static void main(String[] args) {
        int a = 10;
        if (a % 2 == 0)     {
            System.out.println(a + " is an even number");
        }
        else     {
            System.out.println(a + " is an odd number");
        }
    }
}
```

Output

10 is an even number

Using Ternary Operator

The Java ternary operator offers a concise syntax for determining if a condition is true or false. It returns a value based on the outcome of the Boolean test.

One can replace their Java if-else statement using a ternary operator to create compact code. Expert coders enjoy the clarity and simplicity the Java ternary operator adds to their code.

The syntax of the Java Ternary operator is as follows:

```
(condition) ? (return if true) : (return if false);
```

Let's understand the concept with the help of a code example. We will see the above code using the ternary operator instead of the Java If-else statement.

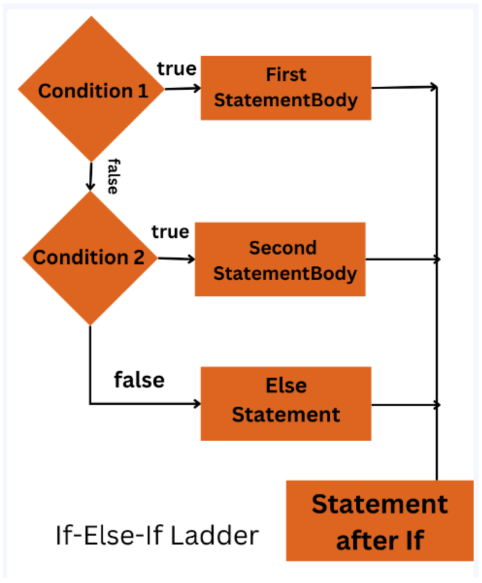
```
class example {
  public static void main( String args[] ) {
    int a = 10;
    String teropt =  (a % 2 == 0) ? a + " is an even number" : a + " is an odd number";
    System.out.println(teropt);
  }
}
```

Output

```
10 is an even number
```

3. Java If-Else-If Ladder Statement

An if-else-if ladder in Java can execute one code block while multiple other blocks are executed.



Syntax

```
if (condition1) {
   // Executable code if condition1 is true,
} else if (condition2) {
   // Executable code if condition2 is true
}


...
else {
   // Executable code if all the conditions are false
}
```

Example

```
import java.util.*;
class sample {
    public static void main(String[] args) {
        int i = 3;
        if (i == 1)
             System.out.println("January");
        else if (i == 2)
             System.out.println("February");
        else if (i == 3)
             System.out.println("March");
        else
             System.out.println("April");
    }
}
```

Output

```
March
```

4. Java Nested If Statement

If conditions inside another if conditions are called as Nested If conditional statements in java.

Syntax

```
if (condition1) {
  // Statement 1 will execute

  if (condition2)  {
   // Statement 2 will execute
  }
}
```

Example

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

        int a=10;

        if(a%5==0)
        {
            //Divisible by 5
            System.out.println("Divisibe by 5");
        }
        else
        {
            //Not divisible by 5
            System.out.println("Not dividible by 5");
        }
    }
}
```

<u>Output</u>

Divisible by 5

5. Java Switch Statement

Unlike the if-else statement, the switch statement has more than one way to be executed. Additionally, it compares the expression's value to its cases by focusing its evaluation on a few primitive or class types.

Syntax

```
switch (Expression) {
   case value 1:
      // Statement 1;
   case value 2:
      // Statement 2;
   case value 3:
      // Statement 3;


      …
   case value n:
      // Statement n;
   Default:
      // default statement;
}
```

Example

```
import java.util.*;
public class daysusingswitch
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        int daynum;
        System.out.print("Enter a day number(1-7) :");
        daynum=sc.nextInt();
        switch(daynum)
        {
            case 1:
                System.out.println("Sunday");
                break;
            case 2:
                System.out.println("Monday");
                break;
            case 3:
                System.out.println("Tuesday");
                break;
            case 4:
                System.out.println("Wednesday");
                break;
            case 5:
                System.out.println("Thursday");
                break;
            case 6:
                System.out.println("Friday");
                break;
            case 7:
                System.out.println("Saturday");
                break;
            default:
                System.out.println("Invalid day num");
                break;
        }
        sc.close();
    }
}
```

Output

```
Enter a day number(1-7) :5
Thursday
```

CHAPETR-11: LOOP STATEMENTS

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

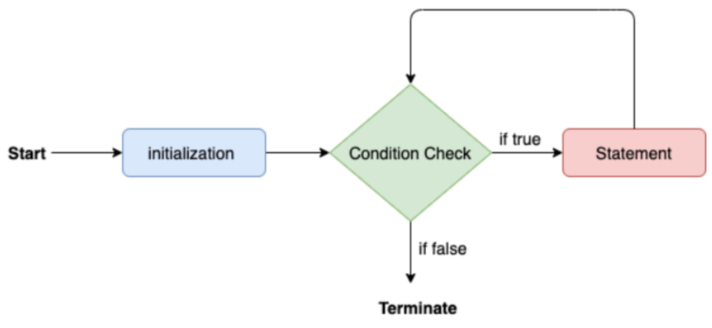
In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

1. Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```



Consider the following example to understand the proper functioning of the for loop in java.

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
//statements
}
```

Consider the following example to understand the functioning of the for-each loop in Java.

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

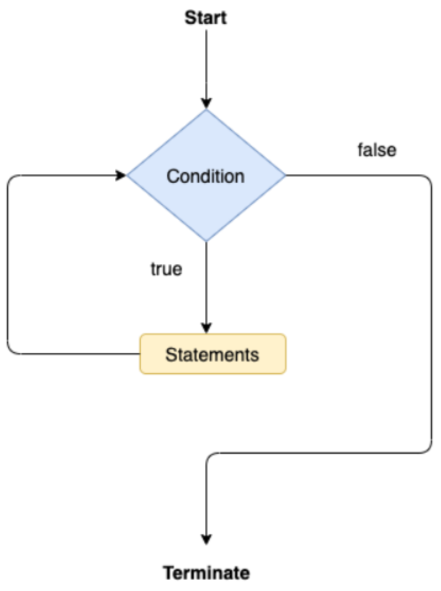
2. Java while loop

The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below

```
while(condition){
//looping statements }
```



```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output

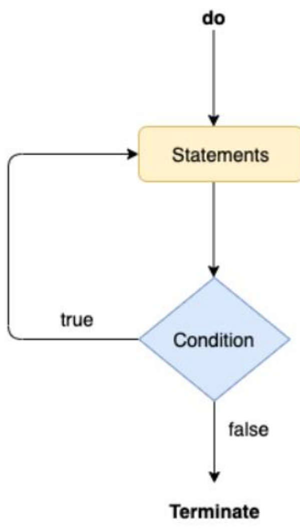
```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

3. Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is

```
do
{
//statements
} while (condition);
```



Example

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);
}
}
```

Output

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Example Programs

```
import java.util.*;
public class leapyear
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter new year ");
        int n=sc.nextInt();
        if ((n%4==0)||((n%100==0) && (n%400==0)))
        {
            System.out.println(n+" is leap year");
        }
        /*if(n%100==0 && n%400==0)
        {
            System.out.println(n+" is a leap year");
        }
        else if((n%4==0) && (n%100!=0))
        {
            System.out.println(n+" is a leap year");
        }*/
        else
        {
            System.out.println(n+" is not a leap year");
        }
        sc.close();
    }
}
```

Output

```
Enter new year
1998
1998 is not a leap year
```

```
import java.lang.Math;
import java.util.*;
public class gp
{
    public static void main(String[] args)
    {
        float a,r,n;
        double result;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter a,r,n values : ");
        a=sc.nextFloat();
        r=sc.nextFloat();
        n=sc.nextFloat();
        result=a*(Math.pow(r,(n-1)));
        System.out.println(result);
        sc.close();
    }
}
```

Output

```
Enter a,r,n values :
4
5
6
12500.0
```

Nested Loop in Java

If a loop exists inside the body of another loop, it's called a nested loop. Here's an example of the nested for loop.

Syntax

```
// outer loop
for (int i = 1; i <= 5; ++i) {
  // codes

  // inner loop
  for(int j = 1; j <=2; ++j) {
    // codes
  }
..
}
```

Here, we are using a for loop inside another for loop.

Example: Java Nested for Loop

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

    int weeks = 3;
    int days = 7;

    // outer loop prints weeks
    for (int i = 1; i <= weeks; ++i) {
      System.out.println("Week: " + i);

      // inner loop prints days
      for (int j = 1; j <= days; ++j) {
        System.out.println("  Day: " + j);
      }
    }
  }
}
```

Output

```
Week: 1
  Day: 1
  Day: 2
  Day: 3
    .....  ..  ....
Week: 2
  Day: 1
  Day: 2
  Day: 3
  ....  ..  ....
....  .. ....
```

CHAPETR-12:. JUMP STATEMENTS

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

Java break statement

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

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

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

```
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output

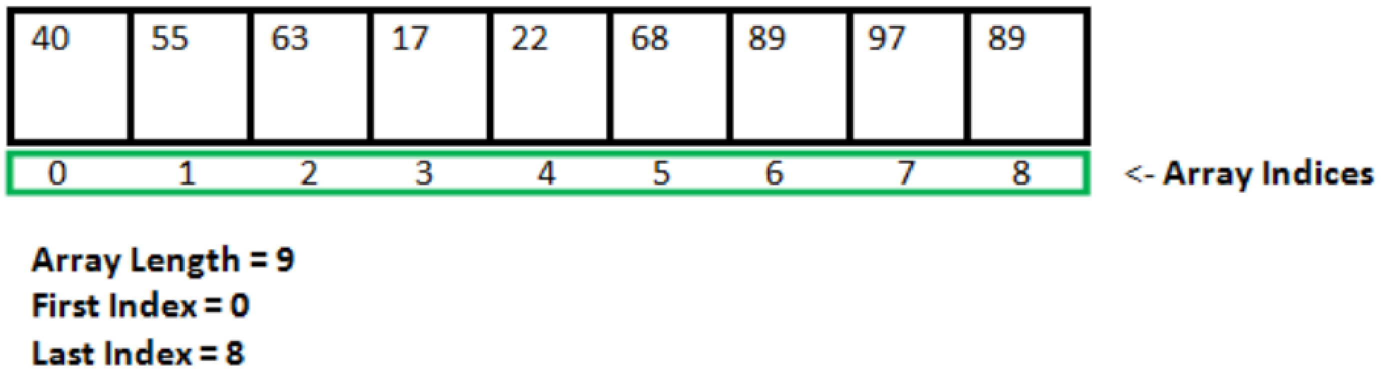
```
0
1
2
3
4
5
6
```

CHAPETR-13: ARRAYS

Array in java is a group of like-typed variables referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Arrays are stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int or short value and not long.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.
- This storage of arrays helps us randomly access the elements of an array [Support Random Access].
- The size of the array cannot be altered(once initialized). However, an array reference can be made to point to another array.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations. In the case of class objects, the actual objects are stored in a heap segment.



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];
```

Example of Java Array

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output

```
10
20
70
40
50
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
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
```

Jagged Array in Java:

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Output

```
0 1 2
3 4 5 6
7 8
```

CHAPETR-14: METHODS

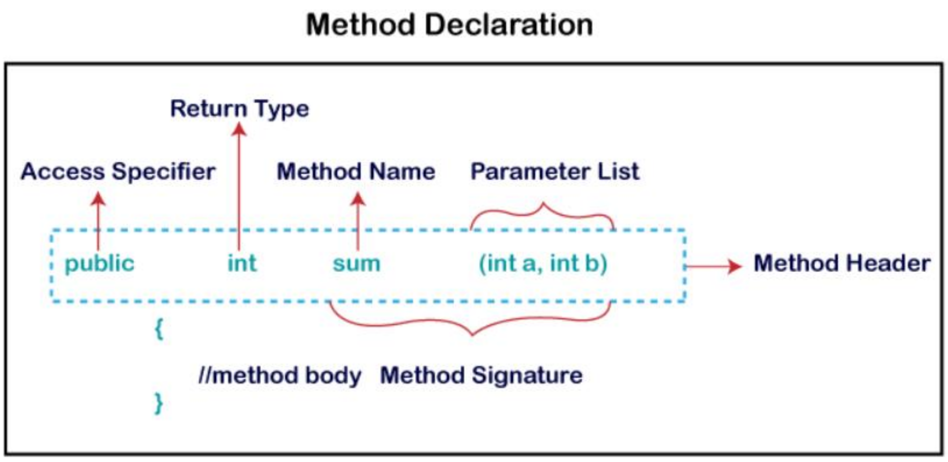
In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using methods. In this section, we will learn what is a method in Java, types of methods, method declaration, and how to call a method in Java.

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the main() method. If you want to read more about the main() method, go through the link https://www.javatpoint.com/java-main-method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as method header, as we have shown in the following figure.



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.

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.

Naming a Method :

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```

Output

The maximum number is: 9

User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

How to Create a User-defined Method
Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

We have defined the above method named findevenodd(). It has a parameter **num** of type int. The method does not return any value that's why we have used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

How to Call or Invoke a User-defined Method
Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
```

In the above code snippet, as soon as the compiler reaches at line **findEvenOdd(num),** the control transfer to the method and gives the output accordingly.
Let's combine both snippets of codes in a single program and execute it.

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
}
```

Output

```
Enter the number: 12
12 is even
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

```
public class Display
{
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output

```
It is an example of a static method.
```

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

```
public class InstanceMethodExample
{
public static void main(String [] args)
{
//Creating an object of the class
InstanceMethodExample obj = new InstanceMethodExample();
//invoking instance method
System.out.println("The sum is: "+obj.add(12, 13));
}
int s;
//user-defined method because we have not used static keyword
public int add(int a, int b)
{
s = a+b;
//returning the sum
return s;
}
}
```

Output

```
The sum is: 25
```

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Abstract Method

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

CHAPETR-15: COMMAND LINE ARGUMENTS

The java command-line argument is an argument i.e., passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behaviour of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}

compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo
```

Output: Your first argument is: sonoo

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

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

for(int i=0;i<args.length;i++)
System.out.println(args[i]);

}
}

compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc
```

Output:

```
 sonoo
 jaiswal
 1
 3
 Abc
```

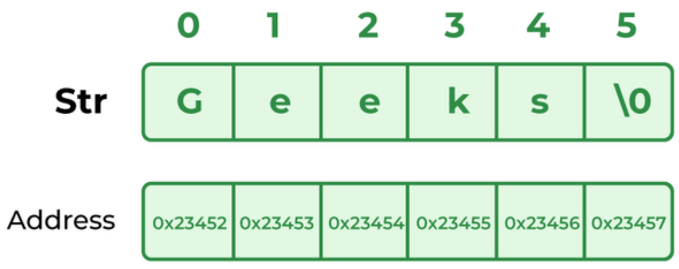
CHAPETR-16: STRINGS

In the given example only one object will be created. Firstly, JVM will not find any string object with the value “Welcome” in the string constant pool, so it will create a new object. After that it will find the string with the value “Welcome” in the pool, it will not create a new object but will return the reference to the same instance. In this article, we will learn about Java Strings.

What are Strings in Java?

Strings are the type of objects that can store the character of values and in Java, every character is stored in 16 bits i,e using UTF 16-bit encoding. A string acts the same as an array of characters in Java.

Example:
String name = "Geeks";



Below is an example of a String in Java:

```
// Java Program to demonstrate
// String
public class StringExample {

    // Main Function
    public static void main(String args[])
    {
        String str = new String("example");
        // creating Java string by new keyword
        // this statement create two object i.e
        // first the object is created in heap
        // memory area and second the object is
        // created in String constant pool.

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

<u>Output</u>
Example

Ways of Creating a String
There are two ways to create a string in Java:
- String Literal
- Using new Keyword

Syntax:

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

1. String literal

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Example:

```
String demoString = “GeeksforGeeks”;
```

2. Using new keyword

- String s = new String(“Welcome”);
- In such a case, 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 the heap (non-pool)

Example:

```
String demoString = new String (“GeeksforGeeks”);
```

Interfaces and Classes in Strings in Java

CharBuffer: This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.

String: It is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

CharSequence Interface

CharSequence Interface is used for representing the sequence of Characters in Java. Classes that are implemented using the CharSequence interface are mentioned below and these provides much of functionality like substring, lastoccurence, first occurence, concatenate , toupper, tolower etc.

1. String
2. StringBuffer
3. StringBuilder

1. String:

String is an immutable class which means a constant and cannot be changed once created and if wish to change, we need to create an new object and even the functionality it provides like toupper, tolower, etc all these return a new object , its not modify the original object. It is automatically thread safe.

Syntax

```
String str= "geeks";
           or
String str= new String("geeks");
```

2. StringBuffer:

StringBuffer is a peer class of String that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences means it is immutable in nature and it is thread safe class , we can use it when we have multi-threaded environment and shared object of string buffer i.e., used by mutiple thread. As it is thread safe so there is extra overhead, so it is mainly used for multithreaded program.

Syntax:

```
StringBuffer demoString = new StringBuffer("GeeksforGeeks");
```

3. StringBuilder:

StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters and it is not thread safe and its used only within the thread , so there is no extra overhead , so it is mainly used for single threaded program.

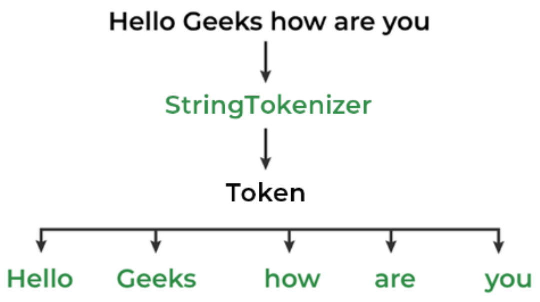
Syntax:

```
StringBuilder demoString = new StringBuilder();
demoString.append("GFG");
```

StringTokenizer :

StringTokenizer class in Java is used to break a string into tokens.

Example:



A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

StringJoiner is a class in *java.util* package which is used to construct a sequence of characters(strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix. Though this can also be done with the help of the StringBuilder class to append a delimiter after each string, StringJoiner provides an easy way to do that without much code to write.

Syntax:

```
public StringJoiner(CharSequence delimiter)
```

Here the JVM checks the String Constant Pool. If the string does not exist, then a new string instance is created and placed in a pool. If the string exists, then it will not create a new object. Rather, it will return the reference to the same instance. The cache that stores these string instances is known as the String Constant pool or String Pool. In earlier versions of Java up to JDK 6 String pool was located inside PermGen(Permanent Generation) space. But in JDK 7 it is moved to the main heap area.

<u>Immutable String in Java :</u>

In Java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once a string object is created its data or state can't be changed but a new string object is created.

Below is the implementation of the topic:

```java
// Java Program to demonstrate Immutable String in Java
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        String s = "Sachin";

        // concat() method appends
        // the string at the end
        s.concat(" Tendulkar");

        // This will print Sachin
        // because strings are
        // immutable objects
        System.out.println(s);
    }
}
```

Output :
Sachin

Here Sachin is not changed but a new object is created with "Sachin Tendulkar". That is why a string is known as immutable.

As you can see in the given figure that two objects are created but s reference variable still refers to "Sachin" and not to "Sachin Tendulkar". But if we explicitly assign it to the reference variable, it will refer to the "Sachin Tendulkar" object.

For Example:

```java
// Java Program to demonstrate Explicitly assigned strings
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        String name = "Sachin";
        name = name.concat(" Tendulkar");
        System.out.println(name);
    }
}
```

Output:
Sachin Tendulkar

Memory Allotment of String :

Whenever a String Object is created as a literal, the object will be created in the String constant pool. This allows JVM to optimize the initialization of String literal.

Example:
```java
String demoString = "Geeks";
```

The string can also be declared using a **new** operator i.e. dynamically allocated. In case of String are dynamically allocated they are assigned a new memory location in the heap. This string will not be added to the String constant pool.

Example:
```java
String demoString = new String("Geeks");
```

If you want to store this string in the constant pool then you will need to “intern” it.
Example:
```java
String internedString = demoString.intern();
// this will add the string to string constant pool.
```

It is preferred to use String literals as it allows JVM to optimize memory allocation.
An example that shows how to declare a String

```java
// Java code to illustrate String
import java.io.*;
import java.lang.*;

class Test {
    public static void main(String[] args)
    {
        // Declare String without using new operator
        String name = "GeeksforGeeks";

        // Prints the String.
        System.out.println("String name = " + name);

        // Declare String using new operator
        String newString = new String("GeeksforGeeks");

        // Prints the String.
        System.out.println("String newString = " + newString);
    }
}
```

Output
```
String Stringname = GeeksforGeeks
String newString = GeeksforGeeks
```

String Methods

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
codePointAt()	Returns the Unicode of the character at the specified index	int
codePointBefore()	Returns the Unicode of the character before the specified index	int
codePointCount()	Returns the number of Unicode values found in a string.	int
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
contentEquals()	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
copyValueOf()	Returns a String that represents the characters of the character array	String
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format()	Returns a formatted string using the specified locale, format string, and arguments	String
getBytes()	Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array	byte[]
getChars()	Copies characters from a string to an array of chars	void
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
intern()	Returns the canonical representation for the string object	String
isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
offsetByCodePoints()	Returns the index within this String that is offset from the given index by codePointOffset code points	int
regionMatches()	Tests if two string regions are equal	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

Program on Methods :

```
import java.util.*;
public class stringmethods
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter fn :");
        String fn=sc.next();
        System.out.println("Enter ln :");
        String ln=sc.next();

        //concatenation
        System.out.println("String after concatenation : ");
        String s=fn.concat(" "+ln);
        System.out.println(fn.concat(" "+ln));

        //length
        System.out.println("length of the fn string is :");
        System.out.println(fn.length());
        System.out.println("Length of the ln string : ");
        System.out.println(ln.length());

        //Accessing Characters
        System.out.println("Character at index 8 in string fn :");
        System.out.println(fn.charAt(7));

        //substring
        System.out.println("Substring of fn at index 4: ");
        System.out.println(fn.substring(4));

        //equal of the string
        System.out.println("comparing string : ");
        System.out.println(fn.equals(ln));

        //Indexing
        System.out.println("Index of fn string :");
        int index=s.indexOf("gunisetty");
        System.out.println(index);

        //Manuplating
        System.out.println("replace fn string :");
        String replaced=fn.replace(fn,"Yeswanth");
        System.out.println(replaced);

        //split
        System.out.println("Split the string s :");
        String[] y=s.split(" ");
        System.out.println(Arrays.toString(y));

        //trim method
        System.out.println("trim the string fn : ");
        System.out.println(fn.trim());
        //case conversion
        System.out.println("convert firstletter of fn to upper case : ");
        String firstletter=fn.substring(0,1).toUpperCase()+fn.substring(1);
        System.out.println(firstletter);
        sc.close();
    }
}
```

Output

```
Enter fn :
yeswanth
Enter ln :
Gunisetty
String after concatenation :
yeswanth Gunisetty
length of the fn string is :
8
Length of the ln string :
9
Character at index 8 in string fn :
h
Substring of fn at index 4:
anth
comparing string :
false
Index of fn string :
-1
replace fn string :
Yeswanth
Split the string s :
[yeswanth, Gunisetty]
trim the string fn :
yeswanth
convert firstletter of fn to upper case :
Yeswanth
```

Example Programs

```
import java.util.*;
public class countchar
{
    public static void main(String[] args)
    {
        int digits=0,chars=0,spl=0;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter string : ");
        String str=sc.next();
        for(int i=0;i<str.length();i++)
        {
            if(Character.isDigit(str.charAt(i)))
            {
                digits++;
            }
            else if(Character.isLetter(str.charAt(i)))
            {
                chars++;
            }
            else
            {
                spl++;
            }
        }
        System.out.println("No.of digts are : "+digits+"\tno.of characters are
:"+chars+"\tno.of spl character are :"+spl);
        sc.close();
    }
}
```

<u>Output</u>

```
Enter string :
srivasavi123
No.of digts are : 3     no.of characters are :9      no.of spl character are :0
```

```
import java.util.*;
public class removeduplicate
{
    public static void main(String[] args)
    {
        String res="";
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter string : ");
        String str=sc.nextLine();
        char string[]=str.toCharArray();
        for(int i=0;i<string.length;i++)
        {
            int j;
            for(j=0;j<i;j++)
            {
                if(string[i]==string[j] && string[i]!=' ' && string[i]!='0' )
                {
                    break;
                }
            }
            if(i==j)
            {
                res=res+string[i];
            }

        }

        System.out.print(res);
        sc.close();
    }
}
```

Output

```
Enter string :
sri vasavi
sri va
```

CHAPETR-17: OBJECT ORIENTED PROGRAMMING

As the name suggests, Object-Oriented Programming or OOPs refers to languages that use objects in programming, they use objects as a primary source to implement what is to happen in the code. Objects are seen by the viewer or user, performing tasks assigned by you. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Access Modifier: Defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:

- **public:** Accessible in all classes in your application.
- **protected:** Accessible within the package in which it is defined and, in its subclass, **(es) (including subclasses declared outside the package)**.
- **private:** Accessible only within the class in which it is defined.
- **default (declared/defined without using any modifier):** Accessible within the same class and package within which its class is defined.

- **The return type**: The data type of the value returned by the method or void if it does not return a value.
- **Method Name**: The rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list**: Comma-separated list of the input parameters that are defined, preceded by their data type, within the enclosed parentheses. If there are no parameters, you must use empty parentheses ().
- **Exception list**: The exceptions you expect the method to throw. You can specify these exception(s).
- **Method body**: It is the block of code, enclosed between braces, that you need to execute to perform your intended operations.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

1.Classes and Objects in Java:

In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities. The class represents a group of objects having similar properties and behaviour.

For example, the animal type **Dog** is a class while a particular dog named **Tommy** is an object of the **Dog** class.

In this article, we will discuss Java objects and classes and how to implement them in our program.

Java Classes

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.

2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
    - Data member
    - Method
    - Constructor
    - Nested Class
    - Interface

Class Declaration in Java

access_modifier **class** <*class_name*>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}

Example of Java Class

```
// Java program to Illustrate Creation of Object
// Using new Instance
// Main class
class GFG {

    // Declaring and initializing string
    String name = "GeeksForGeeks";

    // Main driver method
    public static void main(String[] args)
    {
        // Try block to check for exceptions
        try {
            Class cls = Class.forName("GFG");
            // Creating object of main class
            // using instance method
            GFG obj = (GFG)cls.newInstance();
            // Print and display
            System.out.println(obj.name);
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch (InstantiationException e) {

            e.printStackTrace();
        }
        catch (IllegalAccessException e) {

            e.printStackTrace();
        }
    }
}
```

Output

```
0
null
```

Components of Java Classes

In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access (Refer this for details).
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body is surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

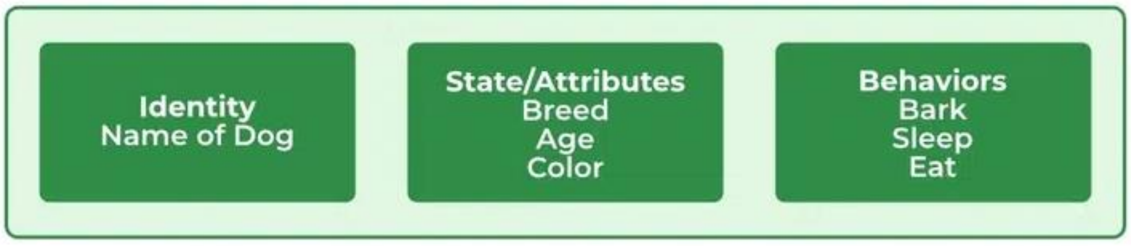
There are various types of classes that are used in real-time applications such as nested classes, anonymous classes, and lambda expressions.

Java Objects:

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. State: It is represented by attributes of an object. It also reflects the properties of an object.
2. Behavior: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. Identity: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", and "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.
Example:

As we declare variables like (type name;). This notifies the compiler that we will use the name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variables , the type must be strictly a concrete class name. In general, we **can't** create objects of an abstract class or an interface.

Dog tuffy;

If we declare a reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

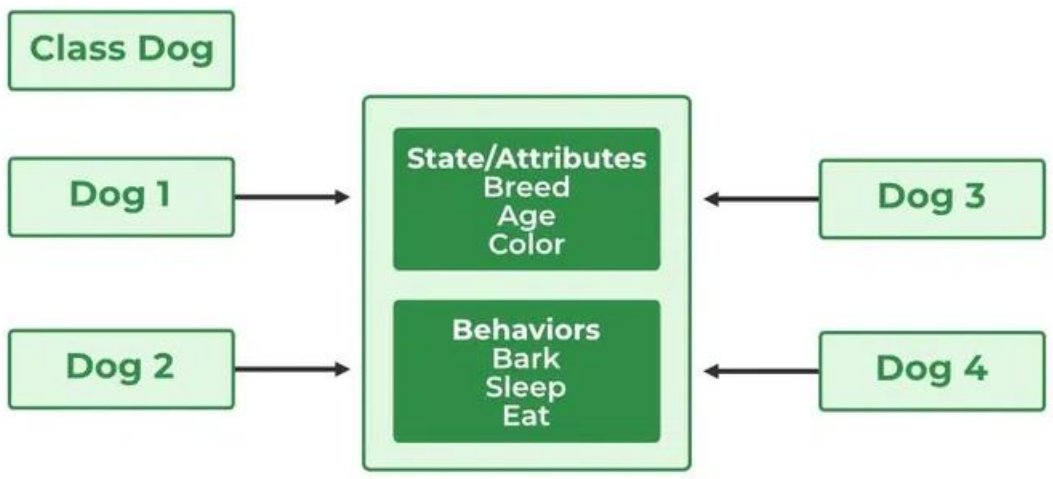
Initializing a Java object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

```
// Class Declaration

public class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed, int age,
            String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName() { return name; }

    // method 2
    public String getBreed() { return breed; }

    // method 3
    public int getAge() { return age; }

    // method 4
    public String getColor() { return color; }

    @Override public String toString()
    {
        return ("Hi my name is " + this.getName()
                + ".\nMy breed,age and color are "
                + this.getBreed() + "," + this.getAge()
                + "," + this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy
            = new Dog("tuffy", "papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Output

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```



2. Constructors:

In Java, Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Example

```
// Java Program to demonstrate
// Constructor
import java.io.*;
class Geeks {
    // Constructor
    Geeks()
    {
        super();
        System.out.println("Constructor Called");
    }

    // main function
    public static void main(String[] args)
    {
        Geeks geek = new Geeks();
    }
}
```

Output Constructor Called

Types of Constructors in Java

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

1. Default Constructor in Java

```
// Java Program to demonstrate
// Default Constructor
import java.io.*;

// Driver class
class GFG {

    // Default Constructor
    GFG() { System.out.println("Default constructor"); }

    // Driver function
    public static void main(String[] args)
    {
        GFG hello = new GFG();
    }
}
```

Output
Default constructor

2. Parameterized Constructor in Java

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.

Example:

```
// Java Program for Parameterized Constructor
import java.io.*;
class Geek {
    // data members of the class.
    String name;
    int id;
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Geek geek1 = new Geek("avinash", 68);
        System.out.println("GeekName :" + geek1.name
                                + " and GeekId :" + geek1.id);
    }
}
```

Output

```
GeekName :avinash and GeekId :68
```

3. Copy Constructor in Java:

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:

```
// Java Program for Copy Constructor
import java.io.*;

class Geek {
    // data members of the class.
    String name;
    int id;

    // Parameterized Constructor
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Geek(Geek obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        System.out.println("First Object");
        Geek geek1 = new Geek("avinash", 68);
        System.out.println("GeekName :" + geek1.name
                                + " and GeekId :" + geek1.id);

        System.out.println();

        // This would invoke the copy constructor.
        Geek geek2 = new Geek(geek1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("GeekName :" + geek2.name
                                + " and GeekId :" + geek2.id);
    }
}
```

Output :

```
First Object
GeekName :avinash and GeekId :68
Copy Constructor used Second Object
GeekName :avinash and GeekId :68
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Output

```
111 Karan 0
222 Aryan 25
```

*****Pillars Of OOP's**

3. Inheritance:

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

Syntax :

```
class derived-class extends base-class
{
   //methods and fields
}
```

Inheritance in Java Example

```
// Java program to illustrate the
// concept of inheritance

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

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

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

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

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

```
// derived class
class MountainBike extends Bicycle {

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

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

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

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

// driver class
public class Test {
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Output

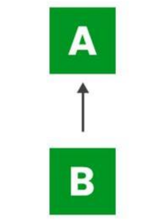
```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

Java Inheritance Types

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1. Single Inheritance:

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Single Inheritance

Example

```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one {
    public void print_for() { System.out.println("for"); }
}

// Driver class
public class Main {
      // Main function
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

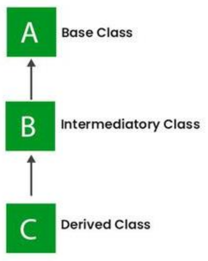
Output

```
Geeks
for
Geeks
```

2. Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class

B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Multilevel Inheritance

Example

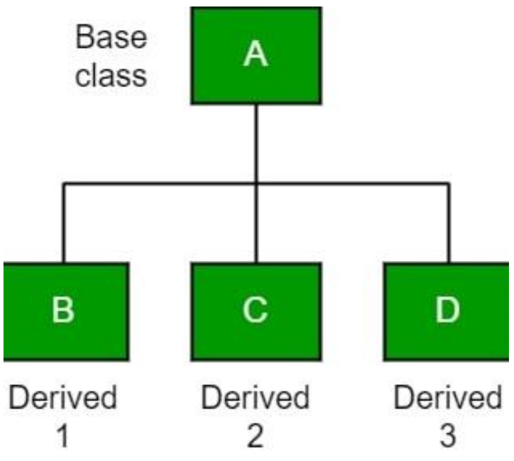
```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class one {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one {
    public void print_for() { System.out.println("for"); }
}

class three extends two {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        three g = new three();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

Output

```
Geeks
for
Geeks
```

3. Hierarchical Inheritance:

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



Example

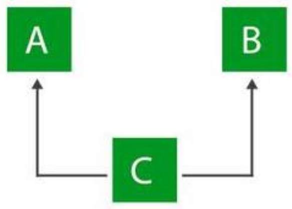
```
// Java program to illustrate the
// concept of Hierarchical inheritance

class A {
    public void print_A() { System.out.println("Class A"); }
}

class B extends A {
    public void print_B() { System.out.println("Class B"); }
}

class C extends A {
    public void print_C() { System.out.println("Class C"); }
}

class D extends A {
    public void print_D() { System.out.println("Class D"); }
}

// Driver Class
public class Test {
    public static void main(String[] args)
    {
        B obj_B = new B();
        obj_B.print_A();
        obj_B.print_B();

        C obj_C = new C();
        obj_C.print_A();
        obj_C.print_C();

        D obj_D = new D();
        obj_D.print_A();
        obj_D.print_D();
    }
}
```

Output

```
Class A
Class B
Class A
Class C
Class A
Class D
```

4. Multiple Inheritance (Through Interfaces):

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



Multiple Inheritance

Example

```
// Java program to illustrate the
// concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

interface one {
    public void print_geek();
}

interface two {
    public void print_for();
}

interface three extends one, two {
    public void print_geek();
}
class child implements three {
    @Override public void print_geek()
    {
        System.out.println("Geeks");
    }

    public void print_for() { System.out.println("for"); }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        child c = new child();
        c.print_geek();
        c.print_for();
        c.print_geek();
    }
}
```

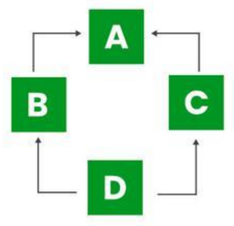
Outputs

```
Geeks
for
Geeks
```

5. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn’t support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Hybrid Inheritance

Example

```java
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}
```

Output

```
true
true
true
```

4. Polymorphism:

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

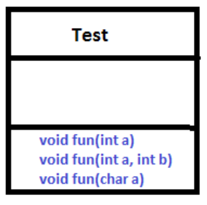
Types of Java polymorphism

In Java polymorphism is mainly divided into two types:

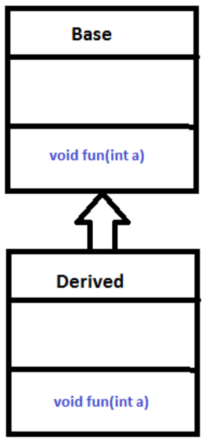
- Compile-time Polymorphism
- Runtime Polymorphism

Compile-Time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.



Overloading



Overriding

Method Overloading

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

Example :

```
// Java Program for Method overloading
// By using Different Types of Arguments

// Class 1
// Helper class
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {

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

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {

        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

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

Output

```
8
34.65
```

Runtime Polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example

```
// Java Program for Method Overriding

// Class 1
// Helper class
class Parent {

    // Method of parent class
    void Print()
    {

        // Print statement
        System.out.println("parent class");
    }
}

// Class 2
// Helper class
```

```
class subclass1 extends Parent {

    // Method
    void Print() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
class subclass2 extends Parent {

    // Method
    void Print()
    {

        // Print statement
        System.out.println("subclass2");
    }
}

// Class 4
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of class 1
        Parent a;

        // Now we will be calling print methods
        // inside main() method

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output

```
subclass1
subclass2
```

5. Encapsulation:

Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

Here's an example of encapsulation:

```
class Person {
    private String name;
    private int age;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }

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

public class Main {
    public static void main(String[] args)
    {
        Person person = new Person();
        person.setName("John");
        person.setAge(30);

        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

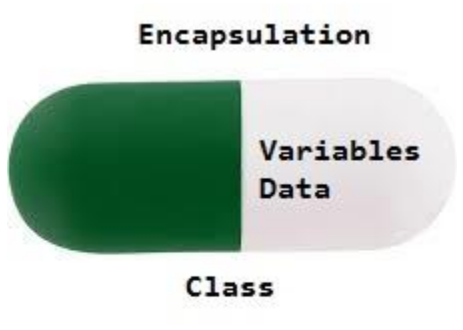
Output

```
Name: John
Age: 30
```

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

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.

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



6. Abstraction:

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Java Abstract classes and Java Abstract methods

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

Algorithm to implement abstraction in Java

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviors and properties of these classes.
3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

Java Abstraction Example

```
// Java program to illustrate the
// concept of Abstraction
abstract class Shape {
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have the constructor
    public Shape(String color)
    {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() { return color; }
}
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius)
    {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override public String toString()
    {
        return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}
class Rectangle extends Shape {

    double length;
    double width;

    public Rectangle(String color, double length,
                    double width)
    {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override double area() { return length * width; }

    @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}
public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellow and area is : 8.0
```

7. <u>Interface:</u>

The interface in Java is *a* mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and multiple inheritances in Java using Interface. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**.

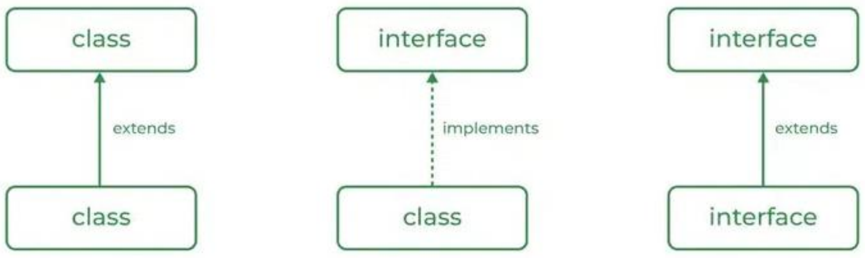
When we decide on a type of entity by its behavior and not via attribute we should define it as an interface.

Syntax for Java Interfaces

```
interface {

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.



Example

```
class Main
{
    public static void main(String[] args)
    {
        Binami b=new Binami();
        b.property();
        b.work();
    }
}
interface chandrababu
{
    public void work();
}
interface jagan
{
    public void property();
}
class Binami implements chandrababu,jagan
{
    public void property()
    {
        System.out.println("Land/house/care,.....");
    }
    public void work()
    {
        System.out.println("Prajaseva");
    }
}
```

Output

```
Land/house/care,.....
Prajaseva
```

Example Programs

```
import java.util.*;
public class calculator
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number a");
        Float a=sc.nextFloat();
        System.out.println("Enter b number");
        Float b=sc.nextFloat();
        sub n1=new sub();
        mul n2=new mul();
        div n3=new div();
        mod n4=new mod();
        n1.Add(a, b);
        n1.Sub(a,b);
        n2.Mul(a,b);
        n3.Div(a,b);
        n4.Mod(a,b);
        sc.close();
    }
}
class sum
{
    void Add(Float a,Float b)
    {
        System.out.println("Addition of two numbers is "+(a+b));
    }
}
class sub extends sum
{
    void Sub(Float a,Float b)
    {
        System.out.println("Substraction of two number is "+(a-b));
    }
}
class mul extends sub
{
    void Mul(Float a,Float b)
    {
        System.out.println("Multiplication of two number is "+(a*b));
    }
}
class div extends mul
{
    void Div(Float a,Float b)
    {
        System.out.println("Division of two numbers is "+(a/b));
    }
}
class mod extends div
{
    void Mod(Float a,Float b)
    {
        System.out.println("Modular division is "+(a%b));
    }
}
```

Output

```
Enter number a
5.5
Enter b number
6.5
Addition of two numbers is 12.0
Substraction of two number is -1.0
Multiplication of two number is 35.75
Division of two numbers is 0.84615386
Modular division is 5.5
```

```java
import java.util.Scanner;
abstract class CalcArea
{
    abstract void findRectangle(double l, double b);
    abstract void findSquare(double s);
    abstract void findCircle(double r);
}
class FindArea extends CalcArea
{
    void findRectangle(double l, double b)
    {
        double area = l*b;
        System.out.println("Area of Rectangle: "+area);
    }
    void findSquare(double s)
    {
        double area = s*s;
        System.out.println("Area of Square: "+area);
    }
    void findCircle(double r)
    {
        double area = 3.14*r*r;
        System.out.println("Area of Circle: "+area);
    }
}
public class Area
{
    public static void main(String args[])
    {
        double l, b, r, s;
        FindArea area = new FindArea();
        Scanner get = new Scanner(System.in);
        System.out.print("\nEnter Length & Breadth of Rectangle: ");
        l = get.nextDouble();
        b = get.nextDouble();
        area.findRectangle(l, b);
        System.out.print("\nEnter Side of a Square: ");
        s = get.nextDouble();
        area.findSquare(s);
        System.out.print("\nEnter Radius of Circle: ");
        r = get.nextDouble();
        area.findCircle(r);
        get.close();
    }
}
```
Output
```
Enter Length & Breadth of Rectangle: 6
4
Area of Rectangle: 24.0
Enter Side of a Square: 5
Area of Square: 25.0
Enter Radius of Circle: 2
Area of Circle: 12.56
```

CHAPETR-18: EXCEPTION HANDLING

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

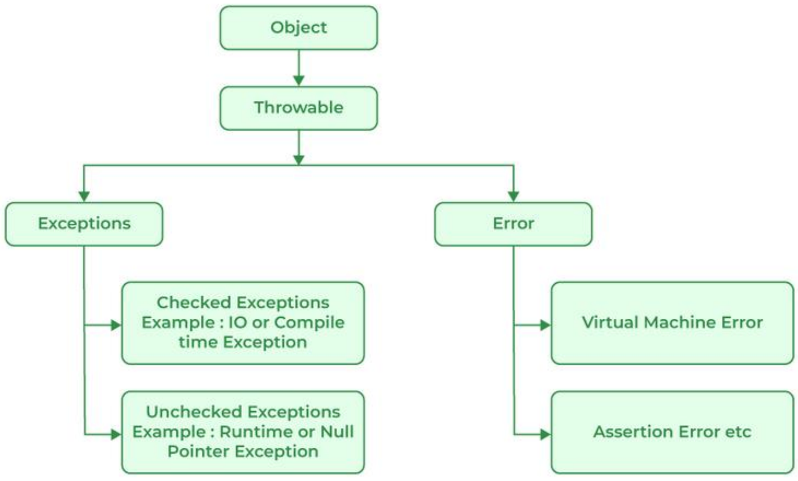
Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
    - Checked Exception
    - Unchecked Exception

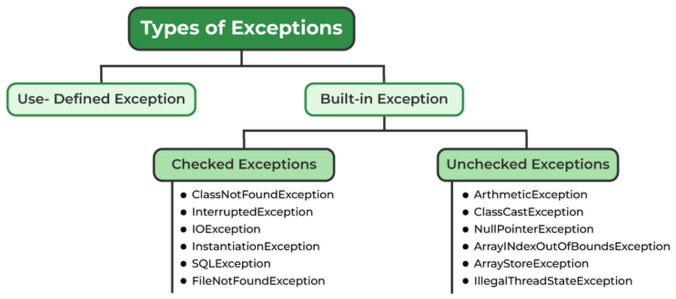
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.



2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

Probabilities of try catch block

1.Single time try catch block

2.Multiple Type Try catch block

3.Try with Multiple catch block

4.Nested Try catch block

5. Catch with try catch block.

1.Single time try catch block

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Program

```
public class Exception
{
    public static void main(String[] args)
    {
        System.out.println("Vasvi");
        System.out.println("Engineering ");
        int[] a={1,2,3};
        try
        {
            System.out.println(a[3]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("exception "+e);
        }
    }
}
```

Output

```
Vasvi
Engineering
exception java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
PS C:\Users\yeswa\Desktop\SOC\Day 10>
```

2.Multiple Type Try catch block

```
public class mutlipletype
{
    public static void main(String[] args)
    {
        System.out.println("Vasvi");
        System.out.println("Engineering ");
        int[] a={1,2,3};
        try
        {
            System.out.println(a[3]);

        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("exception "+e);
        }
        try
        {
            System.out.println(10/0);
        }
        catch(ArithmeticException d)
        {
            System.out.println("Exception "+d);
        }
        System.out.println("college");
    }
}
```

Outputs

```
Vasvi
Engineering
exception java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
Exception java.lang.ArithmeticException: / by zero
college
```

3.Try with Multiple catch block

```
public class trywithmulcatch
{
    public static void main(String[] args)
    {
        System.out.println("Vasvi");
        System.out.println("Engineering ");
        int[] a={1,2,3};
        try
        {
            System.out.println(a[3]);
            System.out.println(10/0);

        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("exception "+e);
        }
        catch(ArithmeticException d)
        {
            System.out.println("Exception "+d);
        }
        System.out.println("college");
    }
}
```

Output:

```
Vasvi
Engineering
exception java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
college
```

4.**Nested Try Catch Block**:

```
public class nestedcatch
{
    public static void main(String[] args)
    {
        System.out.println("Vasvi");
        System.out.println("Engineering ");
        int[] a={1,2,3};
        try
        {

            try
            {
                System.out.println(10/0);
            }
            catch(ArithmeticException d)
            {
                System.out.println("Exception "+d);
            }
            System.out.println(a[3]);

        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("exception "+e);
        }
        System.out.println("college");
    }
}
```

Output:

```
Vasvi
Engineering
Exception java.lang.ArithmeticException: / by zero
exception java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
college
```

5. **Catch with try catch block:**

```
public class catchwithtrycatch
{
    public static void main(String[] args)
    {
        System.out.println("Vasvi");
        System.out.println("Engineering ");
        int[] a={1,2,3};
        try
        {
            System.out.println(a[3]);

        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("exception "+e);
            try
            {
                System.out.println(10/0);
            }
            catch(ArithmeticException d)
            {
                System.out.println("Exception "+d);
            }
        }

        System.out.println("college");

    }
}
```

Output:

```
Vasvi
Engineering
exception java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
Exception java.lang.ArithmeticException: / by zero
College
```

Finally Block :

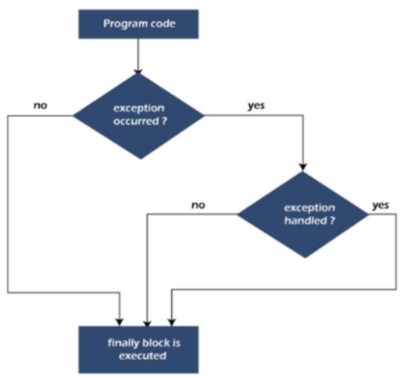
Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not.

Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block

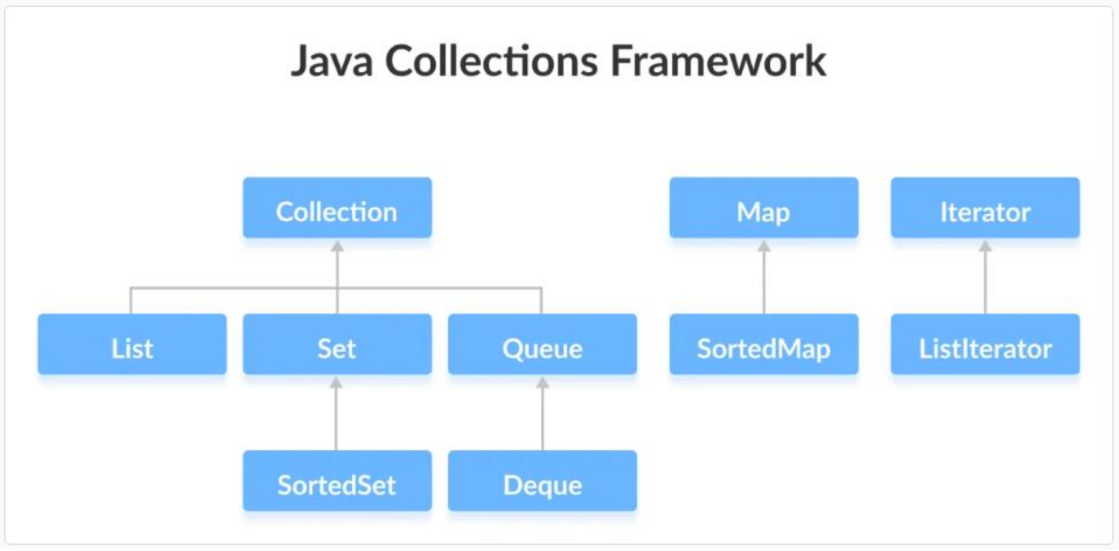


CHAPETR-19: COLLECTIONS

The Java **collections** framework provides a set of interfaces and classes to implement various data structures and algorithms.
For example, the LinkedList class of the collections framework provides the implementation of the doubly-linked list data structure.

Interfaces of Collections FrameWork

The Java collections framework provides various interfaces. These interfaces include several methods to perform different operations on collections.



- Dynamic Array for storing homogenous elements.
- There is no size limit.
- It allows duplicate and maintains insertion order.
- Adding or removing elements at any time.
- Present in java.util. packages.

1. ArrayList:

ArrayList is a Java class implemented using the List interface. Java ArrayList, as the name suggests, provides the functionality of a dynamic array where the size is not fixed as an array. Also as a part of the Collection framework, it has many features not available with arrays.

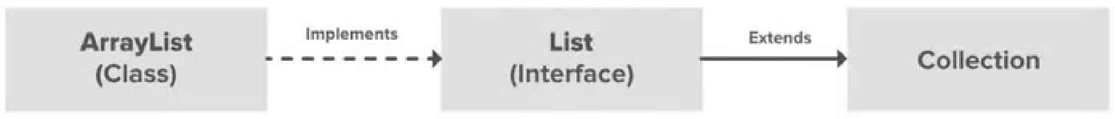
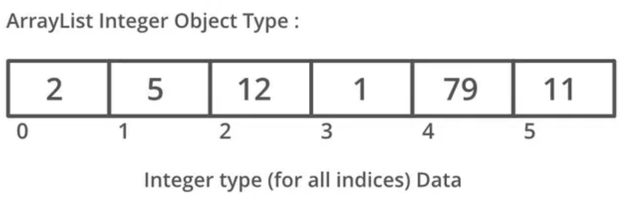


Illustration:


ArrayList Integer Object Type :
2
5
12
1
79
11
0
1
2
3
4
5
Integer type (for all indices) Data

Methods for ArrayList

Method	Method Prototype	Method Description
Add	boolean add(E e)	Adds given element e to the end of the list.
	void add(int index, E element)	Adds given element 'element' at the specified position 'index'.
AddAll	boolean addAll (Collection c)	Adds all the elements in the given collection c to the end of the list.
	boolean addAll (int index, Collection c)	Adds all the elements in the given collection c at the position specified by the 'index' in the list.
Clear	void clear()	Clears the list by removing all the elements from the list.
Clone	Object clone()	Makes a shallow copy of the given ArrayList.
Contains	boolean contains(Object o)	Checks if the list contains the given element 'o'. Returns true if the element is present.
ensureCapacity	void ensureCapacity (int minCapacity)	Increases the capacity of the ArrayList to ensure it has the minCapacity.
Get	E get(int index)	Returns the element in the list present at the position specified by 'index'.
indexOf	int indexOf(Object o)	Returns the index of the first occurrence of element o in the list. -1 if element o is not present in the list.
isEmpty	boolean isEmpty()	Checks if the given list is empty.
Iterator	Iterator iterator()	Returns an iterator to traverse over the list elements in the proper sequence.
lastIndexOf	int lastIndexOf(Object o)	Returns the index of the last occurrence of the specified element o in the list. -1 if the element is not present in the list.
listIterator	ListIterator< E > listIterator()	Returns list iterator to traverse over the elements of the given list.
	ListIterator< E > listIterator(int index)	Returns the list iterator starting from the specified position 'index' to traverse over the elements of the given list.
remove	E remove(int index)	Deletes element at the 'index' in the ArrayList.
	boolean remove(Object o)	Deletes the first occurrence of element o from the list.
removeAll	boolean removeAll(Collection< ? > c)	Removes all the elements from the list that match the elements in given collection c.
removeRange	protected void removeRange (int fromIndex, int toIndex)	Removes elements specified in the given range, fromIndex (inclusive) to toIndex (exclusive) from the list.
retainAll	boolean retainAll(Collection< ? > c)	Retains those elements in the list that match the elements in the given collection c.
set	E set(int index, E element)	Sets the element value at given 'index' to the new value given by 'element'.

size	int size()	Returns the total number of elements or length of the list.
subList	List< E > subList(int fromIndex, int toIndex)	Returns a subList between given range, fromIndex to toIndex for the given list.
toArray	Object[] toArray()	Converts the given list into an array.
	< T > T[] toArray(T[] a)	Converts the given list into an array of the type given by a.
trimToSize	void trimToSize()	Trims the ArrayList capacity to the size or number of elements present in the list.

Example

```
import java.util.*;
public class union
{
    public static void main(String[] args)
    {
        ArrayList<Integer> al=new ArrayList<>();
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter range :");
        int n=sc.nextInt();
        System.out.println("Enter array list 1 elements :");
        for(int i=0;i<n;i++)
        {
            int ele=sc.nextInt();
            al.add(ele);
        }
        ArrayList<Integer> al1=new ArrayList<>();
        System.out.println("Enter array list 2 elements :");
        for(int i=0;i<n;i++)
        {
            int ele=sc.nextInt();
            al1.add(ele);
        }
        int length=al.size();
        ArrayList<Integer> al2=new ArrayList<>();
        al2.addAll(al);
        for(int i=0;i<length;i++)
        {
            for(int j=0;j<length;j++)
            {
                if(al.get(i) == al1.get(j))
                {
                    al1.set(j,0);
                }
            }
            if(al1.get(i)!=0)
            {
                al2.add(al1.get(i));
            }
        }
        System.out.println("Union of two array list al and al1 : "+al2);
        sc.close();
    }
}
```

Output

```
Enter range :
5
Enter array list 1 elements :
1
2
3
4
5
Enter array list 2 elements :
4
5
6
7
8
Union of two array list al and al1 : [1, 2, 3, 4, 5, 4, 5, 6, 7, 8]
```

2. <u>Linked List:</u>

Linked List is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

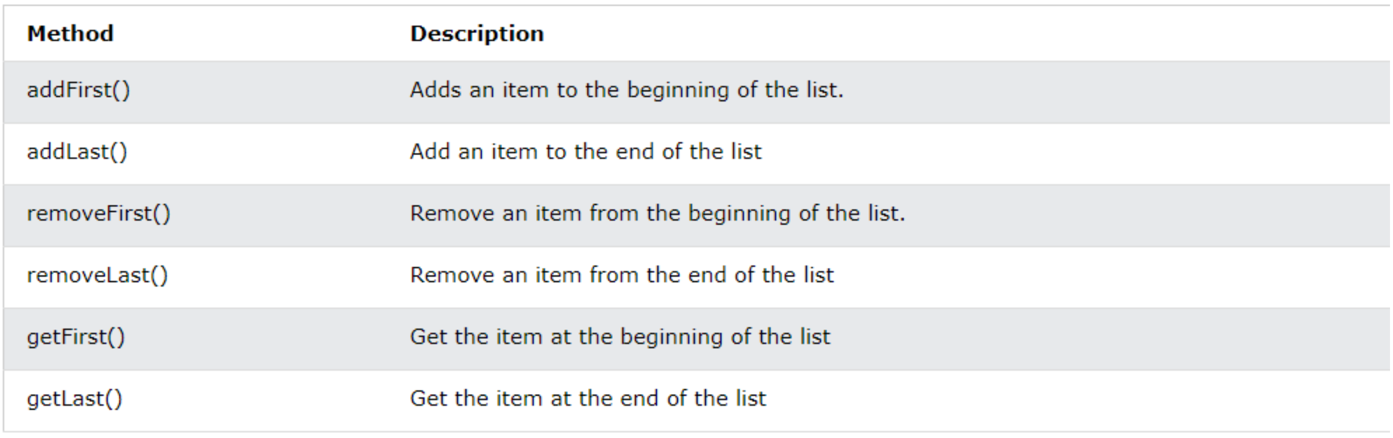
Constructors in the LinkedList:

1. **LinkedList():** This constructor is used to create an empty linked list. If we wish to create an empty LinkedList with the name ll, then, it can be created as:

LinkedList ll = new LinkedList();

2. **LinkedList(Collection C):** This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection's iterator. If we wish to create a LinkedList with the name ll, then, it can be created as:

LinkedList ll = new LinkedList(C);



Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

Example
```
import java.util.*;
public class Linkedlist1
{
    public static void main(String[] args)
    {
        LinkedList<String> al=new LinkedList<String>();
        al.add("C");
        al.add("C++");
        al.add("Java");
        al.add("Python");
        System.out.println("Linked list is "+al);
        System.out.println("Add element in first index of al linkedlist :");
        al.addFirst("Julia");
        System.out.println("linked list al is "+al);
        System.out.println("add element in last index of al linked list");
        al.addLast("Peyga");
        System.out.println("al linked list after add element at last "+al);
        LinkedList<String> l1=new LinkedList<String>();
        l1.add("My SQL");
        l1.add("Access");
        l1.add("Oracle");
        System.out.println(l1);
        al.addAll(l1);
        System.out.println(al);
        System.out.println(l1.contains("Oracle"));
        System.out.println("Element at inedx 1"+l1.get(1));
        System.out.println("peek first element in l1 linked list "+l1.peekFirst());
        System.out.println("Remove first element in al linkedlist : "+al.removeFirst()+"
"+al);
        System.out.println("Remove first element in al linkedlist : "+al.removeLast()+"
"+al);
    }
}
```

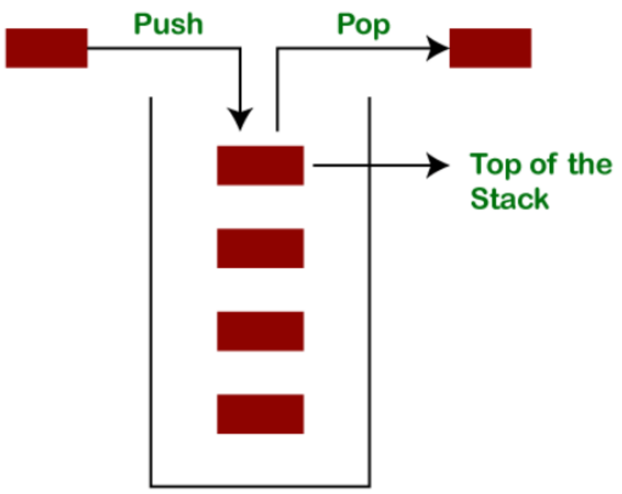
Output
```
Linked list is [C, C++, Java, Python]
Add element in first index of al linkedlist :
linked list al is [Julia, C, C++, Java, Python]
add element in last index of al linked list
al linked list after add element at last [Julia, C, C++, Java, Python, Peyga]
[My SQL, Access, Oracle]
[Julia, C, C++, Java, Python, Peyga, My SQL, Access, Oracle]
true
Element at inedx 1Access
peek first element in l1 linked list My SQL
Remove first element in al linkedlist : Julia [C, C++, Java, Python, Peyga, My SQL, Access, Oracle]
Remove first element in al linkedlist : Oracle [C, C++, Java, Python, Peyga, My SQL, Access]
```

3. STACKS

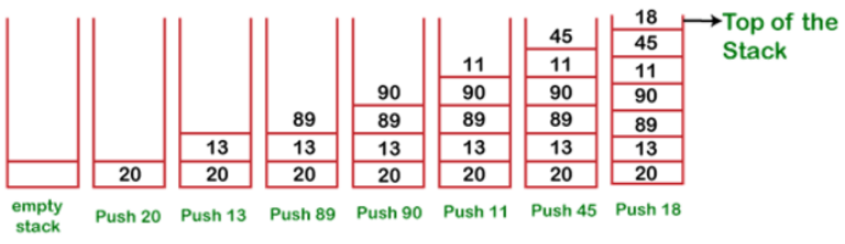
The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the Stack class that provides different operations such as push, pop, search, etc.

In this section, we will discuss the Java Stack class, its methods, and implement the stack data structure in a Java program. But before moving to the Java Stack class have a quick view of how the stack works.

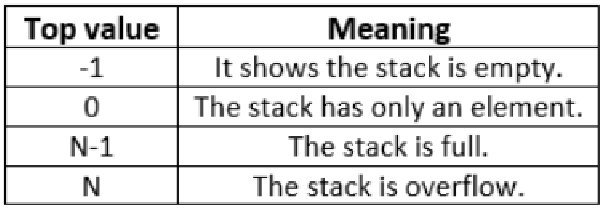
The stack data structure has the two most important operations that are push and pop. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



The following table shows the different values of the top.



Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

Methods of the Stack Class

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Example

```
import java.util.*;
public class stacklist
{
    public static void main(String[] args)
    {
        Stack<Integer> sl=new Stack<Integer>();
        sl.push(25);
        sl.push(76);
        sl.push(74);
        sl.push(100);
        System.out.println(sl);
        if(sl.isEmpty())
        {
            System.out.println("Stack is empty");
        }
        else
        {
            System.out.println(sl);
        }
        sl.pop();
        sl.pop();
        sl.push(200);
        sl.push(20);
        sl.push(10);
        System.out.println(sl);
        sl.push(55);
        sl.push(25);
        System.out.println(sl);
        System.out.println("peek element in given stack is "+sl.peek());
        System.out.println("index of the given element in stack is "+sl.search(200));
    }
}
```

Output

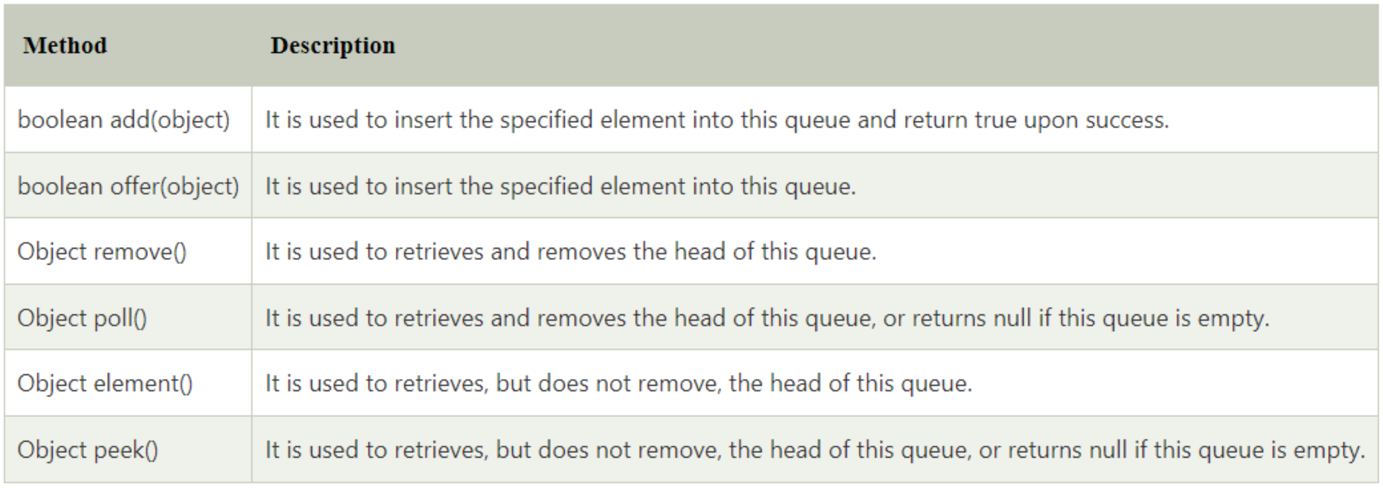
```
[25, 76, 74, 100]
[25, 76, 74, 100]
[25, 76, 200, 20, 10]
[25, 76, 200, 20, 10, 55, 25]
peek element in given stack is 25
index of the given element in stack is 5
```

4. QUEUES:

The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.

Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java. Note that neither of these implementations is thread-safe. PriorityBlockingQueue is one alternative implementation if the thread-safe implementation is needed.

Methods of Java Queue Interface



Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Example

```java
import java.util.*;
public class queue
{
    public static void main(String[] args)
    {
        Deque<String> q=new LinkedList<String>();
        q.add("Vivo");
        q.add("Redmi");
        q.add("Realme");
        q.add("Apple");
        q.offer("Kechoda");
        System.out.println("Given Queue is "+q);
        System.out.println("Peek element of queue is "+q.peek());
        q.remove("Vivo");
        System.out.println("Queue after removing element is "+q);
        System.out.println("Size of the queue is "+q.size());
        q.poll();
        System.out.println("Given Queue :"+q);
        q.offerLast("Samsung");
        System.out.println("After offer last the element the queue is "+q);
    }
}
```

Output :

```
Given Queue is [Vivo, Redmi, Realme, Apple, Kechoda]
Peek element of queue is Vivo
Queue after removing element is [Redmi, Realme, Apple, Kechoda]
Size of the queue is 4
Given Queue :[Realme, Apple, Kechoda]
After offer last the element the queue is [Realme, Apple, Kechoda, Samsung]
```

5. HASHMAP:

The HashMap class of the Java collections framework provides the functionality of the hash table data structure.

It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map.

The HashMap class implements the Map interface.

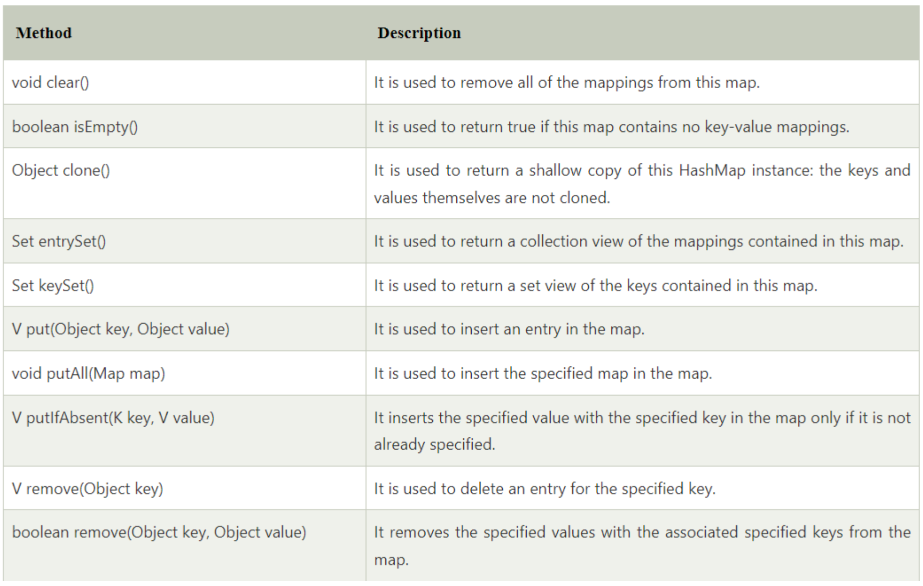
Create a HashMap

In order to create a hash map, we must import the java.util.HashMap package first. Once we import the package, here is how we can create hashmaps in Java.

```
// hashMap creation with 8 capacity and 0.6 load factor
HashMap<K, V> numbers = new HashMap<>();
```

In the above code, we have created a hashmap named numbers. Here, K represents the key type and V represents the type of values.

Methods of Java HashMap class



Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.

Example

```
import java.util.*;

public class hashmap
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> studentmarks = new HashMap<>();
        studentmarks.put("yeswanth",99);
        studentmarks.put("ram ",91);
        studentmarks.put("pawan kalyan",90);
        studentmarks.put("pawan kalyan",100);
        System.out.println(studentmarks.get("pawan kalyan"));
        System.out.println("No.of students :"+studentmarks.size());
        System.out.println(studentmarks.containsKey("yeswanth"));
        for(Map.Entry<String, Integer> m:studentmarks.entrySet())
        {
            System.out.println("{ "+m.getKey()+" : "+m.getValue()+" }");
        }
    }
}
```

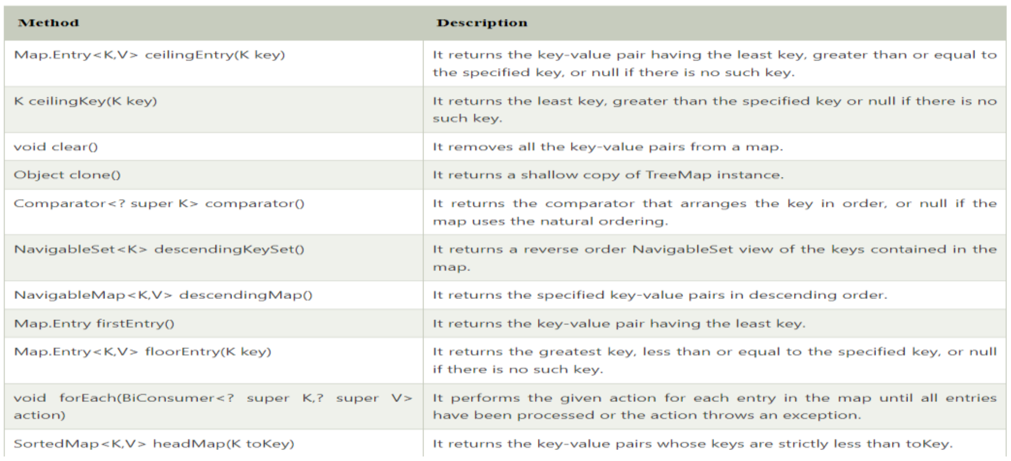
Output

```
100
No.of students :3
true
{ ram  : 91 }
{ pawan kalyan : 100 }
{ yeswanth : 99 }
```

6. TMREEMAP:

The TreeMap in Java is a concrete implementation of the java.util.SortedMap interface. It provides an ordered collection of key-value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.

A TreeMap is implemented using a Red-Black tree, which is a type of self-balancing binary search tree. This provides efficient performance for common operations such as adding, removing, and retrieving elements, with an average time complexity of O(log n).



Method	Description
Map.Entry<K,V> ceilingEntry(K key)	It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key.
K ceilingKey(K key)	It returns the least key, greater than the specified key or null if there is no such key.
void clear()	It removes all the key-value pairs from a map.
Object clone()	It returns a shallow copy of TreeMap instance.
Comparator<? super K> comparator()	It returns the comparator that arranges the key in order, or null if the map uses the natural ordering.
NavigableSet<K> descendingKeySet()	It returns a reverse order NavigableSet view of the keys contained in the map.
NavigableMap<K,V> descendingMap()	It returns the specified key-value pairs in descending order.
Map.Entry firstEntry()	It returns the key-value pair having the least key.
Map.Entry<K,V> floorEntry(K key)	It returns the greatest key, less than or equal to the specified key, or null if there is no such key.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
SortedMap<K,V> headMap(K toKey)	It returns the key-value pairs whose keys are strictly less than toKey.

Example:

```
import java.util.*;

public class tree
{
    public static void main(String[] args)
    {
        TreeMap<Integer,Integer> n=new TreeMap<>();
        n.put(222,75);
        n.put(225,76);
        n.put(221,77);
        n.put(223,78);
        n.put(223,79);
        System.out.println(n.get(221));
        System.out.println("No .of units :"+n.size());
        System.out.println(n.containsKey(224));
        for(Map.Entry<Integer, Integer> m:n.entrySet())
        {
            System.out.println("{ "+m.getKey()+" : "+m.getValue()+" }");
        }
    }
}
```

Output:

```
77
No .of units :4
false
{ 221 : 77 }
{ 222 : 75 }
{ 223 : 79 }
{ 225 : 76 }
```

7. HashSet():

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.

Example

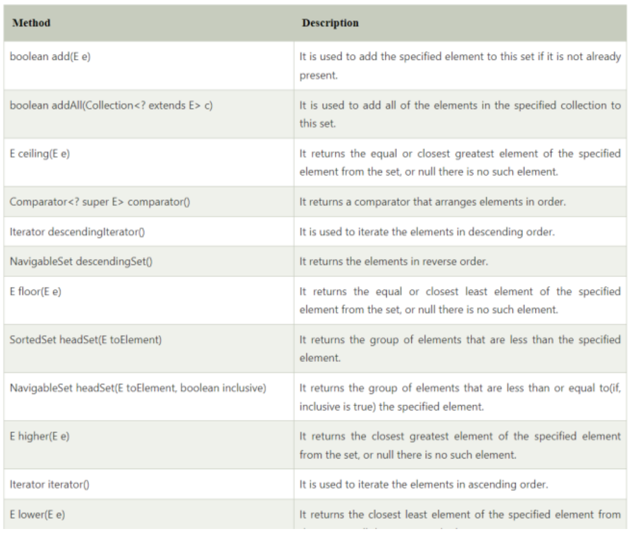
```
import  java.util.*;

public class hashset
{
    public static void main(String[] args)
    {
        HashSet<String> hs=new HashSet<>();
        hs.add("Vivo");
        hs.add("Oppo");
        hs.add("Samsung");
        hs.add("Iphone");
        hs.add("Redmi");
        hs.add("Iphone");
        System.out.println("Set is "+hs);
        System.out.println("No.of units present in set "+hs.size());
        hs.remove("Oppo");
        System.out.println("Set after removing element is "+hs);
        System.out.println(hs.clone());
        Iterator<String> i=hs.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
        hs.clear();
        System.out.println("Set is "+hs);

    }
}
```

Output:

```
Set is [Vivo, Oppo, Iphone, Samsung, Redmi]
No.of units present in set 5
Set after removing element is [Vivo, Iphone, Samsung, Redmi]
[Iphone, Samsung, Vivo, Redmi]
Vivo
Iphone
Samsung
Redmi
Set is []
```

8. TreeSet()

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.



Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns a comparator that arranges elements in order.
Iterator descendingIterator()	It is used to iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from

Example

```
import java.util.*;
public class treeset
{
    public static void main(String[] args)
    {
        TreeSet<Integer> hs=new TreeSet<>();
        hs.add(12);
        hs.add(85);
        hs.add(9);
        hs.add(54);
        hs.add(4);
        hs.add(11);
        System.out.println("Set is "+hs);
        System.out.println("No.of units present in set "+hs.size());
        hs.remove(12);
        System.out.println("Set after removing element is "+hs);
        System.out.println(hs.clone());
        Iterator<Integer> i=hs.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
        hs.clear();
        System.out.println("Set is "+hs);
    }
}
```

Output

```
Set is [4, 9, 11, 12, 54, 85]
No.of units present in set 6
Set after removing element is [4, 9, 11, 54, 85]
[4, 9, 11, 54, 85]
4
9
11
54
85
Set is []
```

CHAPETR-20: FILE HANDLING

In Java, with the help of File Class, we can work with files. This File Class is inside the java.io package. The File class can be used by creating an object of the class and then specifying the name of the file.

Streams in Java

- In Java, a sequence of data is known as a stream.
- This concept is used to perform I/O operations on a file.

There are two types of streams :

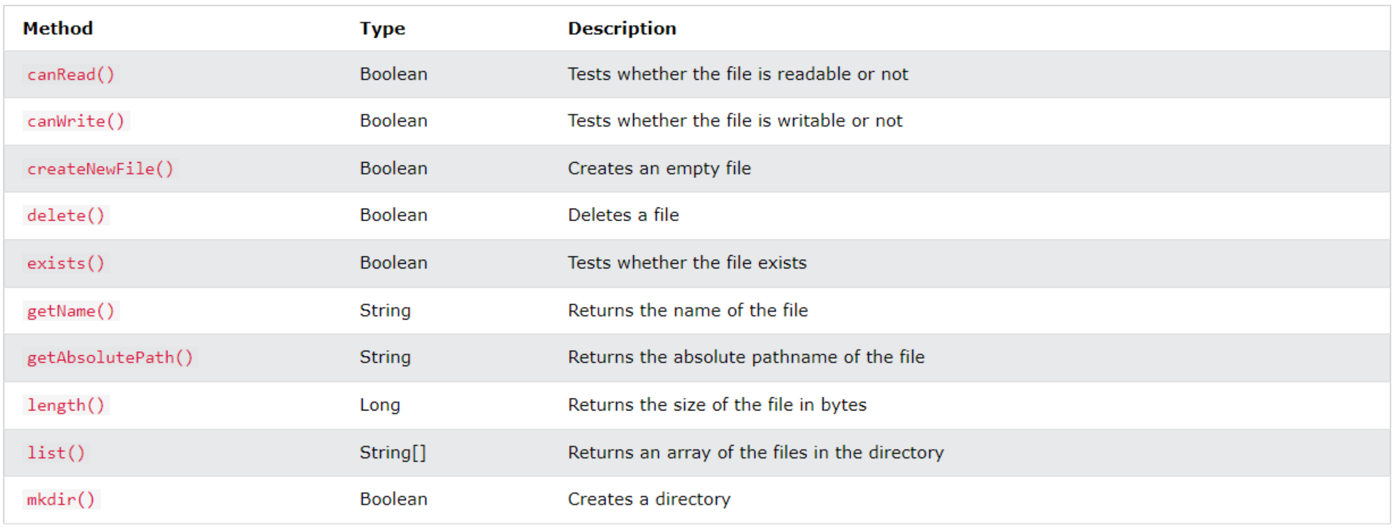
1. Input Stream:

The Java InputStream class is the superclass of all input streams. The input stream is used to read data from numerous input devices like the keyboard, network, etc. InputStream is an abstract class, and because of this, it is not useful by itself. However, its subclasses are used to read data.

There are several subclasses of the InputStream class, which are as follows:

1. AudioInputStream
2. ByteArrayInputStream
3. FileInputStream
4. FilterInputStream
5. StringBufferInputStream
6. ObjectInputStream

The File class has many useful methods for creating and getting information about files. For example:



Method	Type	Description
`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

Example

```
import java.io.File;
import java.io.IOException;
class File1
```

```
{
    public static void main(String[] args)
    {
        try
        {
            File f=new File("Sample1.txt");
            if(f.createNewFile())
            {
                System.out.println("File is created");
            }
            else
            {
                System.out.println("File is already exit");
            }
            if(f.exists())
            {
                System.out.println("File name : "+f.getName());
                System.out.println("File Path :"+f.getAbsolutePath());
                System.out.println(f.canWrite());
                System.out.println(f.canRead());
                System.out.println("File Size :"+f.length());
            }
            else{
                System.out.println("The file doesn't exit.");
            }
        }
        catch(IOException e)
        {
            System.out.println("Excepted occured.");
            e.printStackTrace();
        }
    }
}
```

Output

```
File is already exit
File name : Sample1.txt
File Path :C:\Users\yeswa\Desktop\SOC\Day 14\Sample1.txt
true
true
File Size :0
```

Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

Example

```
import java.io.File;
import java.util.Scanner;
import java.io.FileWriter;
```

```java
import java.io.IOException;

public class File2
{
    public static void main(String[] args)
    {
        try
        {
            File f=new File("eee.txt");
            FileWriter x=new FileWriter(f);
            x.write("The quick browm fox jumps over the lazy dog ");
            x.close();
            Scanner dataReader=new Scanner(f);
            while(dataReader.hasNextLine())
            {
                String fileData=dataReader.nextLine();
                System.out.println(fileData);
            }
            dataReader.close();
            System.out.println("Content is successfully wrote to the file.");
        }
        catch(IOException e)
        {
            System.out.println("Unexcepted occur");
            e.printStackTrace();
        }
    }
}
```

Output

```
The quick browm fox jumps over the lazy dog
Content is successfully wrote to the file.
```

Read a File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter:

Example

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;


public class File3
{
    public static void main(String[] args)
    {
        try
        {
            File f1=new File("eee.txt");
            Scanner dataReader =new Scanner(f1);
            while(dataReader.hasNextLine())
            {
                String fileData = dataReader.nextLine();
                System.out.println(fileData);
            }
            dataReader.close();
        }
        catch(FileNotFoundException exception)
        {
            System.out.println("Unexcepted error occured");
            exception.printStackTrace();
        }
    }
}
```

Output

The quick browm fox jumps over the lazy dog

Delete a File

To delete a file in Java, use the delete() method:

Example

```
import java.io.File;
import java.util.*;

public class File4
{
    public static void main(String[] args)
    {
        File f=new File("eee.txt");
        if(f.delete())
        {
            System.out.println(f.getName()+" is deleetd !");
        }
        else
        {
            System.out.println("Unexcepeted error ");
        }
    }
}
```

Output

eee.txt is deleetd !

Example Programs

```
import java.util.Arrays;
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
```

```java
public class File5
{
    public static void main(String[] args)
    {
        int c=0,i;
        String a="quick";
        try
        {
            File f1=new File("eee.txt");
            Scanner dataReader=new Scanner(f1);
            while(dataReader.hasNextLine())
            {
                String fileData = dataReader.nextLine();

                String[] words =fileData.split("\\s+");
                for(i=0;i<words.length;i++)
                {
                    if(words[i].equalsIgnoreCase(a))
                    {
                        c++;
                    }
                }
                System.out.println(fileData);
            }
            dataReader.close();
        }
        catch(FileNotFoundException exception)
        {
            System.out.println("Unexcepted error occured");
            exception.printStackTrace();

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

Output

```
The quick browm fox jumps over the lazy dog
1
```

21. CONCLUSION

In conclusion, the Java programming language, with its robust features and versatility, remains a cornerstone in the world of software development. Through this skill-oriented course, I have gained a comprehensive understanding of Java's fundamentals, object-oriented principles, and its extensive application in the development of various software solutions.This course has not only enhanced my proficiency in Java programming but has also equipped me with the ability to design and implement complex algorithms, create efficient data structures, and develop scalable applications. The hands-on experience gained through practical exercises and projects has not only fortified my programming skills but has also instilled in me a deep appreciation for the importance of clean, well-documented, and maintainable code.Moving forward, I am confident that the knowledge and expertise acquired during this course will serve as a solid foundation for my future endeavors in the field of software development. With the in-depth understanding of Java, I am prepared to contribute effectively to projects that require innovative solutions and advanced programming techniques. I am eager to apply my skills in real-world scenarios and continue to explore the ever-evolving landscape of Java development, consistently striving for excellence and innovation.

THANK YOU