

Object Oriented Programming

Paper 4

using



JAVA PROGRAMMING

P Veera Venkata Durga PraSad
DEPARTMENT OF COMPUTER SCIENCE (AWDC KKD)

Semester: IV**OBJECT ORIENTED PROGRAMMING USING JAVA****UNIT I:**

Introduction to Java: Features of Java, The Java virtual Machine, Parts of Java

Naming Conventions and Data Types: Naming Conventions in Java, Data Types in Java, Literals. **Operators in Java:** Operators, Priority of Operators.

Control Statements in Java: if... else Statement, do... while Statement, while Loop, for Loop, switch Statement, break Statement, continue Statement, return Statement.

Input and Output: Accepting Input from the Keyboard, Reading Input with Java.util.Scanner Class, Displaying Output with System.out.printf(), Displaying Formatted Output with String.format(). **Arrays:** Types of Arrays, Three Dimensional Arrays (3D array), arrayname. length, Command Line Arguments

UNIT II:

Strings: Creating Strings, String Class Methods, String Comparison, Immutability of Strings. **Introduction to OOPs:** Problems in Procedure Oriented Approach,

Features of Object- Oriented Programming System (OOPS). **Classes and Objects:** Object Creation, Initializing the Instance Variables, Access

Specifiers, Constructors. **Methods in Java:** Method Header or Method Prototype, Method Body, Understanding Methods, Static Methods, Static Block, The keyword 'this', Instance Methods, Passing Primitive Data Types to Methods,

Passing Objects to Methods, Passing Arrays to Methods, Recursion, Factory Methods. **Inheritance:** Inheritance, The keyword 'super', The Protected Specifier, Types of Inheritance.

UNIT III:

Polymorphism: Polymorphism with Variables, Polymorphism using Methods, Polymorphism with Static Methods, Polymorphism with Private Methods, Polymorphism with Final Methods, final Class. **Type Casting:** Types of Data Types, Casting Primitive Data Types, Casting Referenced Data Types, The Object Class.

Abstract Classes: Abstract Method and Abstract Class. **Interfaces:** Interface, Multiple Inheritance using Interfaces. **Packages:** Package, Different Types of Packages, The JAR Files, Interfaces in a Package, Creating Sub Package in a

Package, Access Specifiers in Java, Creating API Document. **Exception Handling:** Errors in Java Program, Exceptions, throws Clause, throw Clause, Types of Exceptions, Re - throwing an Exception.

UNIT - IV

Streams: Stream, Creating a File using FileOutputStream, Reading Data from a File Using FileInputStream, Creating a File using FileWriter, Reading a File using FileReader, Zipping and Unzipping Files, Serialization of Objects, Counting

Number of Characters in a File, File Copy, FileClass

Threads: Single Tasking, Multi Tasking, Uses of Threads, Creating a Thread and Running it, Terminating the Thread, Single Tasking Using a Thread, Multi Tasking Using Threads, MultipleThreads Acting on Single Object, Thread Class Methods, Deadlock of Threads, ThreadCommunication, Thread Priorities, thread Group, Daemon Threads, Applications of Threads, ThreadLife Cycle.

UNIT V:

Applets: Creating an Applet, Uses of Applets, <APPLET> tag, A Simple Applet, An Applet with Swing Components, Animation in Applets, A Simple Game with an Applet, Applet Parameters.

Java Database Connectivity: Database Servers, Database Clients, JDBC (Java DatabaseConnectivity), Working with Oracle Database, Working with MySQL Database, Stages in a JDBCProgram, Registering the Driver, Connecting to a Database, Preparing SQL Statements, Using jdbc-odbc Bridge Driver to Connect to Oracle Database, Retrieving Data from MySQL Database, Retrieving Data from MS Access Database, Stored Procedures and CallableStatements, Types of Result Sets.

UNIT-I

❖ Introduction to Java:

Java is a high-level and purely **object oriented programming language**. It is platform independent, robust, secure, and multithreaded programming language which makes it popular among other OOP languages. It is widely used for software, web, and mobile application development, along with this it is also used in big data analytics and server-side technology. Before moving towards features of Java, let us see how Java originated.

HISTORY:

- In 1990, Sun Microsystems Inc. started developing software for electronic devices. This project was called the Stealth Project (later known as Green Project).
- In 1991, Bill Joy, James Gosling, and Patrick Naughton started working on this project. Gosling decided to use C++ to develop this project, but the main problem he faced is that C++ is platform dependent language and could not work on different electronic device processors.
- As a solution to this problem, Gosling started developing a new language that can be worked over different platforms, and this gave birth to the most popular, platform-independent language known as Oak.
- Yes, you read that right, Oak, this was the first name of Java. But, later it was changed to Java due to copyright issues (some other companies already registered with this name).
- On 23 January 1996, Java's JDK 1.0 version was officially released by Sun Microsystems. This time the latest release of Java is JDK 20.0 (March 2023).
- Now Java is being used in Web applications, Windows applications, enterprise applications, mobile applications, etc. Every new version of Java comes with some new features.

❖ Features of java:

1. Simple:

Java is a simple programming language and easy to understand because it does not contain complexities that exist in prior programming languages. In fact, simplicity was the design aim of Javasoft people, because it has to work on electronic devices where less memory/resources are available. Java contains the same syntax as C, and C++, so the programmers who are switching to Java will not face any problems in terms of syntax.

2. Object-Oriented:

Java is an Object Oriented Programming Language, which means in Java everything is written in terms of classes and objects. Now, what is an Object? The object is

nothing but a real-world entity that can represent any person, place, or thing and can be distinguished from others. Every object near us has some state and behaviour associated with it.

For example, my mobile phone is a real-world entity and has states like colour, model, brand, camera quality, etc, and these properties are represented by variables. Also mobile is associated with actions like, calling, messaging, photography, etc and these actions are represented by methods in Java.

The main concepts of any Object-Oriented Programming language are given below:

- Class and Object
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

3. Platform Independent:

The design objective of javasoft people is to develop a language that must work on any platform. Here platform means a type of operating system and hardware technology. Java allows programmers to write their program on any machine with any configuration and to execute it on any other machine having different configurations.

In Java, Java source code is compiled to bytecode and this bytecode is not bound to any platform. In fact, this bytecode is only understandable by the Java Virtual Machine which is installed in our system. Now programmers write programs only once, compile them, generate the bytecode and then export it anywhere.

4. Portable:

The WORA (Write Once Run Anywhere) concept and platform-independent feature make Java portable. Now using the Java programming language, developers can yield the same result on any machine, by writing code only once. The reason behind this is JVM and bytecode. Suppose you wrote any code in Java, then that code is first converted to equivalent bytecode which is only readable by JVM. We have different versions of JVM for different platforms.

5. Robust:

The Java Programming language is robust, which means it is capable of handling

unexpected termination of a program. There are 2 reasons behind this, first, it has a most important and helpful feature called Exception Handling. If an exception occurs in java code then no harm will happen whereas, in other low-level languages, the program will crash.

Another reason why Java is strong lies in its memory management features. Unlike other low-level languages, Java provides a runtime Garbage collector offered by JVM, which collects all the unused variables

6. Secure:

In today's era, security is a major concern of every application. As of now, every device is connected to each other using the internet and this opens up the possibility of hacking. And our application built using java also needs some sort of security. So Java also provides security features to the programmers. Security problems like virus threats, tampering, eavesdropping, and impersonation can be handled or minimized using Java. Encryption and Decryption feature to secure your data from *eavesdropping* and *tampering* over the internet. An *Impersonation* is an act of pretending to be another person on the internet.

7. Interpreted:

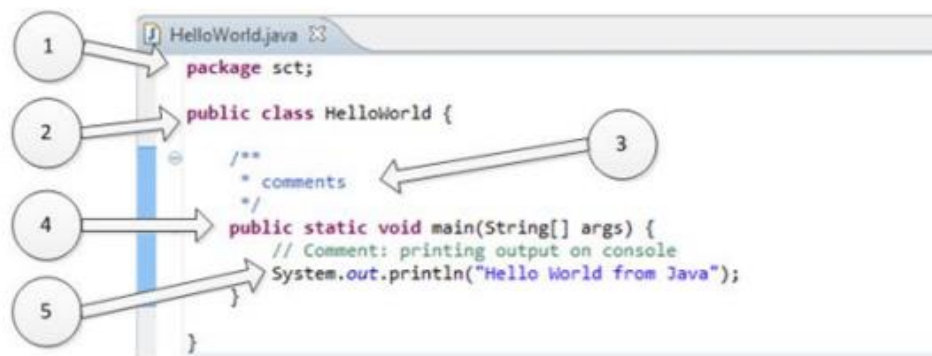
In programming languages, you have learned that they use either the compiler or an interpreter, but Java programming language uses both a compiler and an interpreter. Java programs are compiled to generate bytecode files then JVM interprets the bytecode file during execution. Along with this JVM also uses a JIT compiler (it increases the speed of execution).

8. Multi-Threaded:

Thread is a lightweight and independent subprocess of a running program (i.e, process) that shares resources. And when multiple threads run simultaneously is called multithreading. In many applications, you have seen multiple tasks running simultaneously, for example, Google Docs where while typing text, the spell check and autocorrect tasks are running.

❖ **JAVA PROGRAM STRUCTURE:**

Let's use example of HelloWorld Java program to understand structure and features of class. This program is written on few lines, and its only task is to print "Hello World from Java" on the screen. Refer the following picture.



1. “package sct”:

It is package declaration statement. The package statement defines a name space in which classes are stored. Package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in program. It must be first line of your program or you can omit it.

2. “public class HelloWorld”:

This line has various aspects of java programming.

a. public: This is access modifier keyword which tells compiler access to class.

Various values of access modifiers can be public, protected, private or default (no value).

b. class: This keyword used to declare class. Name of class (HelloWorld) followed by this keyword.

3. Comments section: We can write comments in java in two ways.

a. Line comments: It start with two forward slashes (//) and continue to the end of the current line. Line comments do not require an ending symbol.

b. Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*). Block comments can also extend across as many lines as needed.

4. “public static void main (String []args)”:

Its method (Function) named main with string array as argument.

a. public : Access Modifier

b. static: static is reserved keyword which means that a method is accessible and usable even though no objects of the class exist.

c. void: This keyword declares nothing would be returned from method. Method can return any primitive or object.

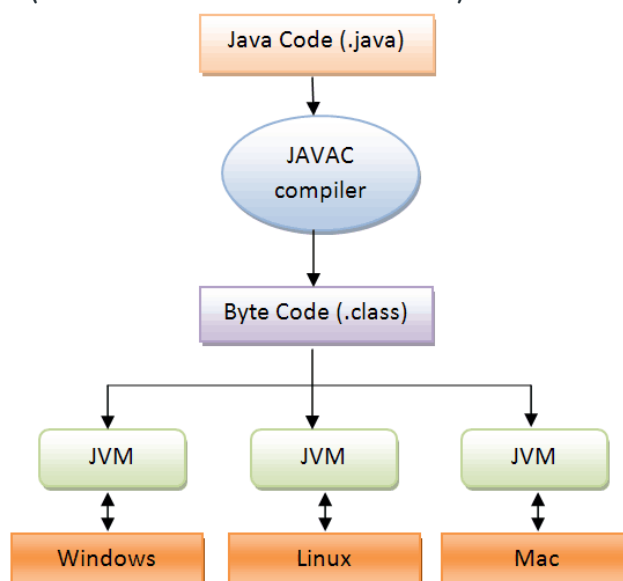
d. Method content inside curly braces. { }

5. System.out.println("Hello World from Java") :

- a. **System**: It is name of Java utility class.
- b. **out**: It is an object which belongs to System class.
- c. **println**: It is utility method name which is used to send any String to console.
- d. **"Hello World from Java"**: It is String literal set as argument to println method.

❖ Java virtual machine:

JVM (Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a Java code. JVM is a part of JRE (Java Runtime Environment).



Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, .class files (contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

❖ Naming Conventions:

- Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
- All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention.

- If you fail to follow these conventions, it may generate confusion or erroneous code.
- These conventions are suggested by several Java communities such as Sun Microsystems and Netscape. Basic naming conventions in Java.
- ✓ **PascalCasing:**
 - PascalCasing is a naming convention where each word in a compound word is capitalized.
 - Class and Interface are characterized into PascalCasing.
 - **Class:**

It should start with UpperCase Letter.

It should be a noun such as Color, Button, System, Thread, MyClass etc.

Example:

```
public class MyClass
{
//code snippet
}
```

- **Interface:**

It should start with the UpperCase Letter.

It should be an adjective such as Runnable, Remote, ActionListener.

Example:

```
interface Printable
{
//code snippet
}
```

- ✓ **camelCasing:**

- In camelCasing, the first word in a compound word is not capitalized.
- Method, variable are characterized into camelCasing.

- **Method:**

It should start with lowercase letter.

It should be a verb such as main(), print(), println().

If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

Example:

```
class Employee
{
//method
void draw()
{
//code snippet
}
```

```
}
```

- **Variable:**

It should start with a lowercase letter such as id,name.

It should not start with the special characters like &(ampersand),

\$(dollar), _(underscore).

If the name contains multiple words,start it with the lowercase letter followed by an uppercase letter such as firstName,lastName.

Example:

```
class Employee
```

```
{
```

```
//variable
```

```
int id;
```

```
//code snippet
```

```
}
```

✓ **snake_casing:**

➤ In snake_casing, underscores separate the words in a compound word.

➤ Packages,constant are characterized into snake_casing.

- **Package:**

It should be a lowercase letter such as java, lang.

If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

Example:

```
//package
```

```
package com.mypackage;
```

```
class Employee
```

```
{
```

```
//code snippet
```

```
}
```

- **Constant:**

It should be in uppercase letters such as RED,YELLOW.

If the name contains multiple words,it should be separated by an underscore(_) such as MAX_PRIORITY.

It may contain digits but not as the first letter.

Example:

```
class Employee
```

```
{
```

```
//constant
```

```
static final int MIN_AGE=18;
```

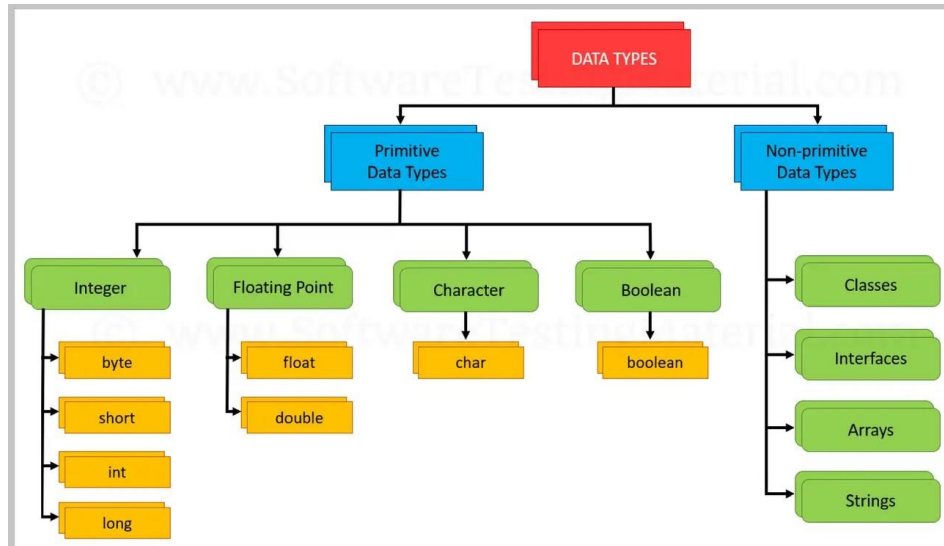
```
//code snippet
```

```
}
```

❖ Datatypes:

Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:



a) **Primitive Data Types:** In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

1. Boolean:

- The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
- The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.
- Default value is false and default size is 1bit.

Example:

Boolean one = **false**

2. Byte:

- The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, byte b = -20

3. Short:

- The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
- The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

```
short s = 10000, short v = -5000
```

4. Int:

- The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.
- The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

```
int a = 100000, int b = -200000
```

5. Long:

- The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$) (inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0.
- The long data type is used when you need a range of values more than those provided by int.

Example:

```
long a = 100000L, long b = -200000L
```

6. Float:

- The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited.
- It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers.
- The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
float f1 = 234.5f
```

7. Double:

- The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited.
- The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

8. Char:

- The char data type is a single 16-bit Unicode character.
- Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example:

```
char letterA = 'A'
```

b) **Non-primitive data types:** Unlike primitive data types, these are not predefined. These are user-defined data types created by programmers. These data types are used to store multiple values.

1.**Class:** A class in Java is a user defined data type i.e. it is created by the user. It acts a template to the data which consists of member variables and methods.

2.**interface:** An interface is similar to a class however the only difference is that its methods are abstract by default i.e. they do not have body. An interface has only the final variables and method declarations. It is also called a fully abstract class.

3.**Array:** An array is a data type which can store multiple homogenous variables i.e., variables of same type in a sequence. They are stored in an indexed manner starting with index 0. The variables can be either primitive or non-primitive data types.

4.**String:** A string represents a sequence of characters for example "Javanotes", "Hello world", etc. String is the class of Java.

❖ Literals:

literals are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.

```
int cost = 340;
```



Types of literals:

There are the majorly following types of literals in Java:

a) Integer Literal:

Integer literals are sequences of digits. There are three types of integer literals:

- ✓ **Decimal integer:** These are the set of numbers that consist of digits from 0 to 9. It may have a positive (+) or negative (-) Note that between numbers commas and non-digit characters are not permitted. For example, **5678**, **+657**, **-89**, etc.
- ✓ **Octal Integer:** It is a combination of number have digits from 0 to 7 with a leading 0. For example, **045**, **027**.
- ✓ **Hexa-Decimal:** The sequence of digits preceded by **0x** or **0X** is considered as hexadecimal integers. It may also include a character from **a** to **f** or **A** to **F** that represents numbers from **10** to **15**, respectively. For example, **0xd**, **0xf**,
- ✓ **Binary Integer:** Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later). Prefix **0b** represents the Binary system. For example, **0b11010**.

b) Character Literal:

A character literal is expressed as a character or an escape sequence, enclosed in a **single** quote (") mark. It is always a type of char. For example, **'a'**, **'%'**, **'\u000d'**, etc.

c) String Literal:

String literal is a sequence of characters that is enclosed between **double** quotes (") marks. It may be alphabet, numbers, special characters, blank space, etc. For example, **"Jack"**, **"12345"**, **"\n"**, etc.

d) Boolean Literal:

Boolean literals are the value that is either true or false. It may also have values 0 and For example, **true**, **0**, etc.

e) Floating Point Literal:

- The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals. Floating-point literals for float type end with **F** or **f**. For example, **6f**, **8.354F**, etc. It is a 32-bit float literal.

- Floating-point literals for double type end with D or d. It is optional to write D or d. For example, 6d, 8.354D, etc. It is a 64-bit double literal.
- It can also be represented in the form of the exponent.

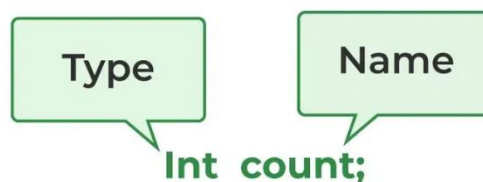
❖ Variables in Java

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

Declare Variables in Java

We can declare variables in Java as pictorially depicted below as a visual aid.



From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

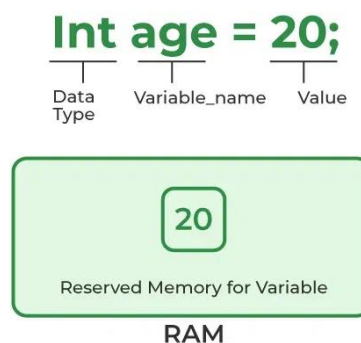
1. **datatype:** Type of data that can be stored in this variable.
2. **data_name:** Name was given to the variable.

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

Initialize Variables in Java

It can be perceived with the help of 3 components that are as follows:



- **datatype:** Type of data that can be stored in this variable.
- **variable_name:** Name given to the variable.
- **value:** It is the initial value stored in the variable.
-

❖ Operators in java:

Operators are used to perform operations on variables and values.

There are many types of operators in Java which are given below:

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

i. **Unary Operator:**

In Java, the **unary operator** is an operator that can be used only with an operand.

It is used to represent the **positive** or **negative** value, **increment/decrement** the value by 1, and **complement** a Boolean value.

Operator name	Symbol	Description	Example	Equivalent Expression
Unary Plus	+	is used to represent the positive value.	+a	a
Unary Minus	-	is used to represent the negative value.	-a	-
Increment Operator	++	increments the value of a variable by 1.	++a or a++	a=a+1
Decrement operator	--	decrements the value of a variable by 1.	--a	a=a-1

			or	
			a--	
Logical Complement Operator	!	Inverts the value of a boolean variable.	!true	-

ii. Arithmetic Operator:

The way we calculate mathematical calculations, in the same way, Java provides **arithmetic operators** for mathematical operations. It provides operators for all necessary mathematical calculations.

Operator	Meaning	Description	Example
+	Addition	Adds two values together	a+b
-	Subtraction	Subtracts one value from another	a-b
*	Multiplication	Multiplies two values	a*b
/	Division	Divides one value by another	a/b
%	Modulus	Returns the division remainder	a%b

iii. Shift Operator:

In Java, **shift operators** are the special type of operators that work on the bits of the data. These operators are used to shift the bits of the numbers from left to right or right to left depending on the type of shift operator used. There are three types of shift operators in Java:

➤ Signed Left Shift Operator (<<):

The signed left shift operator is a special type of operator used to move the bits of the expression to the left according to the number specified after the operator.

➤ Signed Right Shift Operator (>>):

The signed right shift operator is a special type of operator used to move the bits of the expression to the right according to the number specified after the operator.

➤ Unsigned Right Shift Operator (>>>):

The unsigned right shift operator is a special type of right shift operator that does not use the sign bit to fill in the sequence. The unsigned sign shift operator on the right always fills the sequence by 0.

iv. Relational Operator:

Relational operators are used to check the relationship between two operands.

Operator	Description	Example
==	Equal To	5 == 5 returns false
!=	Not Equal To	5 != 5 returns true
>	Greater Than	5 > 5 returns false
<	Less Than	5 < 5 returns true
>=	Greater Than or Equal To	5 >= 5 returns false
<=	Less Than or Equal To	5 <= 5 returns true

v. Bitwise Operator:

In Java, bitwise operators perform operations on integer data at the individual bit-level. Here, the integer data includes `byte`, `int`, `short`, and `long` types of data.

There are 7 operators to perform bit-level operations in Java.

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

vi. Logical Operator:

The **Java Logical Operators** work on the Boolean operand. It's also called Boolean logical operators. It operates on two Boolean values, which return Boolean values as a result.

Operator	Description	Working
&&	Logical AND	both operands are true then only "logical AND operator" evaluate true.
	Logical OR	The logical OR operator is only evaluated as true when

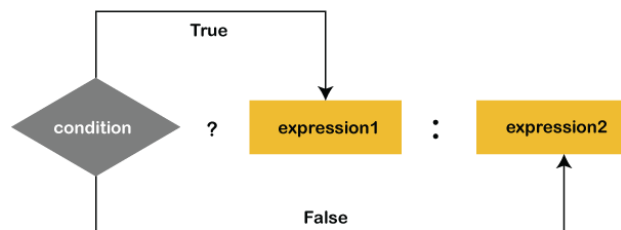
		one of its operands evaluates true. If either or both expressions evaluate to true, then the result is true.
!	Logical Not	Logical NOT is a Unary Operator, it operates on single operands. It reverses the value of operands, if the value is true, then it gives false, and if it is false, then it gives true.

vii. Ternary Operator:

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

Syntax:

variable = (condition) ? expression1 : expression2



viii. Assignment Operations:

Assignment operators are used to assign values to variables.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assigns the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assigns the	C -= A is equivalent to C = C - A

	result to the left operand.	
*=	multiply AND assignment operator. It multiplies right operand with the left operand and assigns the result to the left operand.	*= A is equivalent to C = C * A
/=	divide AND assignment operator. It divides left operand with the right operand and assigns the result to the left operand.	/= A is equivalent to C = C / A
%=	modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	%= A is equivalent to C = C % A
<<=	left shift AND assignment operator.	<<= 2 is same as C = C << 2
>>=	right shift AND assignment operator.	>>= 2 is same as C = C >> 2
&=	bitwise AND assignment operator.	&= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	^= 2 is same as C = C ^ 2
 =	bitwise inclusive OR and assignment operator.	= 2 is same as C = C 2

❖ Priority of Operators (or) Operator Precedence:

- The **operator precedence** represents how two expressions are bind together. In an expression, it determines the grouping of operators with operands and decides how an expression will evaluate. Larger the number higher the precedence.
- While solving an expression two things must be kept in mind the first is a **precedence** and the second is **associativity**.

➤ Precedence:

- ✓ Precedence is the priority for grouping different types of operators with their operands. It is meaningful only if an expression has more than one operator with higher or lower precedence.

- ✓ The operators having higher precedence are evaluated first. If we want to evaluate lower precedence operators first, we must group operands by using parentheses and then evaluate.

➤ **Associativity:**

- ✓ We must follow associativity if an expression has more than two operators of the same precedence.
- ✓ In such a case, an expression can be solved either **left-to-right** or **right-to-left**, accordingly.

The following table describes the precedence and associativity of operators used in Java.

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses	Left to Right
		Array subscript	
		Member selection	
14	++ --	Unary post- increment	Right to Left
		Unary post-decrement	
13	++ -- + - ! ~ (type)	Unary pre-increment	Right to left
		Unary pre-decrement	
		Unary plus	
		Unary minus	
		Unary logical negation	
		Unary bitwise complement Unary type cast	
12	* / %	Multiplication	Left to Right
		Division	
		Modulus	
11	+ -	Addition	Left to right
		Subtraction	
10	<< >> >>>	Bitwise left shift	Left to right
		Bitwise right shift with sign extension	
		Bitwise right shift with zero extension	
9	< <= > >= instanceof	Relational less than	Left to right
		Relational less than or equal	
		Relational greater than	
		Relational greater than or	

		equal Type comparison (objects only)	
8	==	Relational equal to	Left to right
	!=	Relational not equal to	
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5		Bitwise inclusive OR	Left to right
4	&&	logicalAND	Left to right
3		Logical OR	Left to right
2	?:	Ternary conditional	Right to Left
1	= += -= *= /= % =	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

❖ Control statements in java:

- Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear.
- However, Java provides statements that can be used to control the flow of Java code.
- Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements

- if statements
- switch statement

2. Loop statements

- do while loop
- while loop
- for loop

3. Jump statements

- break statement
- continue statement

▪ Decision Making statements:

- As the name suggests, decision-making statements decide which statement to execute and when.
- Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided.
- There are two types of decision-making statements in Java, i.e., *If statement* and *switch statement*.

a) If Statement:

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

i. Simple if statement:

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

Syntax:

```
if(condition)

{

statement 1; //executes when condition is true

}
```

```
class Biggrst {
public static void main(String[] args) {
int x = 10;
int y = 5;
if(x<y) {
System.out.println(x+ " is greater value");
}
}
}
```

Output:

```
10 is greater value
```

ii. if-else statement:

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

Syntax:

```

if(condition)

{

statement 1; //executes when condition is true

}

else

{

statement 2; //executes when condition is false

}

```

```

class EvenOrOdd {
public static void main(String[] args) {
int x = 10;
if(x%2==0) {
System.out.println(x + "is even");
} else
System.out.println(x + " is odd");
}
}

```

Output:

```
10 is even
```

iii. if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree

where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax:

if(condition 1)

{

statement 1; //executes when condition 1 is true

}

else if(condition 2)

{

statement 2; //executes when condition 2 is true

}

else

{

statement 2; //executes when all the conditions are false

}

```
import java.util.Scanner;
class Biggest{
public static void main(String []args){
int a,b,c;
Scanner s=new Scanner(System.in);
System.out.println("enter three numbers");
a=s.nextInt();
b=s.nextInt();
c=s.nextInt();
if((a>b)&&(a>c)){
System.out.print(a+"is biggest");
```

```

}
else if(b>c){
System.out.print(b+"is biggest");
}
else
System.out.print(c+"is biggest");
}
}

```

Output:

```

enter three numbers
30
40
20
40 is biggest

```

iv. Nested if-statement:

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

Syntax:

```

if(condition 1)

{

statement 1; //executes when condition 1 is true

if(condition 2)

{

statement 2; //executes when condition 2 is true

}

else

{

```

```
statement 2; //executes when condition 2 is false
```

```
}
```

```
}
```

b) Switch statement:

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

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java.
- Cases cannot be duplicate.
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.

Syntax:

```
switch (expression)
```

```
{
```

```
  case value1:
```

```
    statement1;
```

```
  break;
```

```
  .
```

```
  .
```

```
  .
```

```
  case valueN:
```

statementN;

break;

default:

default statement;

}

```
class Test {  
  
    public static void main(String args[]) {  
  
        char grade = 'C';  
  
        switch(grade) {  
            case 'A' :  
                System.out.println("Excellent!");  
                break;  
            case 'B' :  
            case 'C' :  
                System.out.println("Well done");  
                break;  
            case 'D' :  
                System.out.println("You passed");  
            case 'F' :  
                System.out.println("Better try again");  
                break;  
            default :  
                System.out.println("Invalid grade");  
        }  
        System.out.println("Your grade is " + grade);  
    }  
}
```

Output

Well done

Your grade is C

▪ **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.

a) **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.

Syntax:

for(initialization, condition, increment/decrement)

{

//block of statements

}

```
public class ForExample {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            System.out.print(i+"\t");
        }
    }
}
```

Output:

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

b) 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.

Syntax:

while(condition)

{

//looping statements

}

```
public class WhileExample {
    public static void main(String[] args) {
        int i=10;
        while(i>0){
            System.out.print(i+"\t");
            i--;
        }
    }
}
```

Output:

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

c) 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.

Syntax:**do**

{

//statements

}

while (condition);

```

public class DoWhileExample {
    public static void main(String[] args) {
        int i=1;
        {
            System.out.print(i+"\t");
            i++;
        } while(i<10);
    }
}

```

Output:

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

▪ 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.

a) Break statement:

- 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.

Syntax:

break;

```
public class ForExample {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            if(i==5)
            {
                break;
            }
            System.out.print(i+"\t");
        }
    }
}
```

Output:

```
1      2      3      4      5
```

b) Continue statement:

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

Syntax:

continue;

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



```

for(int i=1;i<=10;i++){
    if((i>4)&& (i<7))
    {
        continue;
    }
    System.out.print(i+"\t");
}
}

```

Output:

```

1      2      3      4      7      8      9      10

```

- **Return statement:**

In Java programming, the return statement is used for returning a value when the execution of the block is completed.

Returning a value from a method:

- In Java, every method is declared with a return type such as int, float, double, string, etc.
- These return types required a return statement at the end of the method. A return keyword is used for returning the resulted value.
- The void return type doesn't require any return statement. If we try to return a value from a void method, the compiler shows an error.

Following are the important points must remember while returning a value:

- The return type of the method and type of data returned at the end of the method should be of the same type. For example, if a method is declared with the float return type, the value returned should be of float type only.

Syntax:

```
return returnvalue;
```

- ❖ **Accepting input from the keyboard:**

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

1. InputStreamReader:

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

- connects to input stream of keyboard.
- converts the byte-oriented stream into character-oriented stream

BufferedReader class:

BufferedReader class can be used to read data line by line by readLine() method.

Example:

```
InputStreamReader r=new InputStreamReader(System.in);
```

```
BufferedReader br=new BufferedReader(r);
```

In above example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

2. Console:

- The Java Console class is used to get input from console.
- It provides methods to read texts and passwords.
- If you read password using Console class, it will not be displayed to the user.
- The java.io.Console class is attached with system console internally.

Example:

```
String text=System.console().readLine();
```

```
System.out.println("Text is: "+text);
```

3. Scanner:

- Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.
- It provides many methods to read and parse various primitive values.
- The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression.
- By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

- The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc.
- To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Syntax:

Scanner in = **new** Scanner(System.in);

4. DataInputStream:

- A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
- An application uses a data output stream to write data that can later be read by a data input stream.
- DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.

❖ Java Arrays (One Dimensional array)

An array is a collection of similar types of data. The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

1. Declare an array in Java

In Java, here is how we can declare an array.

dataType[] arrayName;

- **dataType** - it can be [primitive data types](#) like `int`, `char`, `double`, `byte`, etc.
- **arrayName** - it is an [identifier](#)

For example,

double[] data;

Here, **data** is an array that can hold values of type `double`.

2. Memory allocation

- we have to allocate memory for the array in Java. *Memory representation*. When you use new to create an array, Java reserves space in memory for it (and initializes the values). This process is called *memory allocation*.

For example,

```
double[] data;
data = new double[10];
```

Here, the array can store **10** elements. We can also say that the **size or length** of the array is 10.

In Java, we can declare and allocate the memory of an array in one single statement. For example,

```
double[] data = new double[10];
```

3. Initialize Arrays in Java

In Java, we can initialize arrays during declaration. For example,

```
int[] age = {12, 4, 5, 2, 5};
```

Here, we have created an array named age and initialized it with the values inside the curly brackets.

Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array

In the Java array, each memory location is associated with a number. The number is known as an array index.

- Array indices always start from 0. That is, the first element of an array is at index 0.
- If the size of an array is n , then the last element of the array will be at index $n-1$.

4. Access Elements of an Array

We can use loops to access all the elements of the array at once.

Looping Through Array Elements

In Java, we can also loop through each element of the array. For example,

Example: Using For Loop

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

        int[] age = {12, 4, 5};
        System.out.println("Using for Loop:");

        for(int i = 0; i < age.length; i++) {
            System.out.println(age[i]);
        }
    }
}
```

Output

Using for Loop:

```
12
4
5
```

In the above example, we are using the [for Loop in Java](#) to iterate through each element of the array. Notice the expression inside the loop,

```
age.length
```

Here, we are using the `length` property of the array to get the size of the array.

❖ Types of Arrays

In Java, there are two types of arrays:

1. Single-Dimensional Array
2. Multi-Dimensional Array

Single-Dimensional Array:

- An array that has **only one subscript** or one dimension is known as a single-dimensional array. It is just a list of the same data type variables.
- One dimensional array can be of either one row and multiple columns or multiple rows and one column.
- The declaration and initialization of an single-dimensional array is same as array's initialization and declaration.

Example:

```
int marks[] = {56, 98, 77, 89, 99};
```

Multi-Dimensional Array:

- A multi-dimensional array is just an array of arrays that represents multiple rows and columns.
- In multi-dimensional arrays, we have two categories:
 - i. Two-Dimensional Arrays
 - ii. Three-Dimensional Arrays

Two-Dimensional Arrays:

- An array involving two subscripts [] [] is known as a two-dimensional array. They are also known as the array of the array. Two-dimensional arrays are divided into rows and columns and are able to handle the data of the table.

Syntax:

```
DataTypeArrayName[row_size][column_size];
```

Example:

```
int arr[5][5];
```

Three-Dimensional Arrays:

- When we require to create two or more tables of the elements to declare the array elements, then in such a situation we use three-dimensional arrays.
- 3D array adds an extra dimension to the 2D array to increase the amount of space.
- Eventually, it is set of the 2D array. Each variable is identified with three indexes; the last two dimensions represent the number of rows and columns, and the first dimension is to select the block size.
- The first dimension depicts the number of tables or arrays that the 3D array contains.

Syntax:DataTypeArrayName[size1][size2][size3];

Example:

```
int a[5][5][5];
```

❖ Command line arguments:

- Java command-line argument is an argument i.e. passed at the time of running the Java program.
- In the command line, the arguments passed from the console can be received in the java program and they can be used as input.
- The users can pass the arguments during the execution bypassing the command-line arguments inside the main() method. We need to pass the arguments as space-separated values.
- We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments.
- These arguments convert into a string array and are provided to the main() function as a string array argument.

- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[].
- It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length.
- Internally, JVM wraps up these command-line arguments into the args[] array that we pass into the main() function.
- We can check these arguments using args.length method. JVM stores the first command-line argument at args[0], the second at args[1], the third at args[2], and so on.

UNIT-II

Strings:

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The `java.lang.String` class is used to create a string object. An array of characters works same as Java string.

For example:

```
char[] ch={'c','o','m','p','u','t','e','r'};
```

```
String s=new String(ch);
```

is same as:

```
String s="computer";
```

❖ Creating Strings:

There are two ways to create String object:

1. By string literal
2. By new keyword

1) By string literal:

- Java String literal is created by using double quotes.
- String objects are stored in a special memory area known as the "string constant pool".
- Each time you create a string literal, the JVM checks the "string constant pool" first.
- If the string already exists in the pool, a reference to the pooled instance is returned.
- If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
- The string literal concept is used to make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Syntax:

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

Example:

```
String s="welcome";
```

2) By new keyword:

`String s=new String("Welcome");` // creates two objects and one reference variable

- In such 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 a heap (non-pool).

Program:

```
public class StringExample
{
    public static void main(String args[])
    {
        String s1="java";
        char ch[]={'s','t','r','i','n','g','s'};
        String s2=new String(ch);
        String s3=new String("example");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
    }
}
```

Output:

```
java
strings
example
```

❖ String class methods:

- The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**.
- By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.
- Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

Some of the important methods are:

1) Java String toUpperCase() and toLowerCase() method:

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

Program:

```
public class Testmethodofstringclass
{
    public static void main(String args[])
    {
        String s="Sachin";
        System.out.println(s.toUpperCase()); //SACHIN
        System.out.println(s.toLowerCase()); //sachin
    }
}
```

```
System.out.println(s);//Sachin(no change in original)
}
}
```

2) Java String trim() method:

The String class trim() method eliminates white spaces before and after the String.

Program:

```
public class Testmethodofstringclass1
{
    public static void main(String args[])
    {
        String s=" Sachin ";
        System.out.println(s);// Sachin
        System.out.println(s.trim());//Sachin
    }
}
```

Output:

```
Sachin
Sachin
```

3) Java String startsWith() and endsWith() method:

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Program:

```
public class Testmethodofstringclass2
{
    public static void main(String args[])
    {
        String s="Sachin";
        System.out.println(s.startsWith("Sa"));//true
        System.out.println(s.endsWith("n"));//true
    }
}
```

Output:

```
true
true
```

4) Java String charAt() Method:

The String class charAt() method returns a character at specified index.

Program:

```
public class Testmethodofstringclass3
{
```

```

public static void main(String args[])
{
    String s="Sachin";
    System.out.println(s.charAt(0)); // S
    System.out.println(s.charAt(3)); // h
}
}

```

Output:

S
h

5) Java String length() Method:

The String class length() method returns length of the specified String.

Program:

```

public class Testmethodofstringclass4
{
    public static void main(String args[])
    {
        String s="Sachin";
        System.out.println(s.length()); // 6
    }
}

```

Output:

6

6) Java String valueOf() Method:

The String class valueOf() method converts given type such as int, long, float, double, boolean, char and char array into String.

Program:

```

public class Stringoperation7
{
    public static void main(String args[])
    {
        int a=10;
        String s=String.valueOf(a);
        System.out.println(s+10);
    }
}

```

Output:

1010

7) Java String replace() Method:

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

Program:

```
public class Stringoperation8
{
    public static void main(String ar[])
    {
        String s1="Java is a programming language. Java is a platform. Java is an Island.";
        String replaceString=s1.replace("Java","Oops");
        //replaces all occurrences of "Java" to "Oops"
        System.out.println(replaceString);
    }
}
```

Output:

Oops is a programming language.Oops is a platform. Oops is an Island.

❖ String comparison:

String is a sequence of characters.InJava,objects of String are immutable which means they are constant and cannot be changed once created.

There are 5 ways to compare two Strings in java:

1) Using user-defined function:

Define a function to compare values with following conditions:

- if(string1>string2)it returns a positive value.
- if both the strings are equal
i.e.(string1==string2)it returns zero.
- if(string1<string2)it returns a negative value.

The value is calculated as (int)str1.charAt(i)-(int)str2.charAt(i)

2) Using String.equals() :

In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are same then it returns true. If any character does not match, then it returns false.

Syntax:

```
str1.equals(str2);
```

Program:

```
class Teststringcomparison1
```

```

{
    public static void main(String args[])
    {
        String s1="Sachin";
        String s2="Sachin";
        String s3=new String("Sachin");
        String s4="Saurav";
        System.out.println(s1.equals(s2)); // true
        System.out.println(s1.equals(s3)); // true
        System.out.println(s1.equals(s4)); // false
    }
}

```

Output:

```

true
    true
    false

```

3) Using String.equalsIgnoreCase() :

The String.equalsIgnoreCase() method compares two strings irrespective of the case (lower or upper) of the string. This method returns true if the argument is not null and the contents of both the Strings are same ignoring case, else false.

Syntax:

```
str2.equalsIgnoreCase(str1);
```

Program:

```

class Teststringcomparison2
{
    public static void main(String args[])
    {
        String s1="Sachin";
        String s2="SACHIN";
        System.out.println(s1.equals(s2)); // false
        System.out.println(s1.equalsIgnoreCase(s2)); // true
    }
}

```

```

}
}

```

Output:

```

false
true

```

4) Using Objects.equals() :

Object.equals(Object a, Object b) method returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.

Syntax:

```
public static boolean equals(Object a, Object b)
```

5) Using String.compareTo() :**Syntax:**

```
int str1.compareTo(String str2)
```

Working:

It compares and returns the following values as follows:

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value

Why not to use == for comparison of Strings?

In general both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

- Main difference between .equals() method and == operator is that one is method and other is operator.
- One can use == operators for reference comparison (address comparison) and .equals() method for content comparison.
- In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects

Program:

```
class Teststringcomparison4
```

```

{
    public static void main(String args[])
    {
        String s1="Sachin";
        String s2="Sachin";
        String s3="Ratan";
        System.out.println(s1.compareTo(s2)); // 0
        System.out.println(s1.compareTo(s3)); // 1(because s1>s3)
        System.out.println(s3.compareTo(s1)); // -1(because s3 < s1 )
    }
}

```

```

class StringFunction{

    public static void main(String []args)

    {

        String s1="ADITYA ";

        String s2="Degree College for Women";

        boolean x=s1.equals(s2);

        System.out.println("String1:"+s1);

        System.out.println("String2:"+s2);

        System.out.println("Compare s1 and s2:"+x);

        System.out.println("Character at 3rd position in "+s1+" is:"+s1.charAt(2));

        System.out.println("Cocadination of two strings is:"+s1.concat(s2));

        System.out.println("Lenth of "+s1+" is:"+s1.length());

        System.out.println("convert "+s1+" in Lower case:"+s1.toLowerCase());

        System.out.println("convert "+s2+" in Upper case:"+s2.toUpperCase());
    }
}

```

```

System.out.println("g index position in "+s2+" is:"+s2.indexOf('g'));

System.out.println("substring upto 4 in "+s1+"is:"+s1.substring(0,4));

System.out.println("substring of after 7th position in "+s2+" is:"+s2.substring(7));

}

}

```

Output:

```

String1:ADITYA

String2:Degree College for Women

Compare s1 and s2:false

Character at 3rd position in ADITYA is:I

Cocadination of two strings is:ADITYA Degree College for Women

Lenth ofADITYA is:7

convert ADITYA in Lower case:aditya

convert Degree College for Women in Upper case:DEGREE COLLEGE FOR WOMEN

g index position in Degree College for Women is:2

substring upto 4 in ADITYA is:ADIT

substring of after 7th position in Degree College for Women is :College for Women

```

❖ Immutability of Strings:

- A String is defined as a sequence or an array of characters. Strings are treated as objects in the Java programming language.
- The term "immutable string" in Java refers to a string object that cannot be altered, but the reference to the object can be changed.

- Every time we make a modification, a new instance of that string is created and the previous value is copied to the new String with the change.
- The String class is marked final to prevent overriding the functionality of its methods.
- In Java, the String class and all wrapper classes which include Boolean, Character, Byte, Short, Integer, Long, Float, and Double are immutable.
- A user is free to create immutable classes of their own

❖ Introduction to OOPS / Features of OOPS:

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object:

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc.
- It can be physical or logical. An Object can be defined as an instance of a class.
- An object contains an address and takes up some space in memory.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class:

- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance:

- *When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.*
- *When we write a class, we inherit properties from other classes.*
- *It provides code reusability. It is used to achieve runtime polymorphism.*

Polymorphism:

- *If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.*
- *In Java, we use method overloading and method overriding to achieve polymorphism.*
- *Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.*

Abstraction:

- *Hiding internal details and showing functionality is known as abstraction.*
- *For example phone call, we don't know the internal processing.*
- *In Java, we use abstract class and interface to achieve abstraction.*

Encapsulation:

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation.*
- *For example, a capsule, it is wrapped with different medicines.*
- *A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.*

Association:

- *Association represents the relationship between the objects. Here, one object can be associated with one object or many objects.*
- *There can be four types of association between the objects:*
 - *One to One*
 - *One to Many*
 - *Many to One, and*
 - *Many to Many*

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

Aggregation:

- *Aggregation is a way to achieve Association.*
- *Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects.*

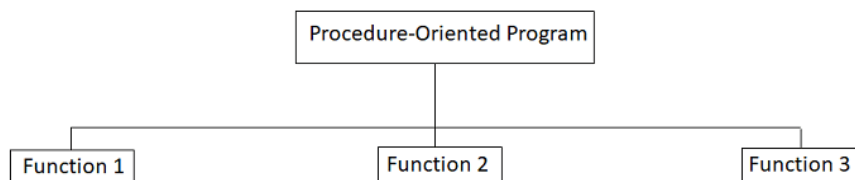
- It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition:

- The composition is also a way to achieve Association.
- The composition represents the relationship where one object contains other objects as a part of its state.
- There is a strong relationship between the containing object and the dependent object.
- It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

❖ Procedure Oriented Approach:

- Procedure-Oriented Programming is the traditional way of programming, where an application problem is viewed as a sequence of steps (algorithms).
- As per the algorithm, the problem is broken down into many modules (functions) such as data entry, reporting, querying modules, etc. as shown in the figure.



Generalized Structure of a Procedure-Oriented Program:

- There are two types of data, which are associated with these modules- one is global and another is local data.
- Global data items are defined in the main program, whereas local data is define within associated functions.
- High-level languages like COBOL, Pascal, BASIC, Fortran, C, etc. are based on a procedure-oriented approach and hence are also called procedural languages.

Problems in Procedure Oriented Approach:

- In large program, it is difficult to identify which data is used for which function.
- Maintaining and enhancing(improve the quality) program code is still difficult because of global data. Focus on functions rather than data.
- It does not model real world problem very well.Since functions are action oriented and do not really correspond to the elements of problem.
- As most of the functions share global data, this data is freely available to all functions. Although this freely available data is easily accessed by any function, it can create certain problems. When a new function is written for analyzing this

data in a different way it is possible that data may be changed accidentally.

Therefore, there is no security of data in procedure-oriented programming.

- Another problem is that whenever data structure is required to be changed, all the functions using that data must also be changed. So, procedure-oriented programming (especially for complex applications) are extremely difficult to modify and maintain.

❖ Differences between OOP and POP:

OOP

OOP, refers to Object Oriented Programming and its deals with objects and their properties. Major concepts of OOPs are –

- Class/objects
- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

POP

POP, refers to Procedural Oriented Programming and its deals with programs and functions. Programs are divided into functions and data is global.

Following are the important differences between OOP and POP.

Sr. No.	Key	OOP	POP
1	Definition	OOP stands for Object Oriented Programing.	POP stands for Procedural Oriented Programming.
2	Approach	OOP follows bottom up approach.	POP follows top down approach.
3	Division	A program is divided to objects and their interactions.	A program is divided into funtions and they interacts.
4	Inheritance supported	Inheritance is supported.	Inheritance is not supported.
5	Access control	Access control is supported via access modifiers.	No access modifiers are supported.
6	Data Hiding	Encapsulation is used to hide data.	No data hiding present. Data is globally accessible.
7	Example	C++, Java	C, Pascal

- **Objects and Classes in Java**

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
3. 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;
}
```

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”.

Note: When we create an object which is a non primitive data type, it's always allocated on the heap memory.

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.

Example:

```
// 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

• Difference between Java Class and Objects

The differences between class and object in Java are as follows:

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

Let's see the example:

```
class Rectangle{
    int length;
    int width;
    void insert(int l,int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
        r2.calculateArea();
    }
}
```

Output:

```
55
45
```

❖ Constructors

In [Java](#), a constructor is a block of codes similar to the method. It is called when an instance of the [class](#) is created. At the time of calling constructor, memory for the object is allocated in the memory.

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

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

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

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

Rules for creating Java constructor

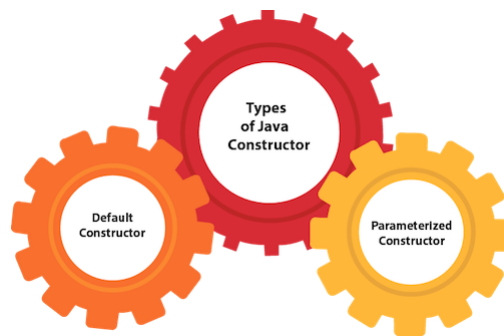
There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

1. `<class_name>(){}`

Example of default constructor

```
class Data {
    //default constructor
    public Data() {
        System.out.println("Hello students");
    }
    public static void main(String[] args) {
        Data d = new Data();
    }
}
```

Output:

```
Hello students
```

Java Parameterized Constructor

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

Use the parameterized constructor

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

Example of parameterized constructor

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

```
class Student{
    int id;
    String name;

    Student(int i,String n){
        id = i;
        name = n;
    }

    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){

        Student4 s1 = new Student4(111,"Karan");
        Student4 s2 = new Student4(222,"Aryan");

        s1.display();
        s2.display();
    }
}
```

Test it Now

Output:

```
111 Karan
222 Aryan
```

❖ 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

```
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
```

❖ Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

❖ Types of Variable:

1. Local Variables :

A variable defined within a block or method or constructor is called a local variable.

- These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
- Initialization of the local variable is mandatory before using it in the defined scope.

2. Instance Variables:

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for Wrapper classes like *Integer* it is *null*, etc.
- Instance variables can be accessed only by creating objects.
- We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

3. Static Variables:

Static variables are also known as class variables.

- These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
- Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for *Wrapper classes* like *Integer* it is *null*, etc.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
- Static variables cannot be declared locally inside an instance method.
- Static blocks can be used to initialize static variables.

❖ Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
public static void main(String args[]){
A obj=new A();
System.out.println(obj.data);//Compile Time Error
obj.msg();//Compile Time Error
}
}
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
```



```

A obj = new A();//Compile Time Error
obj.msg();//Compile Time Error
}
}

```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```

//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;

class B extends A{
public static void main(String args[]){
B obj = new B();
obj.msg();
}
}

```

Output:Hello

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}

//save by B.java

package mypack;
import pack.*;

class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
}
}
```

Output:Hello

❖ Methods in Java:

- The **method in Java** is a collection of statements that perform some specific task and return the result to the caller.
- A Java method can perform some specific task without returning anything.
- Java Methods allow us to **reuse** the code without retyping the code.
- In Java, every method must be part of some class that is different from languages like C, C++, and Python.
- Methods are time savers and help us to reuse the code without retyping the code.
 1. A method is like a function i.e. used to expose the behavior of an object.
 2. It is a set of codes that perform a particular task.

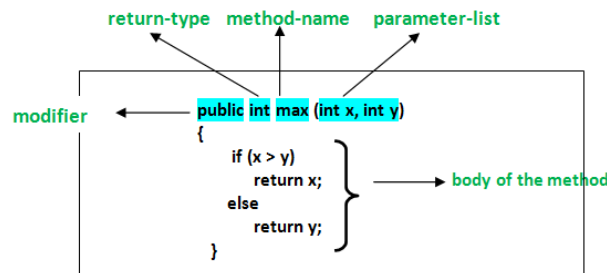
Syntax of Method:

```
<access_modifier><return_type><method_name>( list_of_parameters)

{

    //body

}
```

Example:**Types of Methods in Java:**

There are two types of methods in Java:

1. 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.

2. 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.

Ways to Create Method in Java:

There are two ways to create a method in Java:

1. Instance Method: Access the instance data using the object name. Declared inside a class.

Syntax:

```
// Instance Method
void method_name(){
    body // instance area
}
```

2. Static Method: Access the static data using class name. Declared inside class with **static** keyword.

Syntax:

```
// Static Method
static void method_name(){
```

```

body // static area
}

```

❖ Passing Arguments to Methods in Java:

There are mainly two ways of passing arguments to methods:

- Pass by value
- Pass by reference

Java directly supports passing by value; however, passing by reference will be accessible through reference objects.

1. Pass by value:

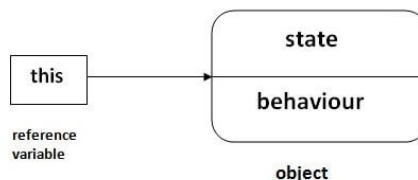
- When the arguments are passed using the pass by value mechanism, only a copy of the variables are passed which has the scope within the method which receives the copy of these variables.
- The changes made to the parameters inside the methods are not returned to the calling method.
- This ensures that the value of the variables in the calling method will remain unchanged after return from the calling method.

2. Pass by reference:

- In the pass by reference mechanism, when parameters are passed to the methods, the calling method returns the changed value of the variables to the called method.
- The call by reference mechanism is not used by Java for passing parameters to the methods.
- Rather it manipulates objects by reference and hence all object variables are referenced.

❖ This keyword:

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.

4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

❖ Recursion:

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
- Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Any method that implements Recursion has two basic parts:

1. Method call which can call itself i.e. recursive
2. A precondition or base condition that will stop the recursion.

Note that a precondition is necessary for any recursive method as, if we do not break the recursion then it will keep on running infinitely and result in a stack overflow.

Syntax:

```
methodName (T parameters...)
{
    if(precondition == true)

//precondition or base condition
{
    return result;
}
return methodName (T parameters...);

//recursive call
}
```

base condition: In a recursive function, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems. The role of the base condition is to stop a recursive function from executing endlessly – once a pre-specified base condition is met, the function knows it's time to exit.

Program: Infinite times

```
public class RecursionExample1
{
    static void p()
    {
```

```

    System.out.println("hello");
    p();
}
public static void main(String[] args)
{
    p();
}
}

```

Output:

hello

hello

...

java.lang.StackOverflowError

Program:finite times

```

public class RecursionExample2
{
    static int count=0;
    static void p(){
        count++;
        if(count<=5){
            System.out.println("hello "+count);
            p();
        }
    }
    public static void main(String[] args) {
        p();
    }
}

```

Output:

hello1

hello2

hello3

hello4

hello5

❖ Factory method/design pattern:

- It is a creational design pattern that talks about the creation of an object.
- The factory design pattern says that define an interface (A java interface or an abstract class) for creating object and let the subclasses decide which class to instantiate.

- The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses.
- Since these design patterns talk about the instantiation of an object and so it comes under the category of creational design pattern.
- If we notice the name Factory method, that means there is a method which is a factory, and in general, factories are involved with creational stuff and here with this, an object is being created.
- It is one of the best ways to create an object where object creation logic is hidden from the client.

Implementation:

1. Define a factory method inside an interface.
2. Let the subclass implements the above factory method and decides which object to create.

In Java, constructors are not polymorphic, but by allowing subclass to create an object, we are adding polymorphic behavior to the instantiation. In short, we are trying to achieve Pseudo polymorphism by letting the subclass to decide what to create, and so this Factory method is also called a virtual constructor.

❖ Inheritance:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of **OOPs** (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new **classes** that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

we use inheritance for

- For **Method Overriding** (so **runtime polymorphism** can be achieved).
- For Code Reusability.

Syntax:

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

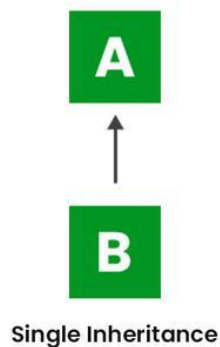
- **Subclass** is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Superclass** is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Types of Java Inheritance

1. Single-level inheritance
2. Multi-level 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.



```

class Animals {
    void print() {
        System.out.println("Living animals");
    }
}

class WildAnimals extends Animals{
    void display() {
        System.out.println("Cheetha is a wild animal");
    }
}

class AnimalExample {
    public static void main(String[] args) {
        WildAnimals WA = new WildAnimals();
        WA.print();
        WA.display();
    }
  
```



```
}
```

Single inheritance

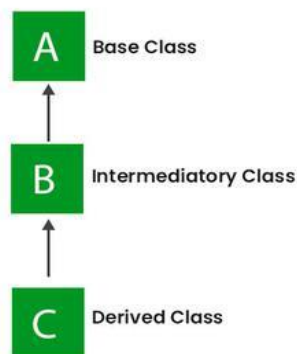
Output

Living animals

Cheetha is a wild animal

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

Multilevel Inheritance

```

class Animals {
    void print() {
        System.out.println("Living animals");
    }
}

class WildAnimals extends Animals{
    void display() {
        System.out.println("Cheetha is a wild animal");
    }
}

class Leopards extends WildAnimals{
    public void display() {
        System.out.println("Leopards are actually the smallest of the cats but is stronger and bulkier than the cheetah");
    }
}

class AnimalExample {
    public static void main(String[] args) {
        Leopards L = new Leopards();
        L.print();
        L.display();
    }
}

```

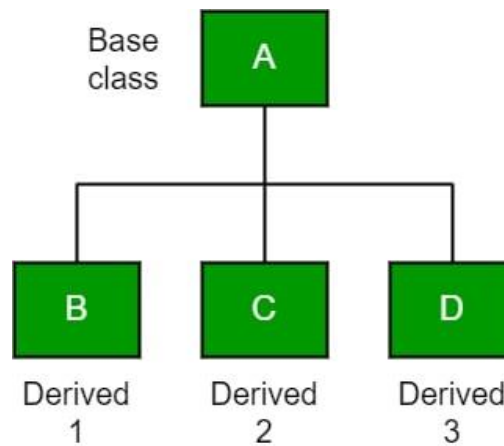
Output

Cheetha is a wild animal

Leopards are actually the smallest of the cats but is stronger and bulkier than the cheetah

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.



```

class Animals {
    void print() {
        System.out.println("Living animals");
    }
}

class WildAnimals extends Animals {
    void display() {
        System.out.println("Cheetha is a wild animal");
    }
}

class DomesticAnimals extends Animals {
    void display() {
        System.out.println("Cat is a domestic animal");
    }
}

class AnimalExample {
    public static void main(String[] args) {
        DomesticAnimals DA= new DomesticAnimals();

        DA.print();
        DA.display();
    }
}
  
```

Output

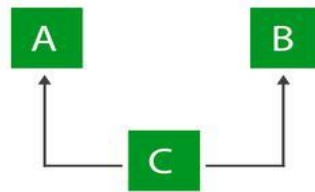
Living animals

Cat is a domestic animal

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

Multiple Inheritance

```

interface C
{
void procedure();
}
interface Java
{
void object();
}
class Programming implements C,Java
{
public void procedure()
{
System.out.println("c is a procedure oriented programming language");
}
public void object()
{
System.out.println("java is an object oriented programming language");
}
}
class ProgrammingExample
{
public static void main(String args[])
{
Programming p= new Programming();
p.procedure();
p.object();
}
}
  
```

Output

c is a procedure oriented programming language

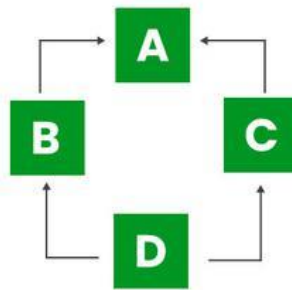
java is an object oriented programming language

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

Hybrid Inheritance

UNIT-III

❖ Polymorphism:

- **Polymorphism in Java** is a concept by which we can perform a *single action indifferent ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.
- Runtime polymorphism is achieved through method overriding, and compile-time polymorphism is achieved through method overloading.

a. Method overriding:

- Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in parent class is called overridden method and the method in child class is called overriding method.
- we cannot override static method because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Rules for Java Method Overriding:

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example:

We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

Program:

In the below, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle
{
    //defining a method
```

```

    void run(){System.out.println("Vehicle is running");
    }
    }
    //Creating a child class
    class Bike2 extends Vehicle
    {
        //defining the same method as in the parent class
        void run(){System.out.println("Bike is running safely");
        }

        public static void main(String args[])
        {
            Bike2 obj = new Bike2();//creating object
            obj.run();//calling method
        }
    }

```

Output:

Bike is running safely

❖ Method Overloading in Java

In Java, Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters

Different Ways of Method Overloading in Java

- Changing the Number of Parameters.
- Changing Data Types of the Arguments.

Changing the Number of Parameters.

```

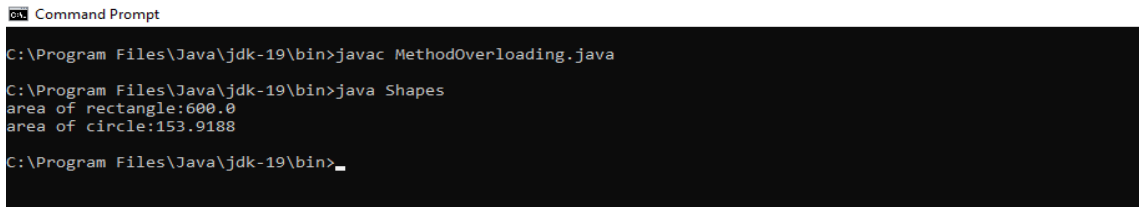
class MethodOverloading
{
    double getArea(double x)
    {
        double    area=3.1412*(x*x);
        return area;
    }
    double getArea(double x,double y) {
        double area=x*y;
        return area;
    }
    double getArea(double x,double y,double z) {
        double area=x*y*z;
    }
}

```

```

        return area;
    }
}
class Shapes{
    public static void main(String args[]) {
        MethodOverloading mo=new MethodOverloading();
        System.out.println("area of rectangle:"+mo.getArea(20,30));
        System.out.println("area of circle:"+mo.getArea(7));
    }
}

```



```

C:\Program Files\Java\jdk-19\bin>javac MethodOverloading.java
C:\Program Files\Java\jdk-19\bin>java Shapes
area of rectangle:600.0
area of circle:153.9188
C:\Program Files\Java\jdk-19\bin>_

```

In this example the `getArea()` method is overloaded three times. The first method takes one argument, the second method takes two arguments and third method takes three arguments.

When an overloaded method is called Java looks for a match between the arguments to call the method and its parameters. This match need not always be exact, sometime when an exact match is not found, Java's automatic type conversion plays a vital role.

Changing Data Types of the Arguments.

Java supports automatic type promotion, like `int` to `long` or `float` to `double` etc. In this example we are doing the same and calling a function that takes **one integer, second float and third long type** argument. At the time of calling we passed integer values and Java treated the second argument as float type. See the below example.

```

class MethodOverloading2
{
    int rectArea(int l,int b) {
        int area=l*b;
        return area;
    }
}

```



```

        float rectArea(float l,float b) {
            float area=l*b;
            return area;
        }

        double rectArea(double l,double b) {
            double area=l*b;
            return area;
        }
    }

    class Shapes{
        public static void main(String args[]) {
            MethodOverloading2 mo2=new MethodOverloading2();
            System.out.println("area of rectangle:"+mo2.rectArea(20,30));
            System.out.println("area of rectangle:"+mo2.rectArea(20.4345,30.76546));
            System.out.println("area of rectangle:"+mo2.rectArea(20.5f,30.5f));

        }
    }
}

```

```

C:\Program Files\Java\jdk-19\bin>javac MethodOverloading2.java

C:\Program Files\Java\jdk-19\bin>java Shapes
area of rectangle:600
area of rectangle:628.67679237
area of rectangle:625.25

```

❖ Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

```
//override
```

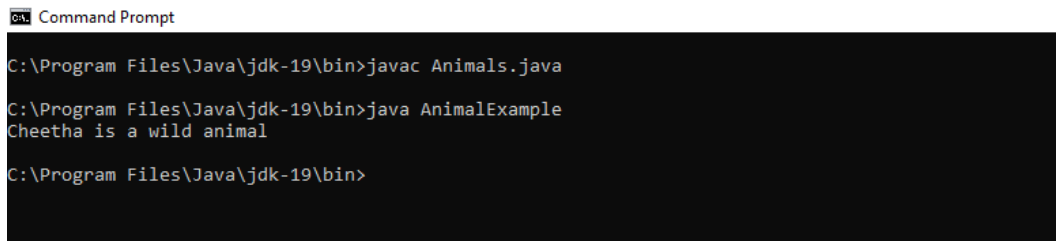
```

class Animals {
    void display() {
        System.out.println("Living animals");
    }
}

class WildAnimals extends Animals{
    void display() {
        System.out.println("Cheetha is a wild animal");
    }
}

class AnimalExample {
    public static void main(String[] args) {
        WildAnimals WA = new WildAnimals();
        WA.display();
    }
}

```



```

C:\Program Files\Java\jdk-19\bin>javac Animals.java

C:\Program Files\Java\jdk-19\bin>java AnimalExample
Cheetha is a wild animal

C:\Program Files\Java\jdk-19\bin>

```

❖ Type Casting:

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Types of Casting:

a. casting primitive types:

- Casting between primitive types enables you to convert the value of one type to another primitive type. This most commonly occurs with the numeric types.
- But one primitive type can never be used in a cast. Boolean values must be either true or false and cannot be used in a casting operation.
- In many casts between primitive types, the destination can hold larger values than the source, so the value is converted easily.

- It allows the developer to cast the value of one primitive into another. The seven primitive data type values are Boolean, Byte, Char, Short, Int, Long, Float and Double.
- There are two sub-types of primitive type casting:

1. Widening Type Casting:

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

- Both data types must be compatible with each other.
 - The target type must be larger than the source type.
- byte -> short -> char -> int -> long -> float -> double**

Program:

```
class Main {
    public static void main(String[] args) {
        // create int type variable
        int num = 10;
        System.out.println("The integer value: " + num);

        // convert into double type
        double data = num;
        System.out.println("The double value: " + data);
    }
}
```

Output:

The integer value: 10
The double value: 10.0

2. Narrowing Type Casting:

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer.

If we do not perform casting then the compiler reports a compile-time error.

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

Program:

```
class Main
{
    public static void main(String[] args)
    {
        // create double type variable
        double num = 10.99;
        System.out.println("The double value: " + num);
        // convert into int type
        int data = (int)num;
```

```

    System.out.println("The integer value: " + data);
}
}

```

Output:

The double value: 10.99

The integer value: 10

b. Reference type casting:

- If two different types of classes are associated with each other by inheritance and one of those classes is the SubClass of another, then these classes can undergo casting.
- It is important to ensure that the casting is in line with run-time rules as well as compile-time rules in Java.
- Reference type casting is further divided into two types:

1. Upcasting:

Upcasting involves the conversion of an object of SubType into an object of SuperType. Java has the provision for assigning the object without requiring the addition of an explicit cast. The compiler would know what is being done and would cast the SubType value to SuperType. This way, the object is brought to a basic level. Developers can add an explicit cast without worrying about any issues.

2. Downcasting:

Downcasting involves the conversion of a SuperType object into a Subtype object. It is the most frequently used casting wherein it is communicated to the compiler that the value of the base object isn't its own but of the SuperType object.

❖ Object class:

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.
- Notice that parent class reference variable can refer the child class object, known as upcasting.
- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Using Object Class Methods

The Object class provides multiple methods which are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public boolean equals(Object obj)	compares the given object to this object.

<code>public String toString()</code>	returns the string representation of this object.
<code>public final void notifyAll()</code>	wakes up all the threads, waiting on this object's monitor.
<code>public final void wait(long timeout,int nanos)throws InterruptedException</code>	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).
<code>protected void finalize()throws Throwable</code>	is invoked by the garbage collector before object is being garbage collected.
<code>public int hashCode()</code>	returns the hashcode number for this object.
<code>protected Object clone() throws CloneNotSupportedException</code>	creates and returns the exact copy (clone) of this object.
<code>public final void notify()</code>	wakes up single thread, waiting on this object's monitor.
<code>public final void wait(long timeout)throws InterruptedException</code>	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).
<code>public final void wait()throws InterruptedException</code>	causes the current thread to wait, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).

❖ **Abstraction:**

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction:

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

1) Abstract class:

- An abstract class in Java is one that is declared with the `abstract` keyword. It may have both abstract and non-abstract methods (methods with bodies).
- An abstract is a java modifier applicable for classes and methods in java but not for Variables.

- Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.

In Java, the following some important observations about abstract classes are as follows:

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. There can be a **final method** in abstract class but any abstract method in class (abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final"
5. We can define static methods in an abstract class
6. We can use the **abstract keyword** for declaring **top-level classes (Outer class) as well as inner classes** as abstract
7. If a **class** contains at least **one abstract method** then compulsory should declare a class as abstract
8. If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method

❖ Abstract Methods:

- Sometimes we require just method declaration in super-classes. This can be achieved by specifying the Java **abstract** type modifier.
- Abstraction can be achieved using abstract class and abstract methods. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the super-class.
- Thus, a subclass must override them to provide a method definition.

Declare Abstract Method in Java: To declare an abstract method, use this general form
abstract type method-name(parameter-list);

Important Points for Abstract Method

Important rules for abstract methods are mentioned below:

- Any class that contains one or more abstract methods must also be declared abstract.
- If a class contains an abstract method it needs to be abstract and vice versa is not true.
- If a non-abstract class extends an abstract class, then the class must implement all the abstract methods of the abstract class else the concrete class has to be declared as abstract as well.
- The following are various **illegal combinations** of other modifiers for methods with respect to abstract modifiers:

- final
- abstract native
- abstract synchronized
- abstract static
- abstract private
- abstract strictfp

❖ Interface

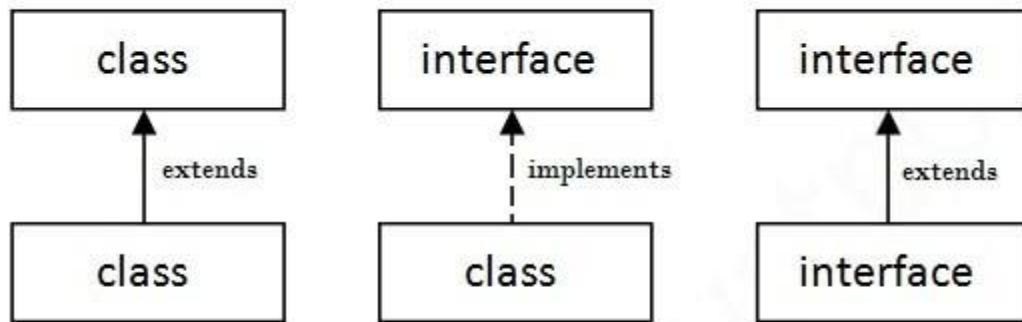
- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- 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**.
- It cannot be instantiated just like the abstract class.

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

The relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

```
interface Students
{
void print();
}
class BcaStudents implements Students
{
public void print()
{
System.out.println("Hello Bca Students");
}
}
class StudentExample
{
public static void main(String args[])
{
BcaStudents bs= new BcaStudents();
bs.print();
}
}
```

Output:

Hello Bca Students

❖ Multiple inheritance using java:

Multiple inheritances can be achieved through the use of interfaces. Interfaces are similar to classes in that they define a set of methods that can be implemented by classes. Here's how to implement multiple inheritance using interfaces in Java.

Step 1: Define the interfaces

```
interface Interface1
{
    void method1();
}
interface Interface2
{
    void method2();
}
```

Step 2: Implement the interfaces in the class

```
public class MyClass implements Interface1, Interface2
{
    public void method1()
    {
        // implementation of method1
    }
    public void method2()
    {
        // implementation of method2
    }
}
```

Step 3: Create an object of the class and call the interface methods

```
MyClass obj = new MyClass();
obj.method1();
obj.method2();
```

❖ Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.

Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

access package from another package

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

❖ Java Packages & API (system packages)

A Java package is a group of related classes and interfaces that are stored together in a directory or subdirectory. Java API (Application Programming Interface) is a set of prewritten

Java classes that are provided to developers as a means of performing common tasks, such as manipulating data or creating graphical user interfaces.

To use a Java package or API, you must first import it into your Java code using the "import" statement.

Built-in Packages

Java comes with several built-in packages that developers can use in their code. These packages are part of the Java API and provide a wide range of functionality, including data manipulation, networking, and user interface design.

Some examples of built-in packages in Java include:

java.lang: provides fundamental classes and interfaces, such as String, Integer, and Boolean.

java.util: provides utility classes and interfaces, such as ArrayList and Date.

java.io: provides classes for input and output operations, such as reading and writing files.

Import a Class

To import a class in Java, you need to use the import statement followed by the fully qualified name of the class.

For example:

```
import java.util.ArrayList;

public class MyClass {
    public static void main(String[] args) {
        ArrayList myList = new ArrayList();
        myList.add("Hello");
        myList.add("World");
        System.out.println(myList);
    }
}
```

Output:

```
Hello World
```

In the above example, we imported the **ArrayList** class from the **java.util** package using the import statement. We then created an instance of the **ArrayList** class and added two strings to it. Finally, we printed the contents of the **ArrayList** using the **System.out.println** statement.

Import a Package

To import an entire package in Java, you need to use the import statement followed by the name of the package.

For example:

```
import java.util.*;

public class MyClass {
    public static void main(String[] args) {
        ArrayList myList = new ArrayList();
        myList.add("Hello");
        myList.add("World");
        System.out.println(myList);
    }
}
```

Output:

```
Hello world
```

In the above example, we imported the entire **java.util** package using the import statement. We then used the **ArrayList** class from the **java.util** package in our code to create an instance of the **ArrayList** class and add two strings to it. Finally, we printed the contents of the **ArrayList** using the **System.out.println** statement.

❖ Jar files:

In Java, JAR stands for Java ARchive, whose format is based on the zip format. The JAR files format is mainly used to aggregate a collection of files into a single one.

It is a single cross-platform archive format that handles images, audio, and class files.

✓ Create a JAR file:

In order to create a .jar file, we can use *jar cf command* in the following ways as discussed below:

Syntax:

```
jar cf jarfilename inputfiles
```

Here, cf represents to create the file. For example , assuming our package pack is available in C:\directory , to convert it into a jar file into the pack.jar , we can give the command as:

```
C:\> jar cf pack.jar pack
```

✓ View a JAR file:

Now, pack.jar file is created. In order to view a JAR file '.jar' files, we can use the command as:

Syntax:

```
jar tf jarfilename
```

Here, tf represents the table view of file contents. For example, to view the contents of our pack.jar file, we can give the command:

```
C:/> jar tf pack.jar
```

Now, the contents of pack.jar are displayed as follows:

```
META-INF/
```

```
META-INF/MANIFEST.MF
```

```
pack/
```

```
pack/class1.class
```

```
pack/class2.class
```

```
..
```

```
..
```

Here class1, class2, etc are the classes in the package pack. The first two entries represent that there is a manifest file created and added to pack.jar. The third entry represents the sub-directory with the name pack and the last two represent the files name in the directory pack.

Note: When we create .jar files, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname. META-INF/MANIFEST.MF

This manifest file is useful to specify the information about other files which are packaged.

✓ Extracting a JAR file:

In order to extract the files from a .jar file, we can use the commands below listed:

```
jar xf jarfilename
```

Here, xf represents extract files from the jar files. For example, to extract the contents of our pack.jar file, we can write:

```
C:\> jar xf pack.jar
```

This will create the following directories in C:\

```
META-INF
```

In this directory, we can see class1.class and class2.class.

```
pack
```

✓ Updating a JAR File :

The Jar tool provides a 'u' option that you can use to update the contents of an existing JAR file by modifying its manifest or by adding files. The basic command for adding files has this format as shown below:

Syntax:

```
jar uf jar-file input-file(s)
```

Here 'uf' represents the updated jar file. For example, to update the contents of our pack.jar file, we can write:

```
C:\>jar uf pack.jar
```

✓ **Running a JAR file:**

In order to run an application packaged as a JAR file (requires the Main-class manifest header), the following command can be used as listed:

Syntax:

```
C:\>java -jar pack.jar
```

❖ Exception Handling

- **Exception Handling** in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved.
- Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Exception:

- It 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

Error:

- It 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.

❖ Types of Java Exceptions:

There are mainly two types of exceptions: checked and unchecked. An error is considered as

the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked Exception:

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception:

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error:

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

❖ Java Exception Keywords:

✓ try:

The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

✓ catch:

The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

✓ finally:

The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

✓ throw:

The "throw" keyword is used to throw an exception.

✓ throws:

The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Example:

```
import java.util.Scanner;
class UncheckedExc
{
    public static void main(String args[])
    {
        int a,b,c;
        Scanner s = new Scanner(System.in);
        System.out.println("enter a and b values");
        a=s.nextInt();
        b=s.nextInt();
        try
        {
            c=a/b;
            System.out.println("a/b="+c);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Error="+e);
        }
        finally
        {
            System.out.println("Program executed Sucessfully");
        }
    }
}
```

Output:

```

C:\Program Files\Java\jdk-19\bin>javac UncheckedExc.java

C:\Program Files\Java\jdk-19\bin>java UncheckedExc
enter a and b values
50
5
a/b=10
Program executed Sucessfully

C:\Program Files\Java\jdk-19\bin>java UncheckedExc
enter a and b values
100
0
Error=java.lang.ArithmeticException: / by zero
Program executed Sucessfully

C:\Program Files\Java\jdk-19\bin>_

```

❖ Re-throwing an exception in java:

Sometimes we may need to rethrow an exception in Java. If a catch block cannot handle the particular exception it has caught, we can rethrow the exception. The rethrow expression causes the **originally thrown object to be rethrown**.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the enclosing try block have an opportunity to catch the exception.

Syntax:

```

catch(Exception e)
{
    System.out.println("An exception was thrown");
    throw e;
}

```

Example:

```

public class RethrowException
{
    public static void test1() throws Exception
    {
        System.out.println("The Exception in test1() method");
        throw new Exception("thrown from test1() method");
    }
    public static void test2() throws Throwable
    {
        try
        {
            test1();
        }
        catch(Exception e)
        {

```

```

            System.out.println("Inside test2() method");throw e;
        }
    }
}

```

```
}  
}  
public static void main(String[] args) throws Throwable  
{  
    try  
    {  
        test2();  
    }  
  
    catch(Exception e)  
    {  
        System.out.println("Caught in main");  
    }  
}  
}
```

Output:

The Exception in test1() method

Inside test2() method

Caught in main

UNIT-IV

❖ **Streams:**

Java provides a new additional package in Java 8 called `java.util.stream`. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing `java.util.stream` package.

Stream provides following features:

- Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

❖ **Reading Data from a File Using `FileInputStream`**

- `FileInputStream` class in Java is useful to read data from a file in the form of a sequence of bytes.
- `FileInputStream` is meant for reading streams of raw bytes such as image data.
- For reading streams of characters, consider using `FileReader`.
- The `read()` method of `InputStream` class reads a byte of data from the input stream.
- The next byte of data is returned, or -1 if the end of the file is reached and throws an exception if an I/O error occurs. Refer to the program.

Syntax:

```
public abstract int read()
```

- **Return Value:** This method returns the next byte of data, or -1 if the end of the stream is reached.

➤ **Exception:** IOException – If an I/O error occurs.

• **Invoking the read() method:**

Follow these steps to read data from a file using FileInputStream, which is ultimately the goal of FileInputStreamClass

Step 1:

Attach a file to a FileInputStream as this will enable us to read data from the file as shown below as follows:

```
FileInputStream fileInputStream = new FileInputStream("file.txt");
```

Step 2:

Now, to read data from the file, we should read data from the FileInputStream as shown below:

```
ch = fileInputStream.read();
```

Step 3(a):

When there is no more data available to read further, the read() method returns -1;

Step 3(b):

Then, we should attach the monitor to the output stream. For displaying the data, we can use System.out.print.

```
System.out.print(ch);
```

Implementation:

Original File content: ("bcastudents.txt")

Welcome to bca students

```
import java.io.File;
import java.io.FileInputStream;
public class InputExample
{
    public static void main(String[] args)
    {
        File file = new File("bcastudents.txt");
        try
        {
            FileInputStream input = new FileInputStream(file);
            int ch;

            while ((ch = input.read()) != -1)
```

```

{
System.out.print((char)ch);
}
}
catch (Exception e)
{
e.printStackTrace();
}
}
}

```

Output

Welcome to bca students

❖ FileOutputStream:

- FileOutputStream class belongs to byte stream and stores the data in the form of individual bytes.
- It can be used to create text files. A file represents storage of data on a second storage media like a hard disk or CD.
- Whether or not a file is available or may be created depends upon the underlying platform.
- Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing objects) at a time.
- In such situations, the constructors in this class will fail if the file involved is already open.
- FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.

Important methods:

- **void close()** : Closes this file output stream and releases any system resources associated with this stream.
- **protected void finalize()** : Cleans up the connection to the file, and ensures that the close method of this file output stream is called when there are no more references to this stream.
- **void write(byte[] b)** : Writes b.length bytes from the specified byte array to this file output stream.
- **void write(byte[] b, int off, int len)** : Writes len bytes from the specified byte array starting at offset off to this file output stream.

- **void write(int b)** : Writes the specified byte to this file output stream.

Following steps are to be followed to create a text file that stores some characters (or text):

1. **Reading data:** First of all, data should be read from the keyboard. For this purpose, associate the keyboard to some input stream class. The code for using `DataInputStream` class for reading data from the keyboard is as:

```
DataInputStream dis =new DataInputStream(System.in);
```

Here `System.in` represent the keyboard which is linked with `DataInputStream` object

2. **Send data to OutputStream:** Now , associate a file where the data is to be stored to some output stream. For this , take the help of `FileOutputStream` which can send data to the file. Attaching the `file.txt` to `FileOutputStream` can be done as:

```
FileOutputStreamfout=new FileOutputStream("file.txt");
```

3. **Reading data from DataInputStream:** The next step is to read data from `DataInputStream` and write it into `FileOutputStream` . It means read data from `dis` object and write it into `fout` object, as shown here:

```
ch=(char)dis.read();
```

```
fout.write(ch);
```

4. **Close the file:** Finally, any file should be closed after performing input or output operations on it, else the data of the may be corrupted. Closing the file is done by closing the associated streams. For example, `fout.close()`: will close the `FileOutputStream` ,hence there is no way to write data into the file.

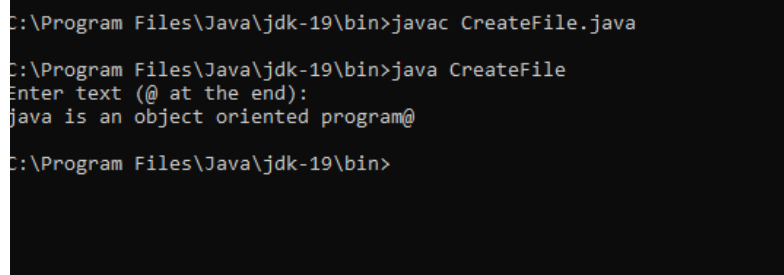
Implementation:

```

import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
class CreateFile
{
    public static void main(String[] args) throws IOException
    {
        DataInputStream dis=new DataInputStream(System.in);
        FileOutputStream fout=new FileOutputStream("file.txt");
        BufferedOutputStream bout=new BufferedOutputStream(fout,1024);
        System.out.println("Enter text (@ at the end):");
        char ch;
        while((ch=(char)dis.read())!='@')
        {
            bout.write(ch);
        }
        //close the file
        bout.close();
    }
}

```

Output:



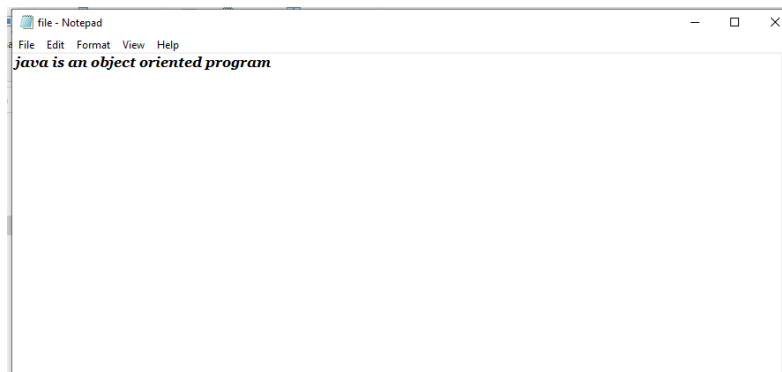
```

C:\Program Files\Java\jdk-19\bin>javac CreateFile.java

C:\Program Files\Java\jdk-19\bin>java CreateFile
Enter text (@ at the end):
java is an object oriented program@

C:\Program Files\Java\jdk-19\bin>

```

❖ **Zippping and Unzipping files:**

Java provides the **java.util.zip** package for zipping and unzipping the files. The class **ZipOutputStream** will be useful for compressing. It is an output stream filter which will write files to any File output stream in zip format. The class **ZipInputStream** will be useful for decompressing. It is an input stream which will read files that are in zip format.

❖ **Serialization of Objects:**

- **Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.
- The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object.
- The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.
- For serializing the object, we call the **writeObject()** method of *ObjectOutputStream* class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class. We must have to implement the *Serializable* interface for serializing the object.
 - ✓ **java.io.Serializable interface:**
- **Serializable** is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The **Cloneable** and **Remote** are also marker interfaces.
- The **Serializable** interface must be implemented by the class whose object needs to be persisted.
- The **String** class and all the wrapper classes implement the *java.io.Serializable* interface by default.
 - ✓ **ObjectOutputStream class:**

The *ObjectOutputStream* class is used to write primitive data types, and Java objects to an *OutputStream*. Only objects that support the *java.io.Serializable* interface can be written to streams.

- **Constructor:**

public ObjectOutputStream(OutputStream out) throws IOException {}:

It creates an ObjectOutputStream that writes to the specified OutputStream.

- **Methods:**

1. **public final void writeObject(Object obj) throws IOException {}:**

It writes the specified object to the ObjectOutputStream.

2. **public void flush() throws IOException {}:**

It flushes the current output stream.

3. **public void close() throws IOException {}:**

It closes the current output stream.

✓ **ObjectInputStream class:**

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

- **Constructor:**

public ObjectInputStream(InputStream in) throws IOException {}:

It creates an ObjectInputStream that reads from the specified InputStream.

- **Methods**

1. **public final Object readObject() throws IOException, ClassNotFoundException{}:**

It reads an object from the input stream.

2. **public void close() throws IOException {}:**

It closes ObjectInputStream.

Example:

In the example, *Student* class implements Serializable interface. Now its objects can be converted into stream. The main class implementation of is showed in the next code.

Student.java

```
import java.io.Serializable;
public class Student implements Serializable
{
    int id;
    String name;
    public Student(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

In this example, we are going to serialize the object of *Student* class from above code. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

Persist.java

```

import java.io.*;
class Persist
{
    public static void main(String args[])
    {
        try
        {
            //Creating the object
            Student s1 =new Student(211,"ravi");
            //Creating stream and writing the object
            FileOutputStream fout=new FileOutputStream("f.txt");
            ObjectOutputStream out=new ObjectOutputStream(fout);
            out.writeObject(s1);
            out.flush();
            //closing the stream
            out.close();
            System.out.println("success");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

Output:

success

❖ Counting Number of Characters in a File:

- Counting the number of characters is essential because almost all the text boxes that rely on user input have certain limitations on the number of characters inserted.
- For example, the character limit on a Facebook post is 63206 characters. Whereas for a tweet on Twitter, the character limit is 140 characters, and the character limit is 80 per post for Snapchat.
- Determining character limits become crucial when the tweet and Facebook post updates are being done through APIs.

In-built Functions Used**1. File(String pathname):**

This function is present under the **java.io.File** package. It creates a new File instance by converting the given pathname string into an abstract pathname.

Syntax:

```
public File(String pathname)
```

Parameters:

pathname - A pathname string

2. FileInputStream(File file) :

This function is present under the **java.io.FileInputStream** package. It creates a **FileInputStream** by opening a connection to an actual file named by the **File** object file in the file system.

Syntax:

```
public FileInputStream(File file) throws FileNotFoundException
```

Parameters:

file - the file to be opened for reading.

Throws:

- **FileNotFoundException** - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.
- **SecurityException** - if a security manager exists and its **checkRead** method denies read access to the file.

3. InputStreamReader(InputStream in):

This function is present under the **java.io.InputStreamReader** package. It creates an **InputStreamReader** that uses the default charset.

Syntax:

```
public InputStreamReader(InputStream in)
```

Parameters:

in - An **InputStream**

4. BufferedReader(Reader in):

This function is present under the **java.io.BufferedReader** package. It creates a buffering character-input stream that uses a default-sized input buffer.

Syntax:

```
public BufferedReader(Reader in)
```

Parameters:

in - A **Reader**

Example:

```
// Java program to count the
// number of lines, words, sentences,
// characters, and whitespaces in a file
```

```
import java.io.*;
public class Test
{
    public static void main(String[] args)throws IOException
```

```

{
File file = new File("C:\\Users\\hp\\Desktop\\TextReader.txt");
FileInputStreamfileInputStream = new FileInputStream(file);
InputStreamReaderinputStreamReader = new InputStreamReader(fileInputStream);
    BufferedReaderbufferedReader = new BufferedReader(inputStreamReader);
String line;
int wordCount = 0;
int characterCount = 0;
int paraCount = 0;
int whiteSpaceCount = 0;
int sentenceCount = 0;
while ((line = bufferedReader.readLine()) != null)
{
if (line.equals(""))
{
    paraCount += 1;
}
else
{
    characterCount += line.length();
    String words[] = line.split("\\s+");
    wordCount += words.length;
    whiteSpaceCount += wordCount - 1;
    String sentence[] = line.split("[!?:.]+");
    sentenceCount += sentence.length;
}
}
if (sentenceCount>= 1)
{
paraCount++;
}
System.out.println("Total word count = "+ wordCount);
System.out.println("Total number of sentences = "+ sentenceCount);
System.out.println("Total number of characters = "+ characterCount);
System.out.println("Number of paragraphs = "+ paraCount);
System.out.println("Total number of whitespaces = "+ whiteSpaceCount);
}
}

```

The **TextReader.txt** file contains the following data -
Hello Ram. My name is Nishkarsh Gandhi.

Output:

```
C:\Users\hp>cd Desktop
```

```
C:\Users\hp\Desktop>javac Test.java
```

```
C:\Users\hp\Desktop>java Test
```

```
Total word count=15
```

```
Total number of sentences=3
```

```
Total number of characters=94
```

```
Number of paragraphs=2
```

```
Total number of whitespaces=20
```

```
C:\Users\hp\Desktop>
```

❖ File Copy:

There are mainly 3 ways to copy files using java language. They are as given below:

1. Using File Stream (Naive method)
2. Using FileChannel Class
3. Using Files class.

Method 1: Using File Stream (Naive method)

This is a naive method where we are using a file input stream to get input characters from the first file and a file output stream to write output characters to another file. This is just like seeing one file and writing onto another.

Example:

```
// Java Program to Copy file using File Stream
```

```
// Importing input output classes
```

```
import java.io.*;
```

```
// Main Class
```

```
public class GFG
```

```
{
```

```
// Main driver method
```

```
public static void main(String[] args)throws IOException
```

```
{
```

```
// Creating two stream
```

```
// one input and other output

FileInputStreamfis = null;

FileOutputStreamfos = null;

// Try block to check for exceptions

try

{

// Initializing both the streams with

// respective file directory on local machine

// Custom directory path on local machine

    fis = new FileInputStream("C:\\Users\\Dipak\\Desktop\\input.txt");

// Custom directory path on local machine

    fos = new FileOutputStream("C:\\Users\\Dipak\\Desktop\\output.txt");

int c;

// Condition check

// Reading the input file till there is input

// present

while ((c = fis.read()) != -1)

{

// Writing to output file of the specified

// directory

    fos.write(c);

}
```

```
// By now writing to the file has ended, so

// Display message on the console

    System.out.println("copied the file successfully");

}

// Optional finally keyword but is good practice to

// empty the occupied space is recommended whenever

// closing files,connections,streams

finally

{

// Closing the streams

if (fis != null)

{

// Closing the fileInputStream

    fis.close();

}

    if (fos != null)

    {

// Closing the fileOutputStream

        fos.close();

    }

}

}
```



```
}
```

Method 2: Using FileChannel Class

This is a class present in java.nio, channels package and is used to write, modify, read files. The objects of this class create a seekable file channel through which all these activities are performed. This class basically provides two methods named as follows:

- **transferFrom(ReadableByteChannelsrc, long position, long count):** Transfers bytes to the channel which calls this method from the src channel. This is called by destination channel. The position is the place of a pointer from where the copy actions are to be started. Count specifies the size of the file which is nearly equal to the amount of content it contains.
- **transferTo(long position, long count, WritableByteChannel target):** Transfers bytes from the source or method calling channel to the destination channel of the file. This method is mainly called using the source channel and Count mentions the size of the source file and position from where the copy is to be made

Hence, we can use any one of the two methods to transfer files data and copy them.

Example:

```
// Java Program to Copy Files Using FileChannel Class
// Importing java.nio package for network linking
// Importing input output classes
import java.io.*;
import java.nio.channels.FileChannel;
// Main Class
public class GFG
{
    // Main driver method
    public static void main(String[] args)throws IOException
    {
        // Creating two channels one input and other output
        // by creating two objects of FileChannel Class
        FileChannelsrc= new FileInputStream(
"C:\\Users\\Dipak\\Desktop\\input.txt").getChannel();
        FileChanneldest= new FileOutputStream(
"C:\\Users\\Dipak\\Desktop\\output.txt").getChannel();
        // Try block to check for exceptions
        try
```

```

{
// Transferring files in one go from source to
// destination using transferFrom() method
dest.transferFrom(src, 0, src.size());
// we can also use transferTo
// src.transferTo(0,src.size(),dest);
}
// finally keyword is good practice to save space in
// memory by closing files, connections, streams
finally
{
// Closing the channels this makes the space
// free
// Closing the source channel
src.close();
// Closing the destination channel
dest.close();
}
}
}

```

Method 3: Using Files Class

This is a class present in java.nio.File package. This class provides 3 methods to copy the files which are as follows:

- **copy(InputStream in, Path target):** Copies all bytes of data from the input file stream to the output path of the output file. It cannot be used to make a copy of a specified part in a source file. Here we are not required to create an output file. It is automatically created during the execution of the code.
- **copy(Path source, OutputStream out):** Copies all bytes from the file specified in the path source to the output stream of the output file.
- **copy(Path source, Path target):** Copies files using the path of both source and destination files. No need to create the output file here also.

Example:

```

import java.nio.file.Files;
import java.io.*;
// save the file named as GFG.java
public class GFG
{
// main method
    public static void main(String[] args) throws IOException

```

```

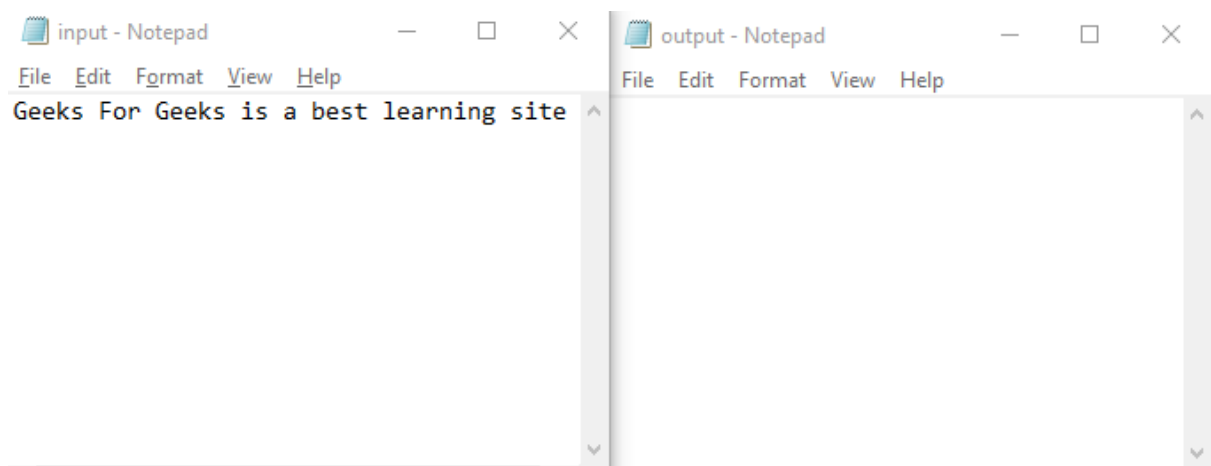
{
// creating two channels
// one input and other output
    File src = new File("C:\\Users\\Dipak\\Desktop\\input.txt");
    File dest = new File("C:\\Users\\Dipak\\Desktop\\output.txt");
// using copy(InputStream,Path Target); method
    Files.copy(src.toPath(), dest.toPath());
// here we are not required to have an
// output file at the specified target.
// same way we can use other method also.
}
}

```

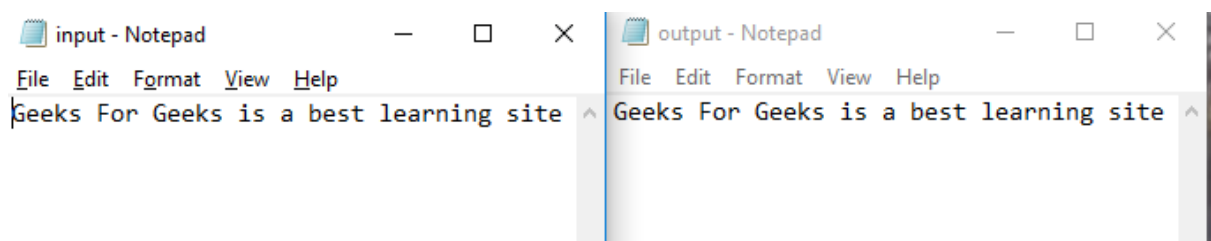
Output: for Method 1,Method 2 and Method 3

copied the file successfully

For the above programs, we require one input.txt and one output.txt file.
Initially, both the text files look like this



After successful execution of the program,



❖ File Class:

- Java File class is Java's representation of a file or directory pathname. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.

- Java File class contains several methods for working with the pathname, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.
 - It is an abstract representation of files and directory pathnames.
 - A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the `getParent()` method of this class.
 - First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
 - Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

Constructors of Java File Class:

- **File(File parent, String child):** Creates a new File instance from a parent abstract pathname and a child pathname string.
- **File(String pathname):** Creates a new File instance by converting the given pathname string into an abstract pathname.
- **File(String parent, String child):** Creates a new File instance from a parent pathname string and a child pathname string.
- **File(URI uri):** Creates a new File instance by converting the given file: URI into an abstract pathname.

Example:

```
import java.io.*;

public class FileDemo
{
    public static void main(String[] args)
    {
        try
        {
            File file = new File("javaFile123.txt");
            if (file.createNewFile())
            {
                System.out.println("New File is created!");
            }
        }
        else
        {
            System.out.println("File already exists.");
        }
    }
}
```

```

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

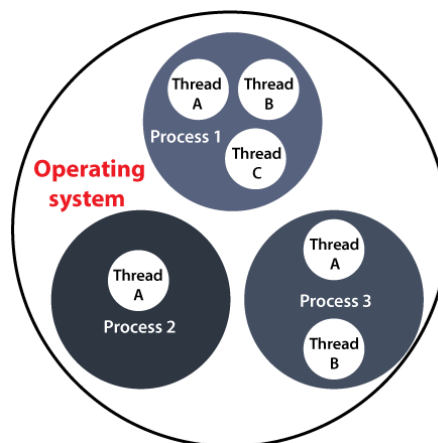
```

Output:

New File is created!

❖ Thread

- Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.
- A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.
- In order to perform complicated tasks in the background, we used the **Threadconcept in Java**. All the tasks are executed without affecting the main program.
- In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



- Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads.
- All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

- **Single Tasking:**

Single-tasking in Java threads is when only one thread is executed at a time. This is achieved by using synchronization and locks to ensure that only one thread can access a particular resource at a time. Single-tasking is useful in situations where multiple threads accessing a resource could cause data corruption or other issues.

- **Multi Tasking:**

Multitasking in Java threads is when multiple threads are executed simultaneously. This is useful in situations where an application needs to perform multiple tasks at the same time, such as in gaming, animation, or multimedia applications. Multitasking is achieved by creating multiple threads and running them concurrently.

❖ **Creating and running a Thread :**

We can create Threads in java using two ways, namely :

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class :

We can run Threads in Java by using Thread Class, which provides constructors and methods for creating and performing operations on a Thread, which extends a Thread class that can implement Runnable Interface. We use the following constructors for creating the Thread:

- Thread
- Thread(Runnable r)
- Thread(String name)
- Thread(Runnable r, String name)

Example:

```
class MyThread extends Thread
{

public void run()
{
System.out.println("My Thread Running...");
}
}

class ThreadExample
{
public static void main(String[] args)
{
MyThread mt = new MyThread();
mt.start();
}
```

}Output:

My Thread Running...

2.By Implementing a Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to

be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Example:

```
class MyThread2 implements Runnable
{

public void run()
{
System.out.println("interface Thread Running...");
}
}

class ThreadExample2
{
public static void main(String[] args)
{
MyThread2 mt2 = new MyThread2();
Thread t = new Thread (mt2);
t.start();
}
}
```

Output:

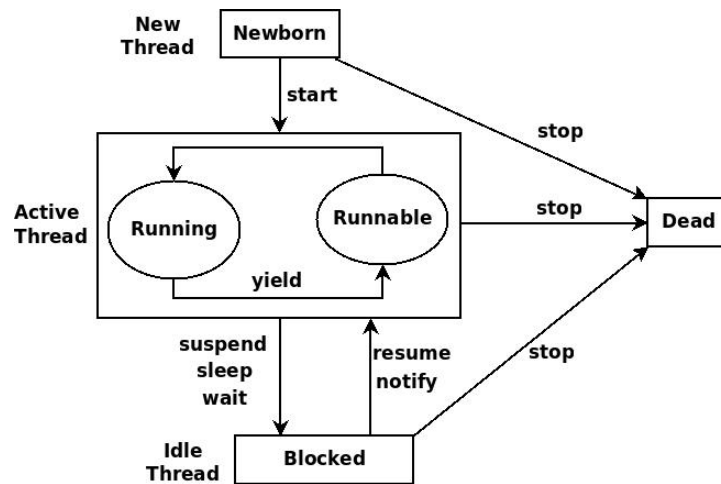
interface Thread Running...

❖ **Lifecycle of Thread:**

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked/Waiting
4. Timed Waiting
5. Terminated

The diagram shown below represents various states of a thread at any instant in time.



1. New Thread:

When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

2. Runnable State:

A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

3. Blocked/Waiting state:

When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting

4. Timed Waiting:

A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

5. Terminated State:

A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Implementing the Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as a summary of which is given below:

1. New

Thread state for a thread that has not yet started.

public static final Thread.State NEW

2. Runnable

Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

public static final Thread.State RUNNABLE

3. Blocked

Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling `Object.wait()`.

public static final Thread.State BLOCKED

4. Waiting

Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

- `Object.wait` with no timeout
- [Thread.join](#) with no timeout
- `LockSupport.park`

public static final Thread.State WAITING

5. Timed Waiting

Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

- `Thread.sleep`
- `Object.wait` with timeout
- `Thread.join` with timeout
- `LockSupport.parkNanos`
- `LockSupport.parkUntil`

public static final Thread.State TIMED_WAITING

6. Terminated

Thread state for a terminated thread. The thread has completed execution.

Declaration: public static final Thread.State TERMINATED

❖ Thread Communication/Inter- Thread Communication:

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of Object class:
 - wait()
 - notify()
 - notifyAll()

1) wait() method:

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method:

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

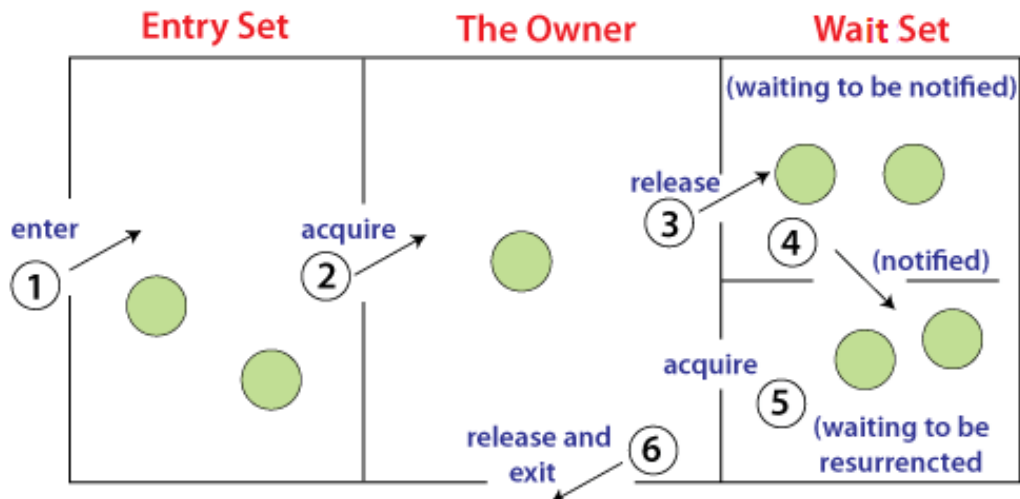
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication:



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

❖ Thread Priorities:

- Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority:

- `public final int getPriority():`

The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

- `public final void setPriority(int newPriority):`

The `java.lang.Thread.setPriority()` method updates or assigns the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`

2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example:

```
// Importing the required classes
import java.lang.*;
public class ThreadPriorityExample extends Thread
{
    // Method 1
    // Whenever the start() method is called by a thread
    // the run() method is invoked
    public void run()
    {
        // the print statement
        System.out.println("Inside the run() method");
    }
    // the main method
    public static void main(String args[])
    {
        // Creating threads with the help of ThreadPriorityExample class
        ThreadPriorityExample th1 = new ThreadPriorityExample();
        ThreadPriorityExample th2 = new ThreadPriorityExample();
        ThreadPriorityExample th3 = new ThreadPriorityExample();
        // We did not mention the priority of the thread.
        // Therefore, the priorities of the thread is 5, the default value
        // 1st Thread
        // Displaying the priority of the thread
        // using the getPriority() method
        System.out.println("Priority of the thread th1 is : " + th1.getPriority());

        // 2nd Thread
        // Display the priority of the thread
        System.out.println("Priority of the thread th2 is : " + th2.getPriority());
        // 3rd Thread
        // // Display the priority of the thread
        System.out.println("Priority of the thread th2 is : " + th2.getPriority());
        // Setting priorities of above threads by
        // passing integer arguments
        th1.setPriority(6);
        th2.setPriority(3);
```

```

th3.setPriority(9);
// 6
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
// 3
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// 9
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
// Main thread
// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
}
}

```

Output:

```

Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10

```

❖ Daemon Threads

- Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection.
- Daemon thread in Java is also a service provider thread that provides services to the user thread.
- Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.
- By default, the main thread is always non-daemon but for all the remaining threads, daemon nature will be inherited from parent to child.

- That is, if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

Example of Daemon Thread in Java:

Garbage collection in Java (gc), finalizer, etc.

Properties of Java Daemon Thread:

- They can not prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution.
- If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
- It is an utmost low priority thread.

Methods of Daemon Thread:

1. void setDaemon(boolean status):

This method marks the current thread as a daemon thread or user thread. For example, if I have a user thread tU then tU.setDaemon(true) would make it a Daemon thread. On the other hand, if I have a Daemon thread tD then calling tD.setDaemon(false) would make it a user thread.

Syntax:

```
public final void setDaemon(boolean on)
```

Parameters:

on: If true, marks this thread as a daemon thread.

Exceptions:

- **IllegalThreadStateException:** if only this thread is active.
- **SecurityException:** if the current thread cannot modify this thread.

2. boolean isDaemon():

This method is used to check that the current thread is a daemon. It returns true if the thread is Daemon. Else, it returns false.

Syntax:

```
public final boolean isDaemon()
```

Returns:

This method returns true if this thread is a daemon thread; false otherwise

Example:

```
// Java program to demonstrate the usage of
// setDaemon() and isDaemon() method.
public class DaemonThread extends Thread
{
    public DaemonThread(String name)
    {
        super(name);
    }
}
```

```

    }
    public void run()
    {
        // Checking whether the thread is Daemon or not
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon thread");
        }

        else
        {
            System.out.println(getName() + " is User thread");
        }
    }
    public static void main(String[] args)
    {
        DaemonThread t1 = new DaemonThread("t1");
        DaemonThread t2 = new DaemonThread("t2");
        DaemonThread t3 = new DaemonThread("t3");
        // Setting user thread t1 to Daemon
        t1.setDaemon(true);
        // starting first 2 threads
        t1.start();
        t2.start();

        // Setting user thread t3 to Daemon
        t3.setDaemon(true);
        t3.start();
    }
}

```

Output:

t1 is Daemon thread
t3 is Daemon thread
t2 is User thread

❖ Deadlock of Threads:

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Figure - 1

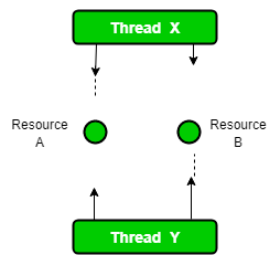
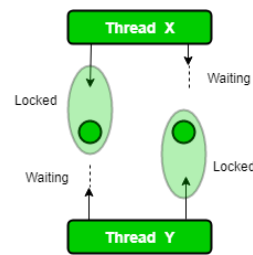


Figure - 2

**Example:**

```

public class TestDeadlockExample1
{
    public static void main(String[] args)
    {
        final String resource1 = "ratan jaiswal";
        final String resource2 = "vimal jaiswal";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread()
        {
            public void run()
            {
                synchronized (resource1)
                {
                    System.out.println("Thread 1: locked resource 1");
                    try
                    {
                        Thread.sleep(100);
                    } catch (Exception e)
                    {
                    }

                    synchronized (resource2)
                    {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        };

        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread()
        {
            public void run()
            {
                synchronized (resource2)

```



```

    {
        System.out.println("Thread 2: locked resource 2");
        try
        {
            Thread.sleep(100);
        }
        catch (Exception e)
        {
        }

        synchronized (resource1)
        {
            System.out.println("Thread 2: locked resource 1");
        }
    }
};
t1.start();
t2.start();
}
}

```

Output:

Thread 1: locked resource 1

Thread 2: locked resource 2

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks:** We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks:** The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join:** A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

❖ Thread Group:

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads. Java thread group is implemented by *java.lang.ThreadGroup* class.

- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

Constructors:

1. **public ThreadGroup(String name):** Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.
2. **public ThreadGroup(ThreadGroup parent, String name):** Creates a new thread group. The parent of this new group is the specified thread group.

Methods:

Here is the list of some important methods available in *java.lang.ThreadGroup*:

1. **int activeCount():**

This method returns the number of active running threads available in a given thread group.

2. **int activeGroupCount():**

This method returns the number of active thread groups running.

3. **destroy():**

This method destroys the thread group and its sub groups if available.

4. **int enumerate(Thread arr[]):**

Call to this method puts the thread available in invoking thread group into the group array of threads.

5. **int enumerate(Thread arr[], boolean recurse):**

Call to this method puts the thread available in invoking thread group into the group array of threads; if the recursive flag is true, then threads in subgroups

are also added to the group.

6. `int enumerate(ThreadGroup[] thgrp):`

This method puts the subgroups of the invoking thread group into the thread group array.

7. `int enumerate(ThreadGroup[] thgrp, boolean recursive):`

This method puts the subgroups of the invoking thread group into the thread group array; if the recursive flag is set to true, then all subgroups of subgroups are added to the group array.

Example:

```
// Java code illustrating activeCount() method
import java.lang.*;
class NewThread extends Thread
{
    NewThread(String threadname, ThreadGroup tgroup)
    {
        super(tgroup, threadname);
        start();
    }
    public void run()
    {
        for (int i = 0; i < 1000; i++)
        {
            try
            {
                Thread.sleep(10);
            }
            catch (InterruptedException ex)
            {
                System.out.println("Exception encountered");
            }
        }
    }
}
public class ThreadGroupDemo
{
    public static void main(String arg[])
    {
        // creating the thread group
        ThreadGroup tgroup = new ThreadGroup("parent thread group");
```

```

        NewThread t1 = new NewThread("one", gfg);
        System.out.println("Starting one");
        NewThread t2 = new NewThread("two", gfg);
        System.out.println("Starting two");
        // checking the number of active thread
        System.out.println("number of active thread: "+ gfg.activeCount());
    }
}

```

Output:

Starting one
 Starting two
 number of active thread: 2

❖ Daemon Threads:

- Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection.
- Daemon thread in Java is also a service provider thread that provides services to the user thread.
- Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.
- By default, the main thread is always non-daemon but for all the remaining threads, daemon nature will be inherited from parent to child.
- That is, if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

Example of Daemon Thread in Java:

Garbage collection in Java (gc), finalizer, etc.

Properties of Java Daemon Thread:

- They can not prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution.
- If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
- It is an utmost low priority thread.

Methods of Daemon Thread:**1. void setDaemon(boolean status):**

This method marks the current thread as a daemon thread or user thread. For example, if I have a user thread tU then tU.setDaemon(true) would make it a

Daemon thread. On the other hand, if I have a Daemon thread tD then calling tD.setDaemon(false) would make it a user thread.

Syntax:

```
public final void setDaemon(boolean on)
```

Parameters:

on: If true, marks this thread as a daemon thread.

Exceptions:

- **IllegalThreadStateException:** if only this thread is active.
- **SecurityException:** if the current thread cannot modify this thread.

2. boolean isDaemon():

This method is used to check that the current thread is a daemon. It returns true if the thread is Daemon. Else, it returns false.

Syntax:

```
public final boolean isDaemon()
```

Returns:

This method returns true if this thread is a daemon thread; false otherwise

Example:

```
// Java program to demonstrate the usage of
// setDaemon() and isDaemon() method.
public class DaemonThread extends Thread
{
    public DaemonThread(String name)
    {
        super(name);
    }
    public void run()
    {
        // Checking whether the thread is Daemon or not
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon thread");
        }
        else
        {
            System.out.println(getName() + " is User thread");
        }
    }
    public static void main(String[] args)
    {
```

```
DaemonThread t1 = new DaemonThread("t1");
DaemonThread t2 = new DaemonThread("t2");
DaemonThread t3 = new DaemonThread("t3");
// Setting user thread t1 to Daemon
t1.setDaemon(true);
// starting first 2 threads
t1.start();
t2.start();

// Setting user thread t3 to Daemon
t3.setDaemon(true);
t3.start();
    }
}
```

Output:

t1 is Daemon thread
t3 is Daemon thread
t2 is User thread

UNIT-V

❖ **Applets:**

- Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side. All applets are sub-classes (either directly or indirectly) of `java.applet.Applet` class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at `main()` method.
- Output of an applet window is not performed by `System.out.println()`. Rather it is handled with various AWT methods, such as `drawString()`.

Advantages of Applet:

There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet:

- Plugin is required at client browser to execute applet.

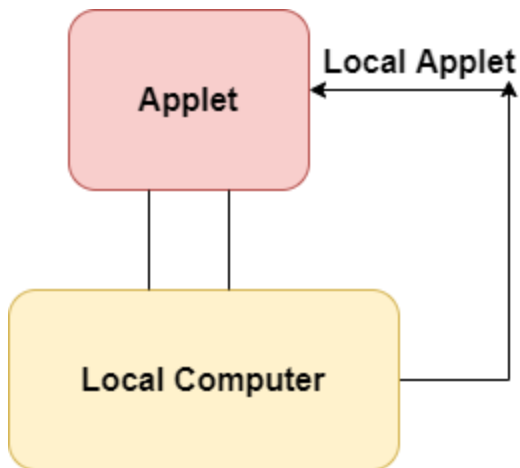
1.Types of Applets:

There are two types of applets in java:

- i) Local Applet
- ii) Remote Applet

i) Local Applet:

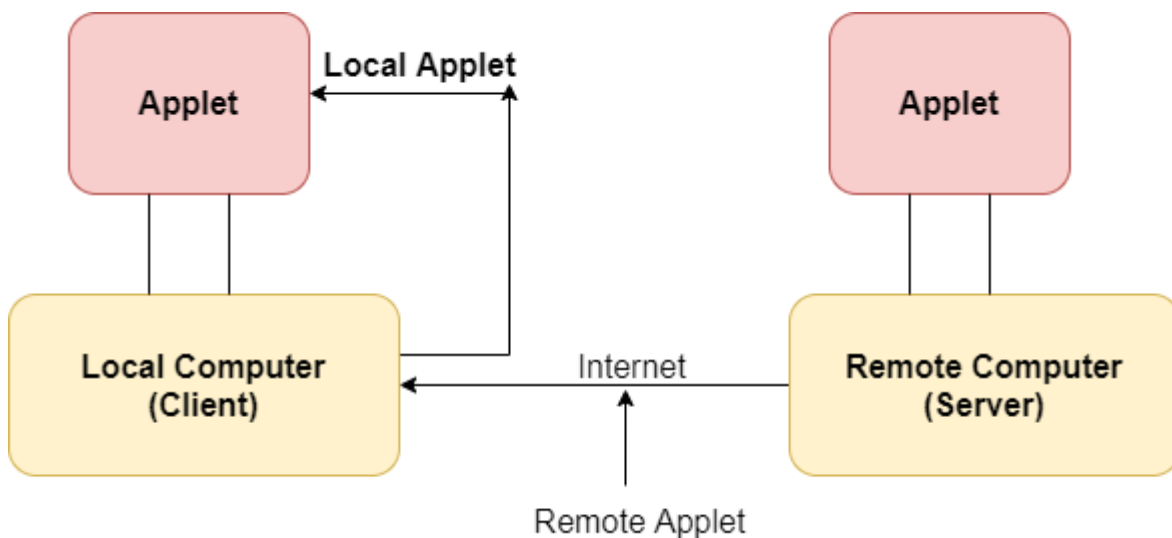
An applet developed locally and stored in a local system is known as a local applet. When a web page is trying to find a local applet, it does not need to use the internet and therefore the local system does not require the internet connection.



ii) Remote Applet:

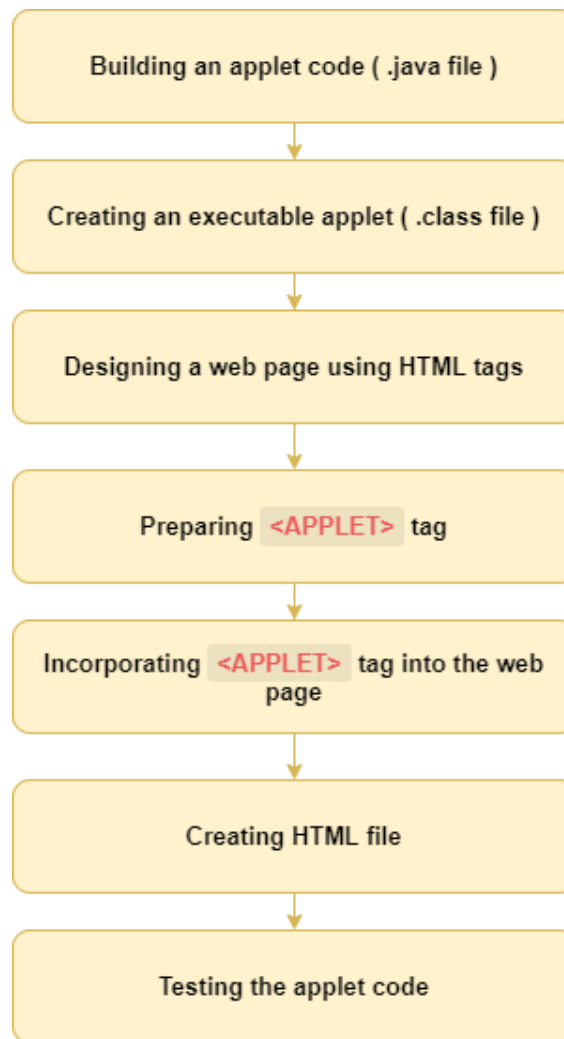
A remote applet is that which is developed by someone else and stored on a remote computer connected to the internet. If our system is connected to the internet, we can download the remote applet onto our system via the internet and run it. In order to locate and load a remote applet, we must know the applet's address on the web. This is known as Uniform Resource Locator (URL).

CODEBASE = `https : // www.shapeai.tech / applets`



❖ Steps Involved in developing and testing in applets

Before we try to write applets, we must ensure that java is installed properly and ensure that either the Java applet viewer or a Java-enabled browser is available. The steps involved in developing and testing in applets:



❖ Creating an Applet:

- **Java Applet Class:**
- For Creating any applet in Java, we use the **java.applet.Applet class**. It has four Methods in its Life Cycle of Java Applet. The applet can be executed using the applet viewer utility provided by JDK.
- A Java Applet was created using the Applet class, i.e., part of the java.applet package.
- The Applet class provides a standard interface between applets and their environment. The Applet class is the superclass of an applet that is embedded in a Web page or viewed by the Java Applet Viewer.
- The Java applet class gives several useful methods to give you complete control over the running of an Applet. Like initializing and destroying an applet, It also

provides ways that load and display Web Colourful images and methods that load and play audio and Videos Clips and Cinematic Videos.

In Java, there are two types of Applet

1. Java Applets based on the AWT(Abstract Window Toolkit) packages by extending its Applet class
2. Java Applets is based on the Swing package by extending its JApplet Class in it.

- **To run an Applet:**

There are two ways to execute a Java Applet:

- By using an HTML file
- By using the appletviewer tool

Program:

java code : saved as BcaApp.java

```
import java.applet.*;
import java.awt.*;
public class BcaApp extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString("welcome to bca students", 300, 150);
    }
}
```

Html code : saved as demo.html

```
<html>
<head>
<title>HTML applet tag</title>
</head>
<body>
<applet code="BcaApp.class" width="300" height="300"> </applet>
</body>
</html>
```

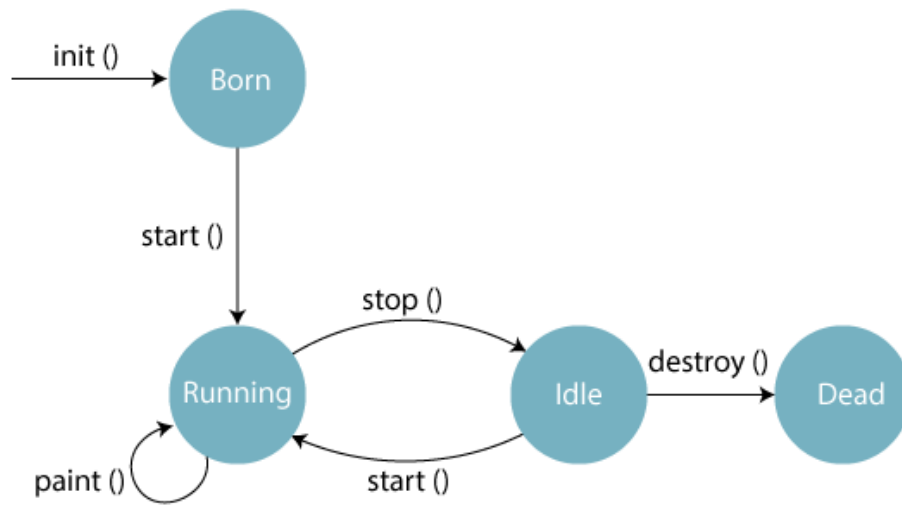
❖ Applet Life Cycle in Java

In Java, an [applet](#) is a special type of program embedded in the web page to generate dynamic content. Applet is a class in Java.

The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has

five core methods namely `init()`, `start()`, `stop()`, `paint()` and `destroy()`. These methods are invoked by the browser to execute.

Methods of Applet Life Cycle



There are five methods of an applet life cycle, and they are:

- **init():** The `init()` method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the `init()` method within the applet.
- **start():** The `start()` method contains the actual code of the applet and starts the applet. It is invoked immediately after the `init()` method is invoked. Every time the browser is loaded or refreshed, the `start()` method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the `init()` method is invoked.
- **stop():** The `stop()` method stops the execution of the applet. The `stop()` method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the `stop()` method is invoked. When we go back to that page, the `start()` method is invoked again.
- **destroy():** The `destroy()` method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.
- **paint():** The `paint()` method belongs to the `Graphics` class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the `start()` method and when the browser or applet windows are resized.

Sequence of method execution when an applet is executed:

1. init()
2. start()
3. paint()

Sequence of method execution when an applet is executed:

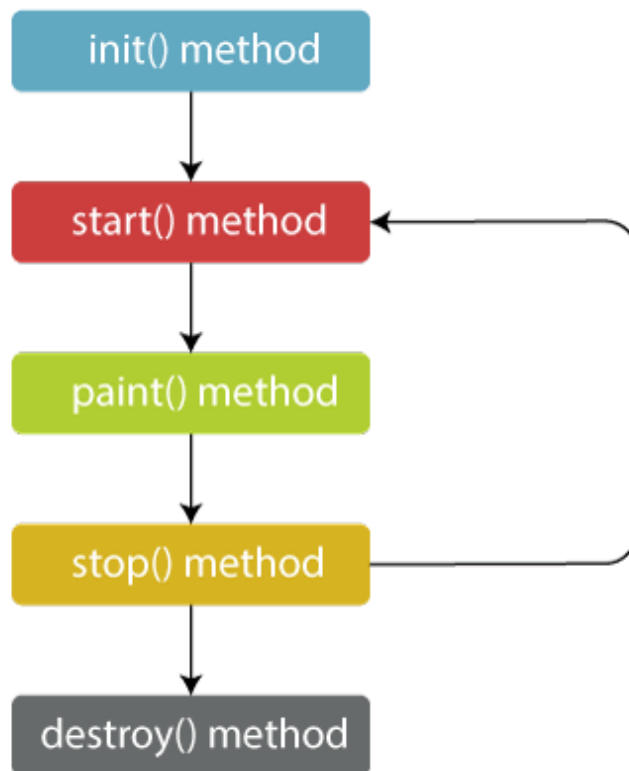
1. stop()
2. destroy()

Applet Life Cycle Working

- The Java plug-in software is responsible for managing the life cycle of an applet.
- An applet is a Java application executed in any web browser and works on the client-side. It doesn't have the main() method because it runs in the browser. It is thus created to be placed on an HTML page.
- The init(), start(), stop() and destroy() methods belongs to the **applet.Applet** class.
- The paint() method belongs to the **awt.Component** class.
- In Java, if we want to make a class an Applet class, we need to extend the **Applet**
- Whenever we create an applet, we are creating the instance of the existing Applet class. And thus, we can use all the methods of that class.

Flow of Applet Life Cycle:

These methods are invoked by the browser automatically. There is no need to call them explicitly.

**Example:**

java code : saved as AppletExp1.java

```
import java.applet.*;
import java.awt.*;
public class AppletExp1 extends Applet
{
    public void init()
    {
        System.out.println("Initializing an applet");
    }

    public void start()
    {
        System.out.println("Starting an applet");
    }
    public void stop()
    {
        System.out.println("Stopping an applet");
    }
}
```

```

    public void destroy()
    {
        System.out.println("Destroying an applet");
    }
}

```

Html code : saved as App.html

```

<applet code="AppletExp1" width=600 height=300>
</applet>

```

❖ <APPLET> tag

- An applet is a Java code that must be executed within another program. It mostly executes in a Java-enabled web browser.
- An applet can be embedded in an HTML document using the tag <APPLET> and can be invoked by any web browser.

Attributes of Applet tag:

An applet tag contains many attributes that give you a good control over the your applet.

i. CODE:

This is used to specify the name of the your applet's **.class** file. This is a **must** required attribute in an <applet> tag.

Example:

```
<applet code ="Applet1" width= 400 height=400>
```

ii. WIDTH:

This is used to specify the width of our applet window to be displayed within a browser or appletviewer. This is also a **must** required attribute in an <applet> tag. **Width** is specified in **pixels**.

iii. HEIGHT:

This is used to specify the height of our applet window in pixels, This is also a must required attribute in an <applet> tag.

Example:

```
<applet code ="Applet1" width= 400 height=400>
```

iv. NAME:

This is used to give a name to an applet class and this name will be used when some

other applet wish to communicate with this applet. This is also an optional attribute in <applet> tag.

Example:

```
<applet code ="Applet1" width= 400 height=400 name = "firstApplet">
</applet>
```

v. CODEBASE:

This is used to specify the directory location containing your applet's .class file.

Example:

```
<applet code ="Applet1" width= 400 height=400
codebase="http://MyWebsite.com//Folder">
</applet>
```

vi. ALT:

This is used to specify the text message that will be displayed in a situation when the browser has recognized the applet tag but it still cannot load the applet. This is an optional attribute in <applet> tag.

Example:

```
<applet code ="Applet1" width= 400 height=400 alt ="Sorry, we can't load this
applet">
</applet>
```

vii. VSPACE:

This is used to specify the vertical space allotted above and below the applet window. VSPACE is specified in **pixels** and this is also an **optional** tag.

Example:

```
<applet code ="Applet1" width= 400 height=400 VSPACE= 50>
</applet>
```

viii. HSPACE:

This is used to specify the horizontal space on the either sides(right and left) of the applet window. HSPACE is also specified in **pixels** and this is also an **optional** attribute in the <applet> tag.

Example:

```
<applet code ="Applet1" width= 400 height=400 HSPACE= 50>
</applet>
```

ix. ALIGN:

This is used to specify how an applet will be aligned with respect to the surrounding content. ALIGN attribute takes any of the values : TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSBOTTOM(absolute bottom) etc

Example:

```
<applet code ="BcaApp.class" width= 300 height =300 ALIGN= TOP>
</applet>
```

Program:

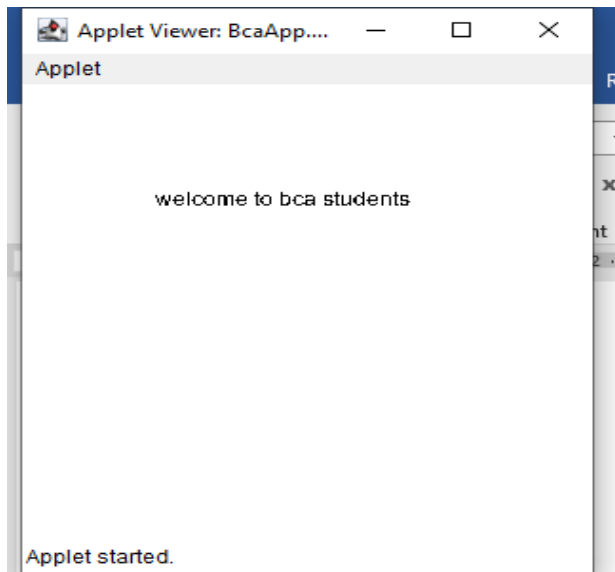
```
import java.applet.*;
import java.awt.*;
public class BcaApp extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString("welcome to bca students", 300, 150);
    }
}
```

Compile: javac BcaApp.java

Html code : saved as AppBca.html

```
<html>
<head>
<title>AppBca</title>
</head>
<body>
<applet code="BcaApp.class" width="300" height="300"> </applet>
</body>
</html>
```

Run: appletviewer AppBca.html



❖ Database servers:

- It is a computer system that allows other systems to access and retrieve data from a database.
- These servers respond to several requests to the clients and run database applications.
- Databases can require extraordinary amounts of disk space and can be accessed by multiple clients at any given time.
- It is also used by many companies for storage purposes. It allows users to access the data with the help of running a query by using a query language specific to the database.
- For example, SQL is a structured query language, which allows executing a query to access the data. The most common types of database server software include DB2, Oracle, Microsoft SQL, and Informix.

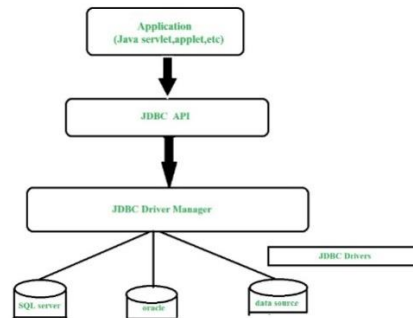
❖ JDBC:

- JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database.
- It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. We can use JDBC API to access tabular data stored in any relational database.
- By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.
- Before JDBC, ODBC API was the database API to connect and execute the query with the database.
- But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API)

that uses JDBC drivers (written in Java language). We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

Architecture of JDBC:



Description:

1. **Application:** It is a java applet or a servlet that communicates with a data source.
2. **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:
3. **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

JDBC Drivers:

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

❖ Working with Oracle Database:

To connect java application with the oracle database, we need to follow 5 following steps. In this example, we are using Oracle 10g as the database. So we need to know following information for the oracle database:

1. Driver class:

The driver class for the oracle database is `oracle.jdbc.driver.OracleDriver`.

2. Connection URL:

The connection URL for the oracle10G database is `jdbc:oracle:thin:@localhost:1521:xe` where `jdbc` is the API, `oracle` is the database, `thin` is the driver, `localhost` is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the `tnsnames.ora` file.

3. Username:

The default username for the oracle database is `system`.

4. Password:

It is the password given by the user at the time of installing the oracle database.

Create a Table:

Before establishing connection, let's first create a table in oracle database. Following is the SQL query to create a table.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
import java.sql.*;
class OracleCon
{
    public static void main(String args[])
    {
        try
        {
            //step1 load the driver class
            Class.forName("oracle.jdbc.driver.OracleDriver");
            //step2 create the connection object
            Connection con=DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
            //step3 create the statement object
            Statement stmt=con.createStatement();
            //step4 execute query
            ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
                System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```
//step5 close the connection object
con.close();
}
catch(Exception e)
{
    System.out.println(e);
}
}
}
```

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

Two ways to load the jar file:

1. paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2. set classpath:

There are two ways to set the classpath:

o temporary:

Firstly, search the ojdbc14.jar file then open command prompt and write:

```
C:>set classpath=c:\folder\ojdbc14.jar;.
```

o permanent:

Go to environment variable then click on new tab. In variable name write classpath and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.

```
C:\oracle\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.
```

❖ Working with MySQL Database:

To connect Java application with the MySQL database, we need to follow 5 following steps.

In this example we are using MySQL as the database. So we need to know following informations for the mysql database:

1. Driver class:

The driver class for the mysql database is **com.mysql.jdbc.Driver**.

2. Connection URL:

The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.

3. Username:

The default username for the mysql database is **root**.

4. Password:

It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
create database ramu;
use ramu;
create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

In this example, ramu is the database name, root is the username and password both.

```
import java.sql.*;
class MysqlCon
{
    public static void main(String args[])
    {
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            Connection con=DriverManager.getConnection( "jdbc:mysql://localhost:3306/ramu","root","root");
            //here ramu is database name, root is username and password
            Statement stmt=con.createStatement();
            ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
            System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
            con.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

The above example will fetch all the records of emp table.

To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.

Two ways to load the jar file:**1. Paste the mysqlconnector.jar file in jre/lib/ext folder:**

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2. Set classpath:

There are two ways to set the classpath:

- **temporary:**

open command prompt and write:

```
C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;;
```

- **permanent:**

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;; as C:\folder\mysql-connector-java-5.0.8-bin.jar;;

❖ Stages in a JDBC Program:

To implement the JDBC code in Java program, typically we have 6 different steps.

Step 1: Import the Packages

Step 2: Load the drivers

In order to begin with, you first need to load the driver or register it before using it in the program. Registration is to be done once in your program. You can register a driver in one of two ways mentioned below as follows:

- a. **using the forName() method :**

Here we load the driver's class file into memory at the runtime. No need of using new or create objects.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- b. **Register the drivers using DriverManager :**

DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time.

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

Step 3: Establish a connection using the Connection class object:

Establish a connection using the Connection class object

After loading the driver, establish connections as shown below as follows:

```
Connection con = DriverManager.getConnection(url,user,password)
```

- **user:** Username from which your SQL command prompt can be accessed.
- **password:** password from which the SQL command prompt can be accessed.
- **con:** It is a reference to the Connection interface.
- **Url:** Uniform Resource Locator which is created as shown below:

Step 4: Create a statement

Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

```
Statement st = con.createStatement();
```

Step 5:Execute the query

Inorder to execute the SQL commands on database, Statement interface provides provides three different methods:

- executeUpdate()
 - executeQuery()
 - execute()
- When we want to run the **non-select** operations then we can choose **executeUpdate()**
- ```
int count =stmt.executeUpdate("non-select command");
```
- When we want to execute **select** operations then we can choose **executeQuery()**
- ```
ResultSetrs=stmt.executeQuery("select command");
```
- When we want to run both select and non-select operations, then we can use **execute()**
- ```
booleanisTrue=stmt.executeQuery("select / non-select command");
```

**Step 6: Close the connections**

finally we need to close the connection object. It is very important step to close the connection. Else we may get JDBCConnectionException exception.

```
con.close();
```

**Example:**

```
packagecom.example.jdbc;
importjava.sql.Connection;
importjava.sql.DriverManager;
importjava.sql.ResultSet;
importjava.sql.Statement;
publicclassJdbc_Select_Example
{
 publicstaticvoidmain(String[]args)throwsException
 {
 // Step - I
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 // Step - II
 Connection con
 =DriverManager.getConnection("jdbc:mysql://localhost:3306/jdbc","root","123456");
 // Step -III
 Statementstmt=con.createStatement();
 // Step - IV
 ResultSetrs=stmt.executeQuery("select * from student");
 // Step - V
 while(rs.next())
 {
 System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
 }
 // Step - VI
```

```
rs.close();
stmt.close();
con.close();
}
}
```

### ❖ Types of Result Sets:

- The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set.
- The java.sql.ResultSet interface represents the result set of a database query.
- A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

There are two types of result sets namely, forward only and, bidirectional.

#### 1) Forward only ResultSet:

- The ResultSet object whose cursor moves only in one direction is known as forward only ResultSet. By default, JDBC result sets are forward-only result sets.
- You can move the cursor of the forward only **ResultSets** using the **next()** method of the ResultSet interface.
- It moves the pointer to the next row from the current position. This method returns a boolean value.
- If there are no rows next to its current position it returns false, else it returns true. Therefore, using this method in the while loop you can iterate the contents of the ResultSet object.

#### Syntax:

```
while(rs.next())
{
}
```

#### Example:

Assume we have a table named dataset with content as shown below:

| Mobile brand   | Unit sale |
|----------------|-----------|
| <b>Iphone</b>  | 3000      |
| <b>Samsung</b> | 4000      |



|              |      |
|--------------|------|
| <b>Nokia</b> | 5000 |
| <b>Vivo</b>  | 1500 |

The following example retrieves all the records of the **Dataset** table and prints the results:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class RetrievingData
{
 public static void main(String args[]) throws Exception
 {
 //Registering the Driver
 DriverManager.registerDriver(new com.mysql.jdbc.Driver());
 //Getting the connection
 String mysqlUrl = "jdbc:mysql://localhost/TestDB";
 Connection con = DriverManager.getConnection(mysqlUrl, "root", "password");
 System.out.println("Connection established.....");
 //Creating a Statement object
 Statement stmt = con.createStatement();
 //Retrieving the data
 ResultSets = stmt.executeQuery("select * from Dataset");
 System.out.println("Contents of the table");
 while(rs.next()) {
 System.out.print("Brand: "+rs.getString("Mobile_Brand")+", ");
 System.out.print("Sale: "+rs.getString("Unit_Sale"));
 System.out.println("");
 }
 }
}
```

**Output:**

Connection established.....

Contents of the table

Brand: Iphone, Sale: 3000

Brand: Samsung, Sale: 4000

Brand: Nokia, Sale: 5000

Brand: Vivo, Sale: 1500

**2) Bidirectional ResultSet:**

- A bi-directional ResultSet object is the one whose cursor moves in both forward and backward directions.
- The createStatement() method of the Connection interface has a variant which accepts two integer values representing the result set type and the concurrency type.

Statement createStatement(int resultSetType, int resultSetConcurrency)

- To create a bi-directional result set you need to pass the type as ResultSet.TYPE\_SCROLL\_SENSITIVE or ResultSet.TYPE\_SCROLL\_INSENSITIVE, along with the concurrency, to this method as:

//Creating a Statement object

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

### Example

Following example demonstrates the creation of bi-directional ResultSet. Here we trying to created a bi-directional ResultSet object which retrieves the data from the table name dataset and, we are trying to print rows of the dataset table from last to first using the **previous()** method.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class BidirectionalResultSet
{
 public static void main(String args[]) throws Exception
 {
 //Registering the Driver
 DriverManager.registerDriver(new com.mysql.jdbc.Driver());
 //Getting the connection
 String mysqlUrl = "jdbc:mysql://localhost/TestDB";
 Connection con = DriverManager.getConnection(mysqlUrl, "root", "password");
 System.out.println("Connection established.....");
 //Creating a Statement object
 Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
 //Retrieving the data
```

```
ResultSetrs = stmt.executeQuery("select * from Dataset");
rs.afterLast();
System.out.println("Contents of the table");
while(rs.previous()) {
 System.out.print("Brand: "+rs.getString("Mobile_Brand")+", ");
 System.out.print("Sale: "+rs.getString("Unit_Sale"));
 System.out.println("");
}
}
}
```

**Output:**

Connection established.....  
Contents of the table  
Brand: Vivo, Sale: 1500  
Brand: Nokia, Sale: 5000  
Brand: Samsung, Sale: 4000  
Brand: IPHONE, Sale: 3000

