Introduction to Java

- **JAVA** was developed by James Gosling at **Sun Microsystems** Inc in the year **1995**, later acquired by Oracle Corporation.
- It is a class-based, object-oriented programming language

History

- It is a programming language created in 1991.
- James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the **Green team** initiated the Java language in 1991.
- In 1995 Java was developed by **James Gosling**, who is known as the Father of Java.
- **Sun Microsystems** released its first public implementation in 1996 as **Java 1.0.**

Java programming language is named JAVA. Why?

After the name OAK, the team decided to give a new name to it and the suggested words were Silk, Jolt, revolutionary, DNA, dynamic, etc. These all names were easy to spell and fun to say, but they all wanted the name to reflect the essence of technology.

In accordance with James Gosling, **Java** the among the top names along with **Silk**, and since java was a unique name so most of them preferred it.

Java is the name of an **island** in Indonesia where the first coffee(named java coffee) was produced. And this name was chosen by James Gosling while having coffee near his office. Note that Java is just a name, not an acronym.

Java Terminology

Before learning Java, one must be familiar with these common terms of Java.

1. **Java Virtual Machine(JVM):** This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.
    - Writing a program is done by a java programmer like you and me.
    - The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
    - In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.

Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the

execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

2. Bytecode in the Development process: As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as **.class** file by the compiler.

3. Java Development Kit(JDK): While we were using the term JDK when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

4. Java Runtime Environment (JRE): JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet supports, and plugins. For running the java program, a computer needs JRE.

Primary/Main Features of Java

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of collection of objects is a way of object-oriented programming, each of which represents an instance of the class. The four main concepts of Object-Oriented programming are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, Explicit memory allocation.

4. Robust: Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so.

6. Distributed: We can create distributed applications using the java programming language. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

10. Dynamic flexibility: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes and even create new classes through sub-classes.

11. Write Once Run Anywhere: As discussed above java application generates a '.class' file which corresponds to our applications(program) but contains code in binary format. It provides ease to architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

12. Power of compilation and interpretation: Most languages are designed with purpose either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

Differences between Java and C++

Parameters	Java	C++
Founder	Java was developed by James Gosling at Sun Microsystems.	C++ was developed by Bjarne Stroustrup at Bell Labs in 1979 as an extension of the C language.
First Release	On May 23, 1995	In October 1985
Stable Release	Java SE 14 or JDK 14 was released on March 17, 2020.	C++17 was released in December 2017.
Official Website	oracle.com/java	isocpp.org
Influenced By:	Java was Influenced by Ada 83, Pascal, C++, C#, etc. languages.	C++ was Influenced by Influenced by Ada, ALGOL 68, C, ML, Simula, Smalltalk, etc. languages.
Influenced to:	Java was influenced to develop BeanShell, C#, Clojure, Groovy, Hack, J#, Kotlin, PHP, Python, Scala, etc. languages.	C++ was influenced to develop C99, Java, JS++, Lua, Perl, PHP, Python, Rust, Seed7, etc. languages.
Platform Dependency	Platform independent, Java bytecode works on any operating system.	Platform dependent, should be compiled for different platforms.
Portability	It can run in any OS hence it is portable.	C++ is platform-dependent. Hence it is not portable.
Compilation	Java is both Compiled and Interpreted Language.	C++ is only Compiled Language.
Memory Management	Memory Management is System Controlled.	Memory Management in C++ is Manual.
Virtual Keyword	It doesn't have Virtual Keyword.	It has Virtual Keyword.
Multiple Inheritance	It supports only single inheritance. Multiple inheritances are achieved partially using interfaces.	It supports both single and multiple Inheritance.

Overloading	It supports only method overloading and doesn't allow operator overloading.	It supports both method and operator overloading.
Pointers	It has limited support for pointers.	It strongly supports pointers.
Libraries	It doesn't support direct native library calls but only Java Native Interfaces.	It supports direct system library calls, making it suitable for system-level programming.
Libraries	Libraries have a wide range of classes for various high-level services.	C++ libraries have comparatively low-level functionalities.
Documentation Comment	It supports documentation comments (e.g., /**.. */) for source code.	It doesn't support documentation comments for source code.
Thread Support	Java provides built-in support for multithreading.	C++ doesn't have built-in support for threads, depends on third-party threading libraries.
Type	Java is only an object-oriented programming language.	C++ is both a procedural and an object-oriented programming language.
Input-Output mechanism	Java uses the (System class): **System.in** for input and **System.out** for output.	C++ uses **cin** for input and **cout** for an output operation.
goto Keyword	Java doesn't support goto Keyword	C++ supports goto keyword.
Structures and Unions	Java doesn't support Structures and Unions.	C++ supports Structures and Unions.
Parameter Passing	Java supports only the Pass by Value technique.	C++ supports both Pass by Value and pass by reference.
Global Scope	It supports no global scope.	It supports both global scope and namespace scope.
Object Management	Automatic object management with garbage collection.	It supports manual object management using new and delete.

Java Basic Syntax

A Java program is a collection of objects, and these objects communicate through method calls to each other to work together.

Basic terminologies in Java

1. Class: The class is a blueprint (plan) of the instance of a class (object). It can be defined as a template which describes the data and behaviour associated with its instance.

- Example: Blueprint of the house is class.

2. Object: The object is an instance of a class. It is an entity which has behaviour and state.

- Example: A car is an object whose **states** are: brand, colour, number-plate.
- Behaviour: Running on the road.

3. Method: The behaviour of an object is the method.

- **Example**: The fuel indicator indicates the amount of fuel left in the car.

Example: Steps to compile and run a java program in a console

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

```
            System.out.println("Java Programming");
        }
}
```

javac First.java
java First

Note: When the class is public, the name of the file has to be the class name.

The Basic Syntax:

1. Comments in Java
There are three types of comments in Java.

i. Single line Comment
// System.out.println("Java Programming");

ii. Multi-line Comment
/*

```
System.out.println("Java Programming");

System.out.println("Alice!");
```

*/

iii. Documentation Comment. Also called a **doc comment**.
/** documentation */

2. Source File Name
The name of a source file should exactly match the public class name with the extension of **.java**. The name of the file can be a different name if it does not have any public class. Assume you have a public class **GFG**.

First.java // valid syntax
first.java // invalid syntax

3. Case Sensitivity
Java is a case-sensitive language, which means that the identifiers ***AB, Ab, aB***, and ***ab*** are different in Java.
System.out.println("Alice"); // valid syntax
system.out.println("Alice"); // invalid syntax

4. Class Names
i. The first letter of the class should be in Uppercase (lowercase is allowed, but not discouraged).

ii. If several words are used to form the name of the class, each inner word's first letter should be in Uppercase.
Underscores are allowed, but not recommended. Also allowed are numbers and currency symbols, although the latter are also discouraged because the are used for a special purpose (for inner and anonymous classes).
class MyJavaProgram // valid syntax

class 1Program // invalid syntax

```
class My1Program     // valid syntax
class $Program       // valid syntax, but discouraged
class My$Program     // valid syntax, but discouraged
class myJavaProgram  // valid syntax, but discouraged
```

5. public static void main(String [] args)
The method main() is the main entry point into a Java program; this is where the processing starts. Also allowed is the signature **public static void main(String... args)**.

6. Method Names
i. All the method names should start with a lowercase letter.
ii. If several words are used to form the name of the method, then each first letter of the inner word should be in Uppercase. Underscores are allowed, but not recommended. Also allowed are digits and currency symbols.

```
public void employeeRecords() // valid syntax
public void EmployeeRecords() // valid syntax, but discouraged
```

7. Identifiers in java
Identifiers are the names of local variables, instance and class variables, labels, but also the names for classes, packages, modules and methods.

i. All identifiers can begin with a letter, a currency symbol or an underscore (_). According to the convention, a letter should be lower case for variables.

ii. The first character of identifiers can be followed by any combination of letters, digits, currency symbols and the underscore. The underscore is not recommended for the names of variables. Constants (static final attributes and enums) should be in all Uppercase letters.

iii. Most importantly identifiers are case-sensitive.

iv. A keyword cannot be used as an identifier since it is a reserved word and has some special meaning.

Legal identifiers: MinNumber, total, ak74, hello_world, $amount, _under_value
Illegal identifiers: 74ak, -amount

8. White-spaces in Java
A line containing only white-spaces, possibly with the comment, is known as a blank line, and the Java compiler totally ignores it.

9. Access Modifiers: These modifiers control the scope of class and methods.

- **Access Modifiers:** default, public, protected, private
- **Non-access Modifiers:** final, abstract,

10. Java Keywords

Keywords or Reserved words are the words in a language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as variable names or objects.

abstract	assert	boolean	break
Byte	case	catch	char
Class	const	continue	default
Do	double	else	enum
extends	final	finally	float
For	goto	if	implements
Import	instanceof	int	interface
Long	native	new	package
private	protected	public	return
Short	static	strictfp	super
Switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Java Hello World Program

- Java is one of the most popular and widely used programming languages and platforms.
- Java is fast, reliable, and secure.
- Java is used in every nook and corner from desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet.

The process of Java programming can be simplified in three steps:

- Create the program by typing it into a text editor and saving it to a file – HelloWorld.java.
- Compile it by typing "javac HelloWorld.java" in the terminal window.
- Execute (or run) it by typing "java HelloWorld" in the terminal window.

```
// This is a simple Java program.
// FileName : "HelloWorld.java".


class HelloWorld
{
        // Your program begins with a call to main().
        // Prints "Hello, World" to the terminal window.
        public static void main(String args[])
        {
                System.out.println("Hello, World");
        }
}
```

1. Class definition

This line uses the keyword **class** to declare that a new class is being defined.
class HelloWorld

2. HelloWorld

It is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace **{** and the closing curly brace **}**.

3. main method:

In the Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

- **public**: So that JVM can execute the method from anywhere.
- **static**: The main method is to be called without an object. The modifiers public and static can be written in either order.
- **void**: The main method doesn't return anything.
- **main()**: Name configured in the JVM.
- **String[]**: The main method accepts a single argument, i.e., an array of elements of type String.

Like in C/C++, the main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

The next line of code is shown here. Notice that it occurs inside the main() method.

System.out.println("Hello, World");

This line outputs the string "Hello, World" followed by a new line on the screen. Output is accomplished by the built-in println( ) method. The **System** is a predefined class that provides access to the system, and **out** is the variable of type output stream connected to the console.

4. compiling the program

- After successfully setting up the environment, we can open a terminal in both Windows/Unix and go to the directory where the file – HelloWorld.java is present.
- Now, to compile the HelloWorld program, execute the compiler – javac, to specify the name of the **source** file on the command line, as shown:

javac HelloWorld.java

- The compiler creates a **HelloWorld.class** (in the current working directory) that contains the **bytecode version of the program.** Now, **to execute** our program, **JVM**(Java Virtual Machine) needs to be called using java, specifying the name of the **class** file on the command line, as shown:

java HelloWorld

- This will print "Hello World" to the terminal screen.

```
F:\>javac HelloWorld.java

F:\>java HelloWorld
Hello, World

F:\>
```

Variable

- A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.
- Variable is a name of memory location.
- There are three types of variables in java: local, instance and static.

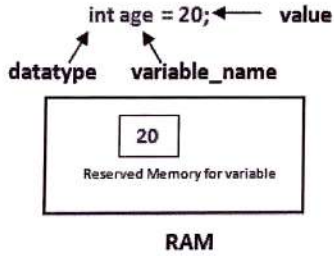
A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

```
int data=50;//Here data is variable
```

How to initialize variables?

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.



There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

```
import java.io.*;
class G {
        public static void main(String[] args)
        {
                int var = 10; // Declared a Local Variable
```

```
            // This variable is local to this main method only
            System.out.println("Local Variable: " + var);
        }
}
```

2) Instance Variable

Instance variables are non-static variables and are declared in a class outside 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.
- Instance Variable can be accessed only by creating objects.

```
import java.io.*;

class G {

  public String ge; // Declared Instance Variable

  public G()
  { // Default Constructor

    this.ge = "Shubham Jain"; // initializing Instance Variable
  }
//Main Method
  public static void main(String[] args)
  {

    // Object Creation
    G name = new G();
    // Displaying O/P
    System.out.println("name is: " + name.ge);
  }
}
```

3) Static variable

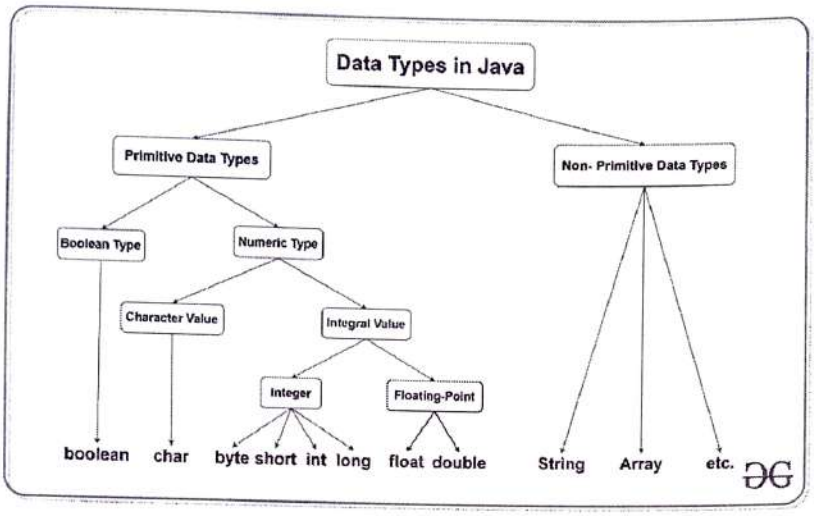
Static variables are also known as Class variables.

- These variables are declared similarly as instance variables. The difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- You can create a single copy of the static variable and share it among all the instances of the class.

```
public class A
{
static int m=100;//static variable
void method()
{
```

```
int n=90;//local variable
}
public static void main(String args[])
{
int data=50;//instance variable
}
}
//end of class
```

Data types in Java



Java has two categories of data:

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

Primitive Data Type

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\', '\n', 'β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

Non-Primitive Data Type or Reference Data Types

The **Reference Data Types** will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, objects, arrays, etc.

Literals in Java

Literal: Any constant value which can be assigned to the variable is called literal/constant.
In simple words, Literals in Java is a synthetic representation of boolean, numeric, character, or string data.

```
// Here 100 is a constant/literal.
int x = 100;
```

- **Integral literals-** For Integral data types (byte, short, int, long)
  int x = 101;
- **Floating Point Literals-** For Floating-point data types, we can specify literals in only decimal form.
  double d = 123.456;
- **Char Literals-** We can specify literal to a char data type as a single character within the single quote.
  char ch = 'a';
- **String Literals-** Any sequence of characters within double quotes is treated as String literals.
  String s = "Hello";

- **Boolean literals-** Only two values are allowed for Boolean literals, i.e., true and false.
  boolean b = true;

Operators in Java

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are:

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

1. Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

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

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

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

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

variable = value;

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

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

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

- Some of the relational operators are-
    - **==, Equal to:** returns true if the left-hand side is equal to the right-hand side.
    - **!=, Not Equal to:** returns true if the left-hand side is not equal to the right-hand side.
    - **<, less than:** returns true if the left-hand side is less than the right-hand side.

- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to:** returns true if the left-hand side is greater than or equal to the right-hand side.

5. Logical Operators: These operators are used to perform "logical AND" and "logical OR" operations.

Conditional operators are:

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

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

condition ? if true : if false

```
public class operators {
  public static void main(String[] args)
  {
    int a = 20, b = 10, c = 30, result;

    // result holds max of three
    // numbers
    result
      = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
    System.out.println("Max of three numbers = "
                + result);
  }
}
```

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

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

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

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

- >>>, **Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

9. instanceof operator: The instance of the operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. General format-
object **instance of** class/subclass/interface

Precedence and Associativity of Operators

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

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

Type conversion

- Java provides various data types just likely any other dynamic languages such as boolean, char, int, unsigned int, signed int, float, double, long, etc in total providing 7 types .
- Every datatype acquires different space while storing in memory.
- When you assign a value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion
- And if not then they need to be cast or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign a value of a smaller data type to a bigger data type.

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

```
class G {

  // Main driver method
  public static void main(String[] args)
  {
    int i = 100;

    // Automatic type conversion
    // Integer to long type
    long l = i;

    // Automatic type conversion
    // long to float type
    float f = l;

    // Print and display commands
    System.out.println("Int value " + i);
    System.out.println("Long value " + l);
    System.out.println("Float value " + f);
  }
}
```

Narrowing or Explicit Conversion

If we want to assign a value of a larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, the target type specifies the desired type to convert the specified value to.

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

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

      // Double datatype
      double d = 100.04;

     long l = (long)d;
       int i = (int)l;

      // Print statements
      System.out.println("Double value " + d);


      System.out.println("Long value " + l);

      System.out.println("Int value " + i);
   }
}
```

Program Control Statements

- **Decision Making in Java (if, if-else, switch, break, continue, jump)-**

A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

Java's Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

1. if: if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)

{

  // Statements to execute if

  // condition is true

}
```

Example:

```
class IfDemo {
   public static void main(String args[])
   {
      int i = 10;

      if (i > 15)
         System.out.println("10 is less than 15");

   }
}
```

2. if-else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
   // Executes this block if
   // condition is true
```

```
}
else
{
  // Executes this block if
  // condition is false
}
```

Example:

```
class IfElseDemo {
  public static void main(String args[])
  {
    int i = 10;

    if (i < 15)
      System.out.println("i is smaller than 15");
    else
      System.out.println("i is greater than 15");
  }
}
```

3. nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
  // Executes when condition1 is true
  if (condition2)
  {
    // Executes when condition2 is true
  }
}
```

Example:

```
class NestedIfDemo {
  public static void main(String args[])
  {
    int i = 10;

    if (i == 10) {
      // First if statement
      if (i < 15)
        System.out.println("i is smaller than 15");

      // Nested - if statement
      // Will only be executed if statement above
```

```
        // it is true
        if (i < 12)
          System.out.println(
            "i is smaller than 12 too");
        else
          System.out.println("i is greater than 15");
      }
    }
}
```

4. if-else-if ladder: Here, a user can decide among multiple options.The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
if (condition)
  statement;
else if (condition)
  statement;
.
.
else
  statement;
```

Example:

```
class ifelseifDemo {
  public static void main(String args[])
  {
    int i = 20;

    if (i == 10)
      System.out.println("i is 10");
    else if (i == 15)
      System.out.println("i is 15");
    else if (i == 20)
      System.out.println("i is 20");
    else
      System.out.println("i is not present");
  }
}
```

5. switch-case: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax:

```
switch (expression)
{
```

```
 case value1:
  statement1;
  break;
 case value2:
  statement2;
  break;
 .
 .
 case valueN:
  statementN;
  break;
 default:
  statementDefault;
}
```

- The expression can be of type byte, short, int char, or an enumeration. Beginning with JDK7, *expression* can also be of type String.
- Duplicate case values are not allowed.
- The default statement is optional.
- The break statement is used inside the switch to terminate a statement sequence.
- The break statement is optional. If omitted, execution will continue on into the next case.

Nested-Switch Statement:
Nested-Switch statements refers to Switch statements inside of another Switch Statements.
Syntax:

```
switch(n)
{
 // code to be executed if n = 1;
 case 1:

 // Nested switch
 switch(num)
 {
  // code to be executed if num = 10
  case 10:
   statement 1;
   break;

  // code to be executed if num = 20
  case 20:
   statement 2;
   break;

  // code to be executed if num = 30
```

```
 case 30:
  statement 3;
  break;

  // code to be executed if num
  // doesn't match any cases
  default:
}


break;

// code to be executed if n = 2;
case 2:
 statement 2;
 break;

// code to be executed if n = 3;
case 3:
 statement 3;
 break;

 // code to be executed if n doesn't match any cases
 default:
}
```

6. jump: Java supports three jump statements: **break, continue** and **return.** These three statements transfer control to another part of the program.

- **Break:** In Java, a break is majorly used for:
  - Terminate a sequence in a switch statement (discussed above).
  - To exit a loop.
  - Used as a "civilized" form of goto.
- **Continue:** Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

```
class ContinueDemo {
  public static void main(String args[])
  {
    for (int i = 0; i < 10; i++) {
      // If the number is even
      // skip and continue
      if (i % 2 == 0)
        continue;

      // If number is odd, print it
      System.out.print(i + " ");
    }
  }}
```

- **Return**-The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Example:

```
class Return {
  public static void main(String args[])
  {
    boolean t = true;
    System.out.println("Before the return.");

    if (t)
      return;

    // Compiler will bypass every statement
    // after return
    System.out.println("This won't execute.");
  }
}
```

- **Loops in Java-**

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

1. **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

   Syntax :

   ```
   while (boolean condition)
   {
    loop statements...
   }
   ```

- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

```
class whileLoopDemo
{
  public static void main(String args[])
  {
    int x = 1;

    // Exit when x becomes greater than 4
    while (x <= 4)
```

```
    {
      System.out.println("Value of x:" + x);

      // Increment the value of x for
      // next iteration
      x++;
    }
  }
}
```

2. for loop: for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.
Syntax:

```
for (initialization condition; testing condition;
                increment/decrement)
{
  statement(s)
}
```

- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:**When the condition becomes false, the loop terminates marking the end of its life cycle.

```
class forLoopDemo
{
  public static void main(String args[])
  {
    // for loop begins when x=2
    // and runs till x <=4
    for (int x = 2; x <= 4; x++)
      System.out.println("Value of x:" + x);
  }
}
```

Enhanced For loop
Java also includes another version of for loop introduced in Java 5. Enhanced for loop provides a simpler way to iterate through the elements of a collection or array. It is inflexible and should be

used only when there is a need to iterate through the elements in a sequential manner without knowing the index of the currently processed element.

Syntax:

```
for (T element:Collection obj/array)
{
  statement(s)
}
```

Example-

```
public class enhancedforloop
{
  public static void main(String args[])
  {
    String array[] = {"Ron", "Harry", "Hermoine"};

    //enhanced for loop
    for (String x:array)
    {
      System.out.println(x);
    }

    /* for loop for same function
    for (int i = 0; i < array.length; i++)
    {
      System.out.println(array[i]);
    }
    */
  }
}
```

3. do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

Syntax:

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

- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

```
class dowhileloopDemo
{
  public static void main(String args[])
  {
    int x = 21;
    do
    {
      // The line will be printed even
      // if the condition is false
      System.out.println("Value of x:" + x);
      x++;
    }
    while (x < 20);
  }
}
```

Nested loop means a loop statement inside another loop statement. That is why nested loops are also called as "**loop inside loop**".

Syntax for Nested For loop:

```
for ( initialization; condition; increment ) {

  for ( initialization; condition; increment ) {

    // statement of inside loop
  }

  // statement of outer loop
}
```

```
class G{
 public static void print2D(int mat[][])
   {
     // Loop through all rows
     for (int i = 0; i < mat.length; i++) {

       // Loop through all elements of current row
       for (int j = 0; j < mat[i].length; j++)
         System.out.print(mat[i][j] + " ");

       System.out.println();
     }
   }

   public static void main(String args[]) throws IOException
   {
     int mat[][] = { { 1, 2, 3, 4 },
                     { 5, 6, 7, 8 },
```

```
                {9, 10, 11, 12 } };
        print2D(mat);
    }
}
```

Output:

1 2 3 4

5 6 7 8

9 10 11 12

Input from the keyboard

- In Java, there are many ways to read strings from input.
- The simplest one is to make use of the class Scanner, which is part of the java.util library and has been newly introduced in Java 5.0.
- Using this class, we can create an object to read input from the standard input channel System.in (typically, the keyboard), as follows:

Scanner scanner = new Scanner(System.in);

Then, we can use the nextLine() method of the Scanner class to get from standard input the next line (a sequence of characters delimited by a newline character), according to the following schema:

```
import java.util.Scanner;

public class KeyboardInput {
 public static void main (String[] args) {
  ...
  Scanner scanner = new Scanner(System.in);
  String inputString = scanner.nextLine();
  ...
  System.out.println(inputString);
  ...
 }
}
```

- import java.util.Scanner; - imports the class Scanner from the library java.util
- Scanner scanner = new Scanner(System.in); - creates a new Scanner object, that is connected to standard input (the keyboard)
- String inputString = scanner.nextLine();
- We can also read in a single word (i.e., a sequence of characters delimited by a whitespace character) by making use of the next() method of the Scanner class.
- **Java Scanner class** provides nextInt() method for reading an integer value, nextDouble() method for reading a double value, nextLong() method for reading a long value, etc.

OOPs (Object-Oriented Programming System)

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**

1. **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.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

2. Class

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

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

Syntax to declare a class:

```
1. class <class_name>{
field;
method;
}
```

3. Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

4. 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.

5. 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.

6. 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.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.


```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
    }
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one.

```
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
```

```
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
System.out.println(s1.name);
 }
}
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent2{
public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Jack";
System.out.println(s1.id+" "+s1.name);//printing members with a white space
}
}
```

Creating Objects Using New Keyword

1. Using new keyword

Using the new keyword in java is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call

```
class G {

  String name = "Java Programming";

  public static void main(String[] args)
  {

    // using new keyword
    G obj = new G();

    // Print and display the object
    System.out.println(obj.name);
  }
}
```

Methods

- A method in Java or Java Method 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.
- Methods in Java allow us to reuse the code without retyping the code.

Method Declaration

In general, method declarations has five components :

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

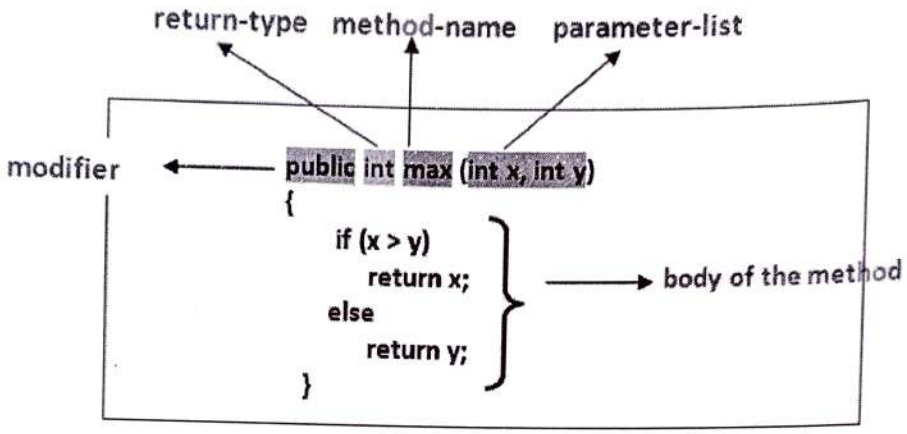
- **public:** It is accessible in all classes in your application.
- **protected:** It is accessible within the class in which it is defined and in its subclass/es
- **private:** It is accessible only within the class in which it is defined.
- **default:** It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

2. The return type: The data type of the value returned by the method or void if does not return a value.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different.

4. Parameter list: Comma-separated list of the input parameters is defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

5. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.



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.

Rules to Name a Method

- While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter.
- In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example, findSum, computeMax, setX and getX.

Method Calling

The method needs to be called for using its functionality.

Let's see an example of the predefined method:

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

How to Create a User-defined Method:

Example1:

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

Example 2:

```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
```

```
int c = add(a, b);  //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2)  //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
```

Method Overloading

- In Java, two or more methods may have the same name if they differ in parameters (different number of parameters, different types of parameters, or both).
- These methods are called overloaded methods and this feature is called method overloading.

void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }

Here, the func() method is overloaded. These methods have the same name but accept different arguments.

Why method overloading?

- Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).
- In order to accomplish the task, you can create two methods sum2num(int, int) and sum3num(int, int, int) for two and three parameters respectively. However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.
- The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

How to perform method overloading in Java?

1. Overloading by changing the number of parameters:

```
class MethodOverloading {
   private static void display(int a){
      System.out.println("Arguments: " + a);
   }

   private static void display(int a, int b){
      System.out.println("Arguments: " + a + " and " + b);
   }

   public static void main(String[] args) {
      display(1);
      display(1, 4);
   }
}
```

2. Method Overloading by changing the data type of parameters:

```
class MethodOverloading {

   // this method accepts int
   private static void display(int a){
      System.out.println("Got Integer data.");
   }

   // this method  accepts String object
   private static void display(String a){
      System.out.println("Got String object.");
   }

   public static void main(String[] args) {
      display(1);
      display("Hello");
   }
}
```

Constructors

- A constructor in Java is a **special method** that is used to initialize objects.
- The constructor is called when an object of a class is created.
- At the time of calling the constructor, memory for the object is allocated in the memory
- Every time an object is created using the new() keyword, at least one constructor is called.

How Constructors are Different from Methods in Java?

- Constructors must have the same name as the class within which it is defined while it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Types of Constructors in Java

Primarily there are two types of constructors in java:

- **No-argument constructor**
- **Parameterized Constructor**

1. No-argument constructor

A constructor that has no parameter is known as the default constructor. If we don't define a constructor in a class, then the compiler creates a **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.

```
class G {
   int num;
   String name;

   // this would be invoked while an object
   // of that class is created.
   G()
         { System.out.println("Constructor called");
}
         }

class GF {
   public static void main(String[] args)
   {
      // this would invoke default constructor.
      G g1 = new G();

      // Default constructor provides the default
      // values to the object like 0, null
      System.out.println(geek1.name);
      System.out.println(geek1.num);
   }}
```

2. Parameterized Constructor

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

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

```
    // Constructor would initialize data members
    // With the values of passed arguments while
    // Object of that class created
    G(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}

// Class 2
class GF {
    // main driver method
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        G g1 = new G("adam", 1);
        System.out.println("Name :" + g1.name
                           + " and Id :" + g1.id);
    }
}
```

Constructors Overloading

Similar to Java method overloading, we can also create two or more constructors with different parameters. This is called constructors overloading.

```
class Main {

 String language;

 // constructor with no parameter
 Main() {
  this.language = "Java";
 }

 // constructor with a single parameter
 Main(String language) {
  this.language = language;
 }

 public void getName() {
  System.out.println("Programming Langauage: " + this.language);
 }

 public static void main(String[] args) {

  // call constructor with no parameter
  Main obj1 = new Main();

  // call constructor with a single parameter
  Main obj2 = new Main("Python");
```

```
    obj1.getName();
    obj2.getName();
  }
}
```

The new operator

The new operator is used in Java to create new objects. It can also be used to create an array object.

Let us first see the steps when creating an object from a class –

- **Declaration** – A variable declaration with a variable name with an object type.
- **Instantiation** – The 'new' keyword is used to create the object.
- **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Syntax

NewExample obj=**new** NewExample();

Points to remember

- It is used to create the object.
- It allocates the memory at runtime.
- All objects occupy memory in the heap area.
- It invokes the object constructor.
- Now, let us see an example –

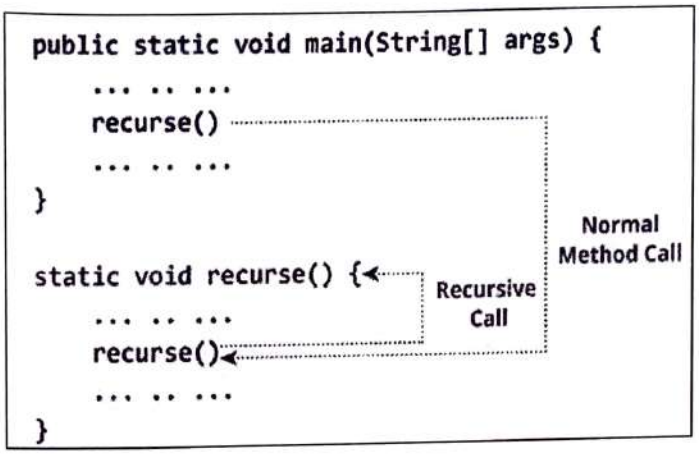
Example

```
public class NewExample1 {

  void display()
  {
    System.out.println("Invoking Method");
  }

  public static void main(String[] args) {
    NewExample1 obj=new NewExample1();
    obj.display();
  }
  }
```

Recursion

- Recursion is the technique of making a function call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

How Recursion works?

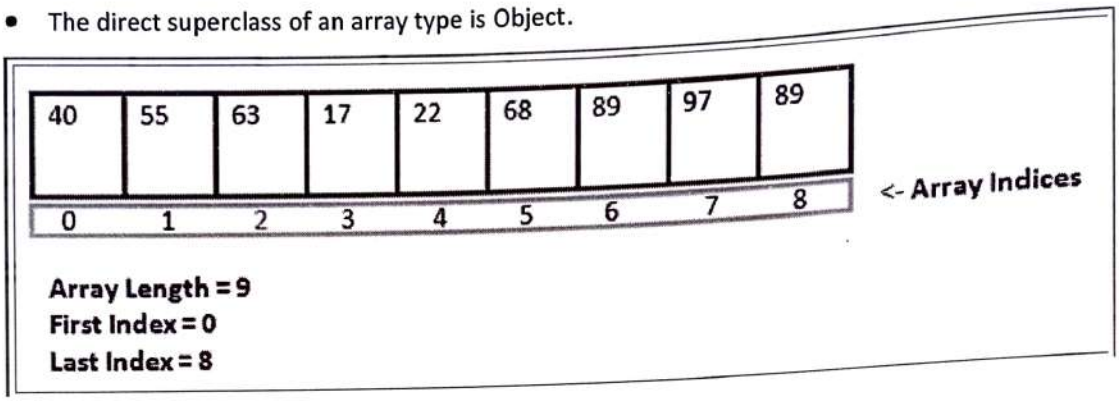


- In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.
- Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

```
class Factorial {

 static int factorial( int n ) {

    if (n != 0)  // termination condition

      return n * factorial(n-1); // recursive call

    else

      return 1; }

public static void main(String[] args) {

    int number = 4, result;

    result = factorial(number);

    System.out.println(number + " factorial = " + result);  }}
```

Arrays in Java

- An array in Java is a group of like-typed variables referred to by a common name.
- The variables in the array are ordered, and each has an index beginning from 0.
- **Length Property:-**Since arrays are objects in Java, we can find their length using the object property length.
- The direct superclass of an array type is Object.



- **Creating, Initializing, and Accessing an Array**

1. One-Dimensional Arrays:

The general form of a one-dimensional array declaration is

type var-name[];

OR

type[] var-name;

- An array declaration has two components: the type and the name.
- type declares the element type of the array. The element type determines the data type of each element that comprises the array.
- Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc.

Example:

```
// both are valid declarations
int intArray[];
or int[] intArray;

byte byteArray[];
short shortsArray[];
boolean booleanArray[];
long longArray[];
float floatArray[];
```

```
double doubleArray[];
char charArray[];
```

Instantiating an Array in Java

- When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like this:
- The general form of *new* as it applies to one-dimensional arrays appears as follows:
  var-name = new type [size];
- Here, type specifies the type of data being allocated,
- size determines the number of elements in the array,
- and var-name is the name of the array variable that is linked to the array.
- To use new to allocate an array, **you must specify the type and number of elements to allocate.**

Example:

```
int intArray[];  //declaring array
intArray = new int[20];  // allocating memory to array
OR
int[] intArray = new int[20]; // combining both statements in one
```

Accessing Java Array Elements using for Loop

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

```
// accessing the elements of the specified array
for (int i = 0; i < arr.length; i++)
 System.out.println("Element at index " + i +
                    " : "+ arr[i]);
```

Example:

```
class G
{
   public static void main (String[] args)
   {
    // declares an Array of integers.
    int[] arr;

    // allocating memory for 5 integers.
    arr = new int[5];

    // initialize the first elements of the array
    arr[0] = 10;

    // initialize the second elements of the array
    arr[1] = 20;
```

```
    //so on...
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;

    // accessing the elements of the specified array
    for (int i = 0; i < arr.length; i++)
      System.out.println("Element at index " + i +
                         " : "+ arr[i]);
  }
}
```

2. Multidimensional Arrays:

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

Syntax to Declare Multidimensional Array in Java

I. dataType[][] arrayRefVar; (or)
II. dataType [][]arrayRefVar; (or)
III. dataType arrayRefVar[][]; (or)
IV. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

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

Example of Multidimensional Java Array

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

Assigning Array Reference

As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to. You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other. For example, consider this program

```
// Assigning array reference variables.
class AssignARef {
  public static void main(String args[]) {
    int i;

    int nums1[] = new int[10];
    int nums2[] = new int[10];

    for(i=0; i < 10; i++)
      nums1[i] = i;

    for(i=0; i < 10; i++)
      nums2[i] = -i;
    System.out.print("Here is nums1: ");
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();



    System.out.print("Here is nums2: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

    nums2 = nums1; // now nums2 refers to nums1 <---- Assign an array reference.

    System.out.print("Here is nums2 after assignment: ");
    for(i=0; i < 10; i++)
      System.out.print(nums2[i] + " ");
    System.out.println();

    // now operate on nums1 array through nums2
    nums2[3] = 99;

    System.out.print("Here is nums1 after change through nums2: ");
    for(i=0; i < 10; i++)
      System.out.print(nums1[i] + " ");
    System.out.println();
  }
}
```

The output from the program is shown here:

```
Here is nums1: 0 1 2 3 4 5 6 7 8 9
Here is nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9
```

String Fundamentals and Handling

- **String Class:**

In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use **double quotes** to represent a string in Java. For example,

```
// create a string
String type = "Java programming";
```

Here, we have created a string variable named type. The variable is initialized with the string Java Programming.

Example: Create a String in Java

```
class Main {
  public static void main(String[] args) {
    // create strings
    String first = "Java";
    String second = "Python";
    String third = "JavaScript";

    // print strings
    System.out.println(first);  // print Java
    System.out.println(second); // print Python
    System.out.println(third);  // print JavaScript
  }
}
```

Java String Operations

1. Get length of a String

To find the length of a string, we use the length() method of the String. For example,

```
class Main {
 public static void main(String[] args) {
// create a string
  String greet = "Hello! World";
  System.out.println("String: " + greet);
// get the length of greet
  int length = greet.length();
  System.out.println("Length: " + length);
 }
}
```

2. Join Two Java Strings

We can join two strings in Java using the concat() method. For example,

```
class Main {
 public static void main(String[] args) {
// create first string
  String first = "Java ";
  System.out.println("First String: " + first);
// create second
  String second = "Programming";
  System.out.println("Second String: " + second);
// join two strings
  String joinedString = first.concat(second);
  System.out.println("Joined String: " + joinedString);
}
```

```
}
```

3. Compare two Strings

In Java, we can make comparisons between two strings using the equals() method. For example,

```
class Main {

 public static void main(String[] args) {

// create 3 strings

  String first = "java programming";

  String second = "java programming";

  String third = "python programming";

// compare first and second strings

   boolean result1 = first.equals(second);

   System.out.println("Strings first and second are equal: " + result1);

// compare first and third strings

   boolean result2 = first.equals(third);

   System.out.println("Strings first and third are equal: " + result2);

 }

}
```

4.Char charAt(int i): Returns the character at i^{th} index.
"JavaProgramming".charAt(3); // returns 'a'

5.String substring (int i): Return the substring from the i^{th} index character to end.
"JavaProgramming ".substring(3); // returns "aProgramming"

6. String substring (int i, int j): Returns the substring from i to j-1 index.
"GeeksforGeeks".substring(2, 5); // returns "eks"

7. int indexOf (String s): Returns the index within the string of the first occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.indexOf("Share"); // returns 6
```

8. int indexOf (String s, int i): Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";
```

```
int output = s.indexOf("ea",3);// returns 13
```

9. Int lastIndexOf(String s): Returns the index within the string of the last occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.lastIndexOf("a"); // returns 14
```

10.String toLowerCase(): Converts all the characters in the String to lower case.

```
String word1 = "HeLLo";
String word3 = word1.toLowerCase(); // returns "hello"
```

Immutable String in Java

In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

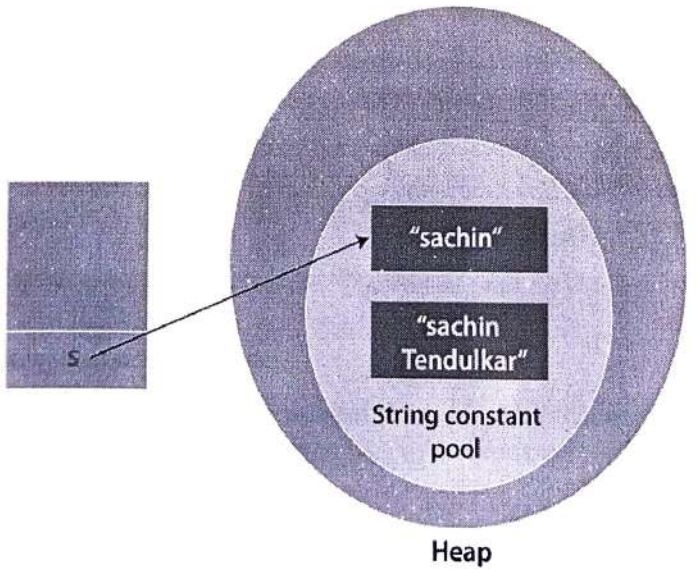
Once String object is created its data or state can't be changed but a new String object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
  String s="Sachin";
  s.concat(" Tendulkar");//concat() method appends the string at the end
  System.out.println(s);//will print Sachin because strings are immutable objects
}
}
```

Output

Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but *s* reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

```
class Testimmutablestring1{
 public static void main(String args[]){
  String s="Sachin";
  s=s.concat(" Tendulkar");
  System.out.println(s);
 }
}
```

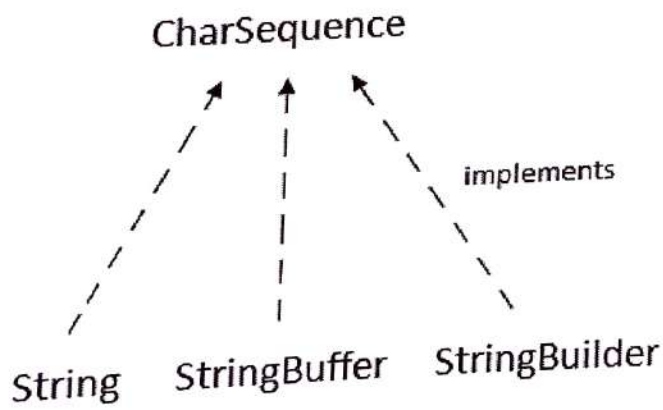
Output

Sachin Tendulkar

Three String-Related Language Features

CharSequence Interface

- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



- The Java String is immutable which means it cannot be changed.
- Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

- **StringBuffer Class:**

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

```
class StringBufferExample{
        public static void main(String args[]){
        StringBuffer sb=new StringBuffer("Hello ");
        sb.append("Java");//now original string is changed
        System.out.println(sb);//prints Hello Java
        }
}
```

Output:
Hello Java

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

3) StringBuffer replace() Method

The replace() method replaces the given String from the specified beginIndex and endIndex.

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
```

```
}
}
```

4) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

5) StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH } }
```

- **StringBuilder Class:**

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

Syntax:-

```
class StringBuilderExample{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Note: Same Methods as String Buffer.

String Constructors

- The String object can be created explicitly by using the new keyword and a constructor in the same way as you have created objects in previously. For example The statement
  String str = new String("Welcome to Java**");**

Here, String ("Welcome to Java") is actually a constructor of the form String (string literals).

- You can also create a string from an array of characters. To create a string initialised by an array of characters, use the constructor of the form
  String (charArray)
  For example, consider the following character array.
  char[] charArray ={'H','i',' ','D','I','N','E','S','H'};
- In addition to these two constructors, String class also supports the following constructors,
  • **String ()**: It constructs a new String object which is initialized to an empty string (" "). For example:
  String s = new String();
  Will create a string reference variable s that will reference an empty string.

Escape character in Java Strings

The escape character is used to escape some of the characters present inside a string.

Suppose we need to include double quotes inside a string.

```
// include double quote
String example = "This is the "String" class";
```

Since strings are represented by double quotes, the compiler will treat "This is the " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Java. For example,

```
// use the escape character
String example = "This is the \"String\" class.";
```

Creating strings using the new keyword:

Since strings in Java are objects, we can create strings using the new keyword as well. For example,

```
// create a string using the new keyword
String name = new String("Java String");
```

In the above example, we have created a string name using the new keyword.
Here, when we create a string object, the String() constructor is invoked.

```
class Main {
```

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

  // create a string using new
  String name = new String("Java String");

  System.out.println(name); // print Java String
 }
}
```

Inheritance Basics

Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.

Important terminology:

- **Super Class:** The class whose features are inherited is known as superclass(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is **extends**.

Syntax :

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

Example:

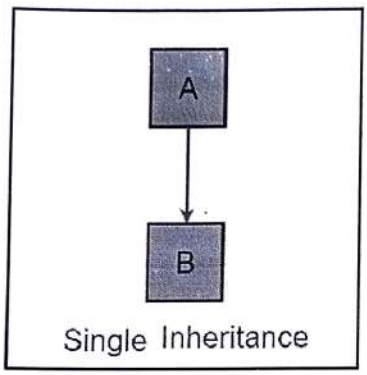
```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
 Programmer p=new Programmer();
 System.out.println("Programmer salary is:"+p.salary);
 System.out.println("Bonus of Programmer is:"+p.bonus);
```

```
}
}
```

Types of Inheritance in Java

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

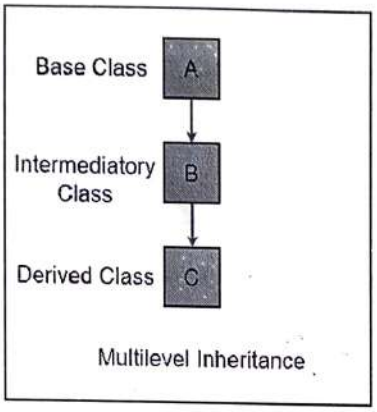
1. Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Example:

```
class one {
   public void print_ge()
   {
      System.out.println("Java");
   }
}

class two extends one {
   public void print_for() { System.out.println("Is"); }
}
// Driver class
public class Main {
   public static void main(String[] args)
   {
      two g = new two();
      g.print_ge();
      g.print_for();
      g.print_ge();
   }
}
```

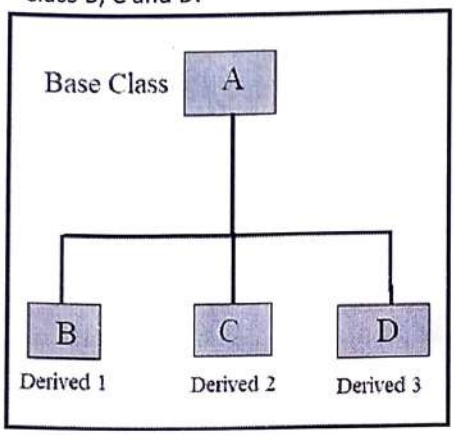
2. Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



Example:

```
class one {
   public void print_ge()
   {
      System.out.println("Java");
   }
}

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

class three extends two {
   public void print_ge()
   {
      System.out.println(“Java");
   }
}

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

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 class B, C and D.

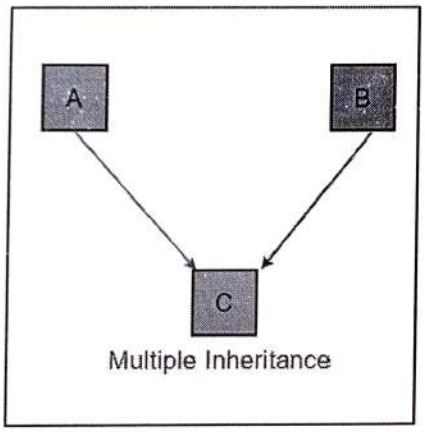


Example:

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

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

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

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

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

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

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

```
    }
}
```

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.



Member Access(Access Modifiers) In Inheritance

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.

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.

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 modifer.

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

Inheritance and Constructors in Java

- It is very important to understand how the constructors get executed in the inheritance concept. In the inheritance, the constructors never get inherited to any child class.
- In java, the default constructor of a parent class called automatically by the constructor of its child class. That means when we create an object of the child class, the parent class constructor executed, followed by the child class constructor executed.

Let's look at the following example java code.

```
class ParentClass{

        int a;

        ParentClass(){

                System.out.println("Inside ParentClass constructor!");
```

```
        }
    }
    class ChildClass extends ParentClass{
        ChildClass(){
            System.out.println("Inside ChildClass constructor!!");
        }
    }
    class ChildChildClass extends ChildClass{
        ChildChildClass(){
            System.out.println("Inside ChildChildClass constructor!!");
        }
    }
    public class ConstructorInInheritance {
        public static void main(String[] args) {
            ChildChildClass obj = new ChildChildClass();}}
```

Output:

```
Inside ParentClass constructor!
Inside ChildClass constructor!!
Inside ChildChildClass constructor!!
```

- However, if the parent class contains both default and parameterized constructor, then only the default constructor called automatically by the child class constructor.

Let's look at the following example java code.

```
class ParentClass{
    int a;
    ParentClass(int a){
        System.out.println("Inside ParentClass parameterized constructor!");
        this.a = a;
```

```
        }
        ParentClass(){
                System.out.println("Inside ParentClass default constructor!");
        }
}
class ChildClass extends ParentClass{
        ChildClass(){
                System.out.println("Inside ChildClass constructor!!");
        }
}
public class ConstructorInInheritance {
        public static void main(String[] args) {
                ChildClass obj = new ChildClass();}}
```

Output:

```
Inside ParentClass default constructor!
Inside ChildClass constructor!!
```

Constructor in inheritance

- A constructor is a method with the same name as the class name and is invoked automatically when a new instance of a class is created.
- Constructors of both classes must be executed when the object of child class is created.
- Sub Class's constructor invokes constructor of super class.
- Explicit call to the super class constructor from sub class's can be made using super().
- Super() should be the first statement of child class constructor.
- if u don't write super() explicitly then java compiler implicitly write the super().

Super Keyword

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables: This scenario occurs when a derived class and base class has same data members.

```
class Vehicle
{
   int maxSpeed = 120;
}


/* sub class Car extending vehicle */
class Car extends Vehicle
{
   int maxSpeed = 180;


   void display()
   {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}


/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

Output:

Maximum Speed: 120

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}
/* Subclass Student */
class Student extends Person
{
        void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
```

```
}

/* Driver program to test */

class Test

{

        public static void main(String args[])

        {

                Student s = new Student();

                // calling display() of Student

                s.display();

        }

}
```

Output:

```
This is student class
This is person class
```

3. Use of super with constructors: super keyword can also be used to access the parent class constructor. One more important thing is that, "super' can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept

```
class Person
{
 Person()
   {
     System.out.println("Person class Constructor");
   }
}



/* subclass Student extending the Person class */
class Student extends Person
{
  Student()
  {
    // invoke or call parent class constructor
    super();


    System.out.println("Student class Constructor");
```

```
  }
}
 /* Driver program to test*/
class Test
{
   public static void main(String[] args)
  {
  Student s = new Student();
   }
}
```

Output:

Person class Constructor
Student class Constructor

Other Important points:

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

Abstraction in Java

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.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.

Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
 abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Output:

running safely

Interface in Java

- 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.

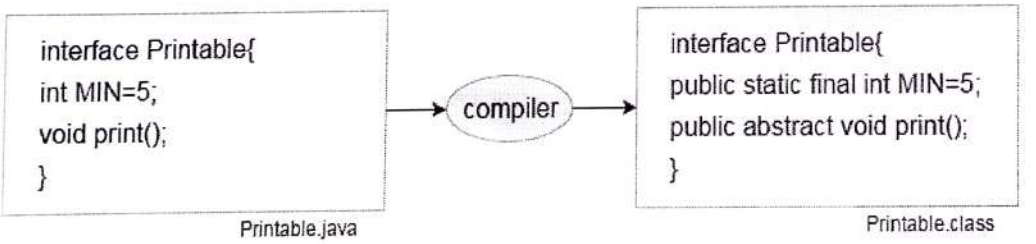
How to declare an interface?

- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

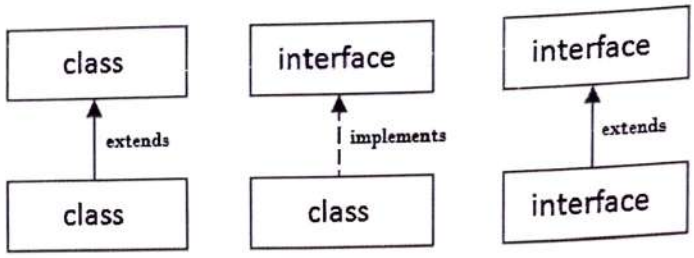
```
interface <interface_name>{

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

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



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 1:

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

Hello

Java Interface Example 2:

```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
```

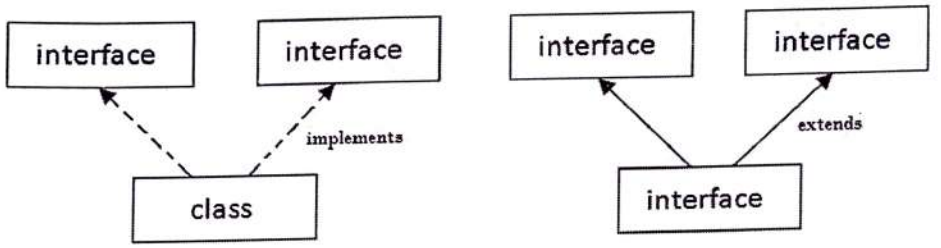
```
d.draw();
}}
```

Output:

drawing circle

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

Can we cast an object reference to an interface reference in java?

Yes, you can.

If you implement an interface and provide body to its methods from a class. You can hold object of that class using the reference variable of the interface i.e. cast an object reference to an interface reference.

But, using this you can access the methods of the interface only, if you try to access the methods of the class a compile time error is generated.

Example

In the main method we are assigning the object of the class to the reference variable of the interface and trying to invoke both the method.

```
interface MyInterface{
  public static int num = 100;
  public void display();
}
public class InterfaceExample implements MyInterface{
  public void display() {
    System.out.println("This is the implementation of the display method");
  }
  public void show() {
    System.out.println("This is the implementation of the show method");
  }
  public static void main(String args[]) {
    MyInterface obj = new InterfaceExample();
    obj.display();
    obj.show();
  }
}
```

Compile time error

On compiling, the above program generates the following compile time error –

```
InterfaceExample.java:16: error: cannot find symbol
  obj.show();
    ^
symbol: method show()
location: variable obj of type MyInterface
1 error
```

To make this program work you need to remove the line calling the method of the class as –

Example

```
interface MyInterface{
  public static int num = 100;
  public void display();
}
public class InterfaceExample implements MyInterface{
  public void display() {
    System.out.println("This is the implementation of the display method");
  }
  public void show() {
    System.out.println("This is the implementation of the show method");
  }
  public static void main(String args[]) {
    MyInterface obj = new InterfaceExample();
    obj.display();
    //obj.show();
  }
}
```

Now, the program gets compiled and executed successfully.

Output:
This is the implementation of the display method

Therefore, you need to cast an object reference to an interface reference. Whenever you need to call the methods of the interface only.

Difference between Abstract Class and Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Following are the important differences between Class and an Interface.

Sr. No.	Key	Class	Interface
1	Supported Methods	A class can have both an abstract as well as concrete methods.	Interface can have only abstract methods. Java 8 onwards, it can have default as well as static methods.
2	Multiple Inheritance	Multiple Inheritance is not supported.	Interface supports Multiple Inheritance.
3	Supported Variables	final, non-final, static and non-static variables supported.	Only static and final variables are permitted.
4	Implementation	A class can implement an interface.	Interface can not implement an interface, it can extend an interface.
5	Keyword	A class is declared using class keyword.	Interface is declared using interface keyword.
6	Inheritance	A class can inherit another class using extends keyword and implement an interface.	Interface can inherit only an inteface.
7	Inheritance	A class can be inherited using extends keyword.	Interface can only be implemented using implements keyword.
8	Access	A class can have any type of members like private, public.	Interface can only have public members.
9	Constructor	A class can have constructor methods.	Interface can not have a constructor.

Java Packages

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- **Built-in Packages** (packages from the Java API)
- **User-defined Packages** (create your own packages)

1. Built-in Packages

- The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the **import** keyword:

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the **Scanner** class, **which is used to get user input.**

Example

```
import java.util.Scanner;
```

In the example above, **java.util** is a package, while **Scanner** is a class of the **java.util** package.

To use the **Scanner** class, create an object of the class and use any of the available methods found in the **Scanner** class documentation. In our example, we will use the **nextLine()** method, which is used to read a complete line

Import a Package

There are many packages to choose from. In the previous example, we used the **Scanner** class from the **java.util** package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the **java.util** package:

Example

```
import java.util.*;
```

2. User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

To create a package, use the **package** keyword

MyPackageClass.java

```
package mypack;

class MyPackageClass {

  public static void main(String[] args) {

    System.out.println("This is my package!");

  }

}
```

- Save the file as **MyPackageClass.java**, and compile it:
- **C:\Users\Your Name>javac MyPackageClass.java**
- **Then compile the package:**
- **C:\Users\Your Name>javac -d . MyPackageClass.java**
- **This forces the compiler to create the "mypack" package.**

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

- When we compiled the package in the example above, a new folder was created, called "mypack".

- To run the **MyPackageClass.java** file, write the following:

- C:\Users\Your Name>java mypack.MyPackageClass
- The output will be:
  This is my package!

Packages and Member Access (Access Modifiers)

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

Static import

- In Java, static import concept is introduced in 1.5 version.
- With the help of static import, we can access the static members of a class directly without class name or any object.
- For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly. According to SUN microSystem, it will improve the code readability and enhance coding.
- But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should **not** go for static import.

```
// Java Program to illustrate
// calling of predefined methods
// without static import
class Geeks {
   public static void main(String[] args)
   {
      System.out.println(Math.sqrt(4));
      System.out.println(Math.pow(2, 2));
      System.out.println(Math.abs(6.3));
   }
}
```

```
// Java Program to illustrate
// calling of predefined methods
// with static import
import static java.lang.Math.*;
class Test2 {
```

```
    public static void main(String[] args)
    {
      System.out.println(sqrt(4));
      System.out.println(pow(2, 2));
      System.out.println(abs(6.3));
    }
}
```

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.
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- 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.

An exception can occur for many reasons. Some of them are:

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

What is an Error?

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

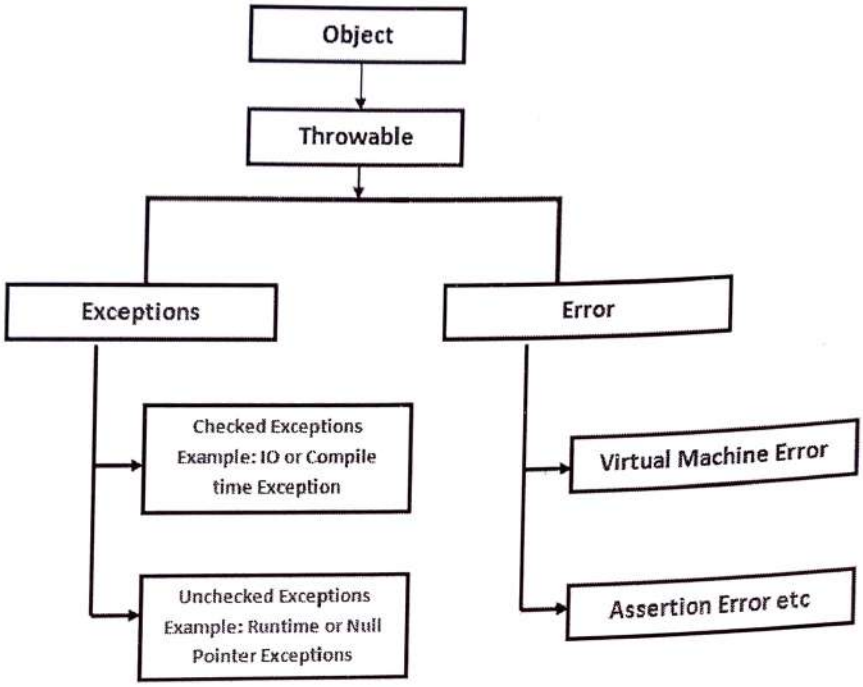
Error vs Exception

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

Exception Hierarchy

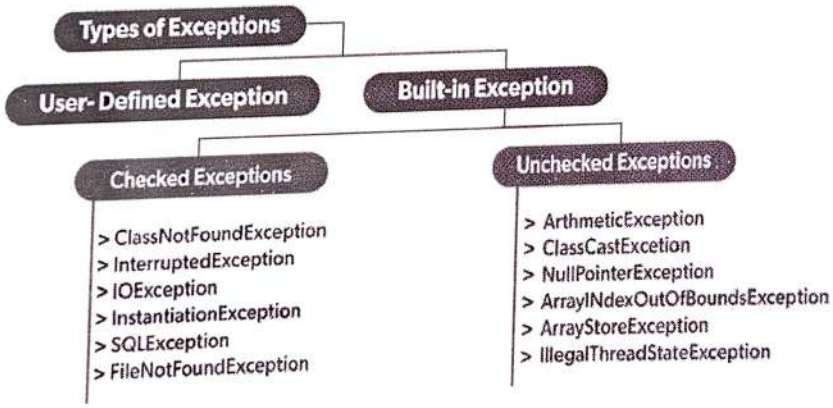
All exception and errors types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another

branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Types of Exceptions

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



Exceptions can be Categorized into 2 Ways:

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

- UncheckedException

2. User-Defined Exceptions

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

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

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

The advantages of Exception Handling in Java are as follows:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
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.

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

```
public class JavaExceptionExample{
public static void main(String args[]){
 try{
   //code that may raise exception
   int data=100/0;
 }catch(ArithmeticException e){System.out.println(e);}
 //rest code of the program
 System.out.println("rest of the code...");
        }
    }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Multiple-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Let's see a simple example of java multi-catch block.

```
public class MultipleCatchBlock1 {

  public static void main(String[] args) {

      try{
         int a[]=new int[5];
         a[5]=30/0;
        }
        catch(ArithmeticException e)
          {
           System.out.println("Arithmetic Exception occurs");
          }
        catch(ArrayIndexOutOfBoundsException e)
          {
           System.out.println("ArrayIndexOutOfBounds Exception occurs");
          }
        catch(Exception e)
          {
           System.out.println("Parent Exception occurs");
          }
        System.out.println("rest of the code");
  }
}
```

Output:
Arithmetic Exception occurs
rest of the code

Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }

   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Java Nested try Example:

```
public class NestedTryBlock{
 public static void main(String args[]){
//outer try block
 try{
 //inner try block 1
  try{
   System.out.println("going to divide by 0");
   int b =39/0;
  }
```

```
    //catch block of inner try block 1
    catch(ArithmeticException e)
    {
     System.out.println(e);
    }


    //inner try block 2
    try{
    int a[]=new int[5];

    //assigning the value out of array bounds
     a[5]=4;
     }

    //catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
      System.out.println(e);
    }


    System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  {
   System.out.println("handled the exception (outer catch)");
  }

  System.out.println("normal flow..");
 }
}
```

Java finally block

- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.

```
class TestFinallyBlock {

 public static void main(String args[]){

 try{

//below code do not throw any exception
```

```
  int data=25/5;
  System.out.println(data);
 }
//catch won't be executed
 catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of the code...");
 }
}
```

Java throw keyword

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

```
throw Instance i.e.,
throw new exception_class("error message");
throw new IOException("sorry device error");
```

Example:

```
public class TestThrow1 {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age<18) {
      //throw Arithmetic exception if not eligible to vote
```

```
        throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
        System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
   }
}
```

Subclass Exceptions

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

1. If the superclass method does not declare an exception:

```
import java.io.*;
class Parent{

 // defining the method
 void msg() {
   System.out.println("parent method");
   }
}

public class TestExceptionChild extends Parent{

 // overriding the method in child class
 // gives compile time error
 void msg() throws IOException {
   System.out.println("TestExceptionChild");
 }

 public static void main(String args[]) {
  Parent p = new TestExceptionChild();
```

```
    p.msg();
   }
  }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild.java
TestExceptionChild.java:14: error: msg() in TestExceptionChild cannot override msg(
) in Parent
  void msg() throws IOException {
       ^
  overridden method does not throw IOException
1 error
```

2. If the superclass method declares an exception:

```
import java.io.*;
class Parent{
 void msg()throws ArithmeticException {
  System.out.println("parent method");
 }
}

public class TestExceptionChild2 extends Parent{
 void msg()throws Exception {
  System.out.println("child method");
 }

 public static void main(String args[]) {
  Parent p = new TestExceptionChild2();

 try {
 p.msg();
 }
 catch (Exception e){}

 }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild2.java
TestExceptionChild2.java:9: error: msg() in TestExceptionChild2 cannot override msg
() in Parent
  void msg()throws Exception {
       ^
  overridden method does not throw Exception
1 error
```

Multithreading

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- Each part of such program is called a thread.
- So, threads are light-weight processes within a process.
- Java Multithreading is mostly used in games, animation, etc.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

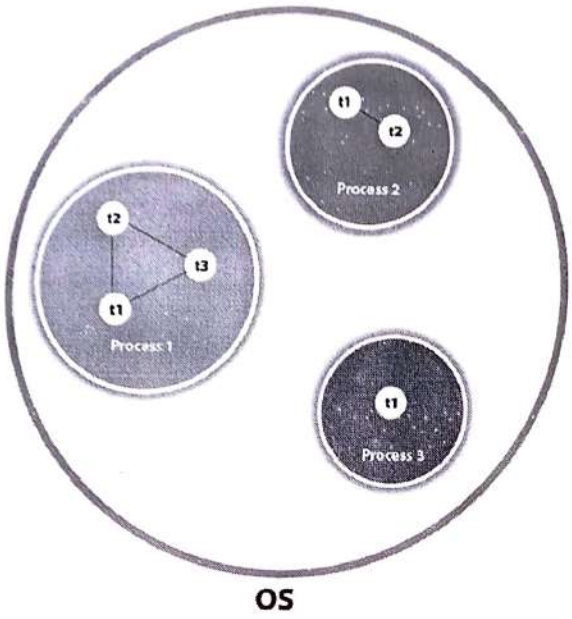
2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



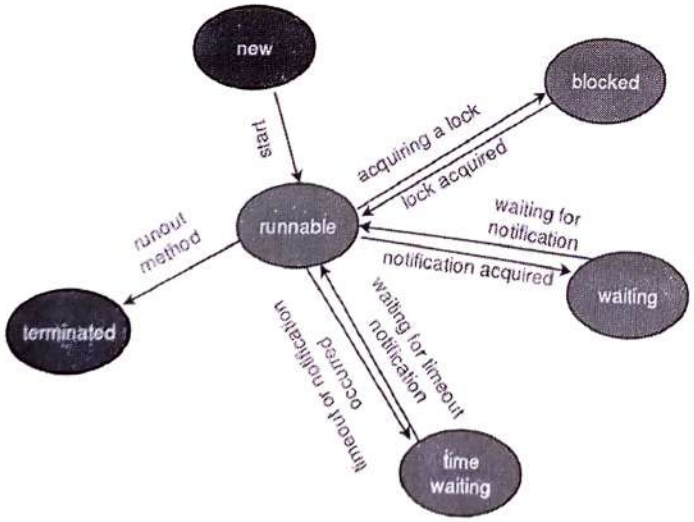
As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Lifecycle of a Thread in Java

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
4. Waiting
5. Timed Waiting
6. Terminated

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



Life Cycle of a thread

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.

Threads can be created by using two mechanisms :

1. **Extending the Thread class**
2. **Implementing the Runnable Interface**

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class.

- This class overrides the run() method available in the Thread class.
- A thread begins its life inside run() method.
- We create an object of our new class and call start() method to start the execution of a thread.
- Start() invokes the run() method on the Thread object.

Example:

```
class MultithreadingDemo extends Thread {
  public void run()
  {
    try {
```

```
            // Displaying the thread that is running
            System.out.println(
              "Thread " + Thread.currentThread().getId()
              + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
              = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Output

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method.

Then we instantiate a Thread object and call start() method on this object.

```
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
              "Thread " + Thread.currentThread().getId()
              + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
```

```
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Output

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Creating Multiple Threads in Java

Now, we will learn methods of creating multiple **threads in Java program.** Basically, when we need to perform several tasks at a time, we can create multiple threads to perform multiple tasks in a program. For example, to perform two tasks, we can create two threads and attach them to two tasks. Hence, creating multiple threads in Java programming helps to perform more than one task simultaneously.

In multiple threading programming, multiple threads are executing simultaneously that improves the performance of CPU because CPU is not idle if other threads are waiting to get some resources.

Multiple threads share the same address space in the heap memory. Therefore, It is good to create multiple threads to execute multiple tasks rather than creating multiple processes. Look at the below picture.

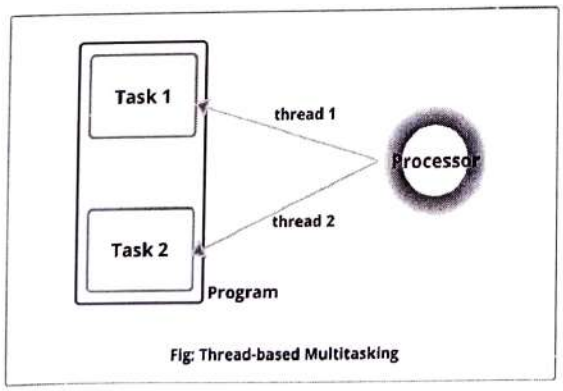


Fig: Thread-based Multitasking

Suppose there is one person (1 thread) to perform these two tasks. In this case, he has to first cut ticket and then come along with us to show seats. Then, he will go back to door to cut second ticket and then again enter the hall to show seat for second ticket.

Like this, he is taking a lot of time to perform these tasks one by one. If theater manager employs two persons (2 threads) to perform these two tasks, one person will cut the ticket and another person will show seat.

Thus, when the second person will be showing seat, the first person will cut the ticket. Like this, both persons will act simultaneously and wastage of time will not happen.

Let's create a program where we will try to implement this realtime scenario. Look at the following source code.

```
// Two threads performing two tasks at a time.
public class MyThread extends Thread
{
// Declare a String variable to represent task.
 String task;

MyThread(String task)
{
 this.task = task;
}
public void run()
{
for(int i = 1; i <= 5; i++)
{
 System.out.println(task+ " : " +i);
 try
 {
  Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.
 }
 catch(InterruptedException ie) {
  System.out.println(ie.getMessage());
 }
} // end of for loop.
} // end of run() method.
public static void main(String[] args)
{
// Create two objects to represent two tasks.
 MyThread th1 = new MyThread("Cut the ticket"); // Passing task as an argument to its constructor.
 MyThread th2 = new MyThread("Show your seat number");

// Create two objects of Thread class and pass two objects as parameter to constructor of Thread
class.
 Thread t1 = new Thread(th1);
 Thread t2 = new Thread(th2);
 t1.start();
 t2.start();
 }
}
```

Output:

```
Cut the ticket : 1
Show your seat number : 1
Show your seat number : 2
Cut the ticket : 2
Show your seat number : 3
Cut the ticket : 3
Show your seat number : 4
Cut the ticket : 4
Show your seat number : 5
Cut the ticket : 5
```

In the preceding example program, we have created two threads on two objects of MyThread class.

Here, we created two objects to represent two tasks.
When we will run the above program, the main thread starts running immediately. Two threads will generate from the main thread that will perform two different tasks.

Priority of a Thread (Thread Priority)

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.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the 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 assign 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 of priority of a Thread:

```
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 argvs[])
{
// 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
```

Java Synchronization

- Synchronization is a process of handling resource accessibility by multiple thread requests.
- The main purpose of synchronization is to avoid thread interference.
- At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- The synchronization keyword in java creates a block of code referred to as critical section.

General Syntax:

```
synchronized (object)
{
 //statement to be synchronized
}
```

Why we need Syncronization?

If we do not use syncronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization

```
class Table{
void printTable(int n){//method not synchronized
 for(int i=1;i<=5;i++){
  System.out.println(n*i);
  try{
  Thread.sleep(400);
  }catch(Exception e){System.out.println(e);}
 }

}
}

class MyThread1 extends Thread{
Table t;
```

```
 MyThread1(Table t){
 this.t=t;
 }
 public void run(){
 t.printTable(5);
 }

 }
 class MyThread2 extends Thread{
 Table t;
 MyThread2(Table t){
 this.t=t;
 }
 public void run(){
 t.printTable(100);
 }
 }

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
```

```
}

}
```

Output:

```
5

100

10

200

15

300

20

400

25

500
```

1. Java Synchronized Method

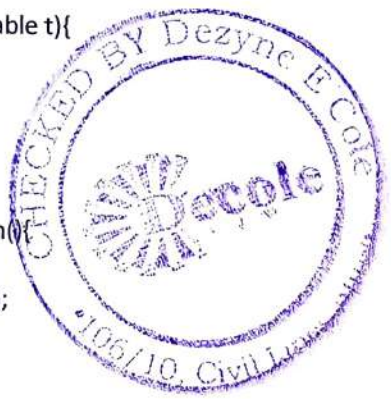
- If you declare any method as synchronized, it is known as synchronized method.

- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

//example of java synchronized method

```
class Table{

synchronized void printTable(int n){//synchronized method

 for(int i=1;i<=5;i++){

  System.out.println(n*i);

  try{
```

```
    Thread.sleep(400);
   }catch(Exception e){System.out.println(e);}
  }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
```



```
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Synchronized Block in Java

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

- If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

```
synchronized (object reference expression) {
 //code block
}
```

Example of Synchronized Block

```
class Table

{

 void printTable(int n){

  synchronized(this){//synchronized block

   for(int i=1;i<=5;i++){

    System.out.println(n*i);

    try{

     Thread.sleep(400);

    }catch(Exception e){System.out.println(e);}

   }

  }

 }//end of the method

}


class MyThread1 extends Thread{

Table t;

MyThread1(Table t){

this.t=t;

}
```

```
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

5

10

15

20

25

100

200

300

400

500

Inter-thread Communication in Java

- **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.

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()

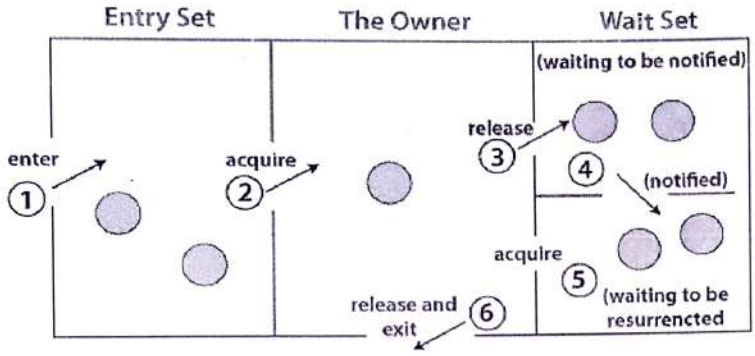
3) notifyAll() method

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 on 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.

Example of Inter Thread Communication in Java

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
```

```
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

Java Thread suspend() method

- The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs.
- This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Syntax

public final void suspend()

Return

This method does not return any value.

Example

```
public class JavaSuspendExp extends Thread
{
  public void run()
  {
```

```
    for(int i=1; i<5; i++)
    {
      try
      {
        // thread to sleep for 500 milliseconds
        sleep(500);
        System.out.println(Thread.currentThread().getName());
      }catch(InterruptedException e){System.out.println(e);}
      System.out.println(i);
    }
  }
  public static void main(String args[])
  {
    // creating three threads
    JavaSuspendExp t1=new JavaSuspendExp ();
    JavaSuspendExp t2=new JavaSuspendExp ();
    JavaSuspendExp t3=new JavaSuspendExp ();
    // call run() method
    t1.start();
    t2.start();
    // suspend t2 thread
    t2.suspend();
    // call run() method

    t3.start();

  }

}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

Java Thread resume() method

The **resume()** method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again.

Syntax

public final void resume()

Return value

This method does not return any value.

Example

```
public class JavaResumeExp extends Thread
{
  public void run()
  {
    for(int i=1; i<5; i++)
    {
      try
      {
        // thread to sleep for 500 milliseconds
        sleep(500);
        System.out.println(Thread.currentThread().getName());
      }catch(InterruptedException e){System.out.println(e);}
      System.out.println(i);
    }
  }
  public static void main(String args[])
  {
    // creating three threads
    JavaResumeExp t1=new JavaResumeExp ();
    JavaResumeExp t2=new JavaResumeExp ();
    JavaResumeExp t3=new JavaResumeExp ();
    // call run() method
    t1.start();
    t2.start();
    t2.suspend(); // suspend t2 thread
    // call run() method
    t3.start();
    t2.resume(); // resume t2 thread
  }
}
```

Output:

```
Thread-0
1
Thread-2
```

```
1
Thread-1
1
Thread-0
2
Thread-2
2
Thread-1
2
Thread-0
3
Thread-2
3
Thread-1
3
Thread-0
4
Thread-2
4
Thread-1
4
```

Java Thread stop() method

The **stop()** method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.

Syntax

```
public final void stop()
```

Example

```
public class JavaStopExp extends Thread
{
  public void run()
  {
    for(int i=1; i<5; i++)
    {
      try
      {
        // thread to sleep for 500 milliseconds
        sleep(500);
        System.out.println(Thread.currentThread().getName());
      }catch(InterruptedException e){System.out.println(e);}
      System.out.println(i);
    }
}
public static void main(String args[])
{
  // creating three threads
```

```
    JavaStopExp t1=new JavaStopExp ();
    JavaStopExp t2=new JavaStopExp ();
    JavaStopExp t3=new JavaStopExp ();
    // call run() method
    t1.start();
    t2.start();
    // stop t3 thread
    t3.stop();
    System.out.println("Thread t3 is stopped");
  }
}
```

Determining When a Thread Ends

- It is often useful to know when a thread has ended. For example, in the preceding examples, for the sake of illustration it was helpful to keep the main thread alive until the other threads ended.
- In those examples, this was accomplished by having the main thread sleep longer than the child threads that it spawned. This is, of course, hardly a satisfactory or generalizable solution!
- Fortunately, **Thread** provides two means by which you can determine if a thread has ended. First, you can call **isAlive( )** on the thread. Its general form is shown here:
- final boolean isAlive( )
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

University Pattern Questions

1. What is object oriented programming (OOP)?
2. What is abstraction?
3. Define polymorphism in OOP?
4. What is the advantage of inheritance in OOP?
5. What is a constructor?
6. What is a class in OOP?
7. How do you declare an object in Java?
8. What is an applet?
9. Explain importing package in Java?
10. What is the use of interfaces in Java?
11. What are the application areas for Java?
12. What makes Java a secure internet programming language?
13. Explain the usage of "this" keyword in Java?
14. What is multithreading?
15. What is Event handling? Explain Event classes?
16. List and Explain the key characteristics of Java programming language?
17. Write a small program to illustrate the use of classes in Java. Also explain nested classes?
18. Write a small program to explain the usage of methods in Java. Explain method overloading?
19. Explain Exception handing mechanism in Java.Illustrate using a small example program?
20. Write short notes on any two of the following:
    a. Java Packages
    b. I/0 (input/output) stream management.
    c. Javax.swing package
    d. Event Listener interfaces
21. Why Java is platform independent language?
22. What is the range of long data type?
23. What will be the output of following code :

```
int a=10;
System.out.println(a>>2);
```

24. Which operator in Java is known as ternary operator?
25. What is bytecode?
26. Define command line argument.
27. Explain the garbage collection.
28. Describe the JVM and Java API.
29. Define bitwise and shiftwise operator with example.
30. What is base class of all Java classes ?
31. What is return type of constructor?
32. What is final variable?
33. What is method overloading?Explain it with an example.
34. What is final class? How it is useful?
35. What is constructor?Explain various types of constructors with suitable example.
36. Explain the term run time polymorphism.
37. List various types of inheritance.
38. Write the uses of 'this' keyword.
39. What is the difference between String and StringBuffer class?
40. Write a program to create your own exception if marks entered are less than 0 or >100.
41. Explain life cycle of Thread?

42.Give an example to explain multithreading concept in Java.
43. Describe the charAt() method related to string.
44. Define access specifiers.
45. What is synchronization?
46. Write the difference between String and StringTokenizer. Explain through example.
47. Distinguish between Final,Finally and Finalize.
48. Explain Legacy classes.
49. What is Thread? Write different ways for creating thread.Explain thread properties with example
50. Create a program which explains Vector and Stack class.
51. What is package?How a user defined package is created?Explain with example.
52. a) What is Object Oriented Programming Paradigm ? Explain briefly some features of java lang.
53. What is JVM ? Explain concept of byte code generation in java.
54. a) What is datatypes ? Explain various datatypes of java.
55. Write a program to sort 10 numbers in ascending order.
56. What is a package. How user defined package is created. Discuss them with example.
57. Explain briefly :
    a) Garbage Collector
    b) Abstract Class
    c) Super Keyword
    d) This Keyword
    e) Throws
58. What do you understand by Polymorphism?
59. What is overloading?
60. What is Thread?
61. Differentiate between string and string buffer.
62. Explain the need of interfaces.
63. Give difference between Java and C++.
64. Write a program of method overriding.
65. What is JVM? Explain the advantages of JVM. Also give functionality of loader.
66. Write a program to sort elements of one dimensional array. Also include exception handling code in program.
67. What is inheritance? Explain different types of inheritance with example.
68. What is multithreading? Write a program to implement multithreading.
69. What is package? Write steps to create and run a package. Also give advantage of package.
70. How class path is set for packages.
71. How inheritance is implemented by interface.
72. What is the role of finally?
73. How object reference is different from object.
74. Define instance of operator.s
75. Define relational operators and Boolean logic operators.
76. What is the difference between type conversion and type casting?
77. Differentiate between the use of throw and throws.
78. Explain Thread synchronization and thread execution
79. What is conceptual difference between abstraction and encapsulation?
80. How does Java implement the Portability of code?
81. Write note on New and This.
82. What is an exception? Explain the procedure of how to handle exception.
83. What is package? Write steps to create and run a package. Also give advantage of package.
84. Explain the use of 'Static' and 'Final' keyword.
85. What is For-Each loop in java. Write a program which shows addition of two 2*2 matrix's.
86. What is Exception Handling? Explain the concept of multiple catch statements. Write a program

Which implements finally block.

87. What is need of Synchronization in multithreading? Explain synchronized method with example?
88. What are Threads? How threads are created in Java? Explain the concept of thread priority with an example.
89. What is abstract class.
90. What is Runnable Interface.
91. What is difference between Class and Interface? Explain it with program.
92. What is multithreading and how java implements it? Also explain difference between synchronized method and synchronized block with a suitable program.
93. Describe charAt() method related to string.
94. Define Access specifiers.