**Java** Introduction:

* + Java is a high level, object-oriented, secure programming language.
  + It is developed as Sun Microsystems by James Gosling in 1991.
  + It is also popularly known as OAK.
  + In 1995, Sun Microsystems is changed as Java.
  + It is takeover by Oracle Corporation in 2005.

Editions of Java:

* + Java Standard Edition – create programs for desktop computers.
  + Java Enterprise Edition – large programs run on the server and manages the high-level complex and traffic transactions.
  + Java Micro Edition – develop applications for set-top boxes, mobiles, etc..,

Types of Java Applications:

* + Standalone Applications – AWT, Swing and JavaFX are uses GUI components like buttons, list scroll panel, menu, etc...,
  + Enterprise Applications – distributed in nature.
  + Web Applications – run on the server and JSP, Spring, Servlet, Herbinate technologies used for web applications.
  + Mobile Applications – Java ME is a cross-platform to develop mobile applications which run across smartphones. Java platform is used in App development in Android.

Java Platform:

* + Java Platform is a collection of programs.
  + It helps to develop and run a program in Java programming language.
  + Eclipse is used for Java.
  + It is the environment for developing and managing Java applets and applications.

Features of Java:

* + **Simple:**

Java is simple, clean and easy to understand.

* + **Object-Oriented:**

In java, everything in the form of object. It means has a soe data and behaviour. Atleast one program has a one class and object.

* + **Robust:**

Check error at run time and compile time.

* + **Secure:**

Java is a secure programming language.

* + **Platform-independent:**

We have write code once and runs at anywhere like Linux OS, Windows OS, Mac OS.

* + **Portable:**

Java Byte code can be carried to any platform. No implementation-dependent features. Everything related to storage is predefined.

* + **High Performance:**

Java is an interpreted language. Java enables high performance with the use of the Just-In-Time compiler.

* + **Distributed:**

Java also has networking facilities. It is designed for the distributed environment of the internet because it supports TCP/IP protocol. It can run over the internet. EJB and RMI are used to create a distributed system.

* + **Multithreading:**

Java also supports multi-threading. It means to handle more than one job a time.

Reference Link**:**

[**https://www.javatpoint.com/java-basics**](https://www.javatpoint.com/java-basics)

Variables:

* The name which we can be stores some values.
* In java, a specified format to declare a variable.

Data Types:

* Data Types means different size and values stored in the variable.
* It have two types:

1. Primitive Data Types
2. Non-Primitive Data Types
3. **Primitive Data Types:**
   * + Primitive has only single values and have no special capabilities.
     + It starts with lowercase letter.
     + It has 8 types:
4. Byte (number,1 byte)
5. Short (number,2 bytes)
6. Int (number, 4 bytes)
7. Long (number, 4 bytes)
8. Float (float number, 8 bytes)
9. Double (float number, 8 bytes)
10. Char (a character, 2 bytes)
11. Boolean (true or false, 1 bytes)
12. **Non-Primitive Data Types:**
    * + Non-Primitive Data types also known as reference types.
      + Because, it is referred to the object.
      + It is not predefined but created by the programmer.
      + It can be assigned with null.
      + It starts with uppercase letters.
      + It has 2 types:
13. String
14. Array

String:

* String means array of characters (or) sequence of characters.
* It stores the address of an object in memory.
* Strings are immutable, which means that they cannot be changed once they are created.
* Keyword: “String”

Example Program:

package lesson1;

public class MyClass {

public static void main (String [ ] args){

String myString = “Hello World”;

int myStringLength = myString.length();

String myStringinCase1 = myString.toLowerCase();

String myStringinCase2 = myString.toUpperCase();

System.out.println(myStringLength);

System.out.println(myStringinCase1);

System.out.println(myStringinCase2);

System.out.println(myString.replace(‘e’ , ‘a’));

System.out.println(myStringindexOf(‘o’));

}

}

Output:

11

hello world

HELLO WORLD

Hallo World

4

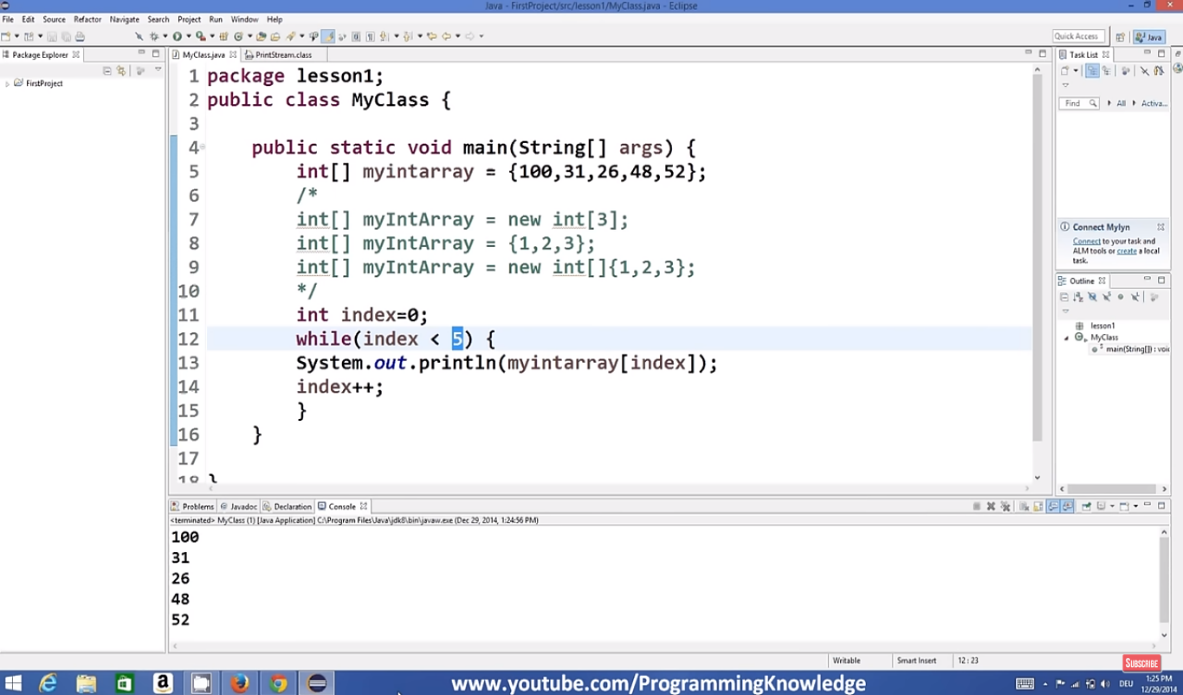
Array:

* Array is same as variable but stores more than one values at a time.
* But only condition is array is only storing same kind of arrays.

Array Declaration:

* int [] myIntArray = new int[3];
* int [] myIntArray = {1,2,3}
* int [] myIntArray = new int[] {1,2,3}

Example Program:



Reference Link**:**

**For String:**

[**https://www.youtube.com/watch?v=sliTbMkQBZ4&list=PLS1QulWo1RIbfTjQvTdj8Y6yyq4R7g-Al&index=15**](https://www.youtube.com/watch?v=sliTbMkQBZ4&list=PLS1QulWo1RIbfTjQvTdj8Y6yyq4R7g-Al&index=15)

**For Array:**

[**https://www.youtube.com/watch?v=zQgsXkYqjc&list=PLS1QulWo1RIbfTjQvTdj8Y6yyq4R7g-Al&index=14**](https://www.youtube.com/watch?v=zQgsXkYqjc&list=PLS1QulWo1RIbfTjQvTdj8Y6yyq4R7g-Al&index=14)

Conditional Statements:

**If-Else Statement:**

* It is a conditional statement which evaluates some condition whether it is true or false based upon this condition it executes some code.
* Keyword: “If”
* Inside the two brackets evaluate some Boolean condition or Boolean expression.
* Syntax:

if(condition){

// block of code to be executed if the condition is true

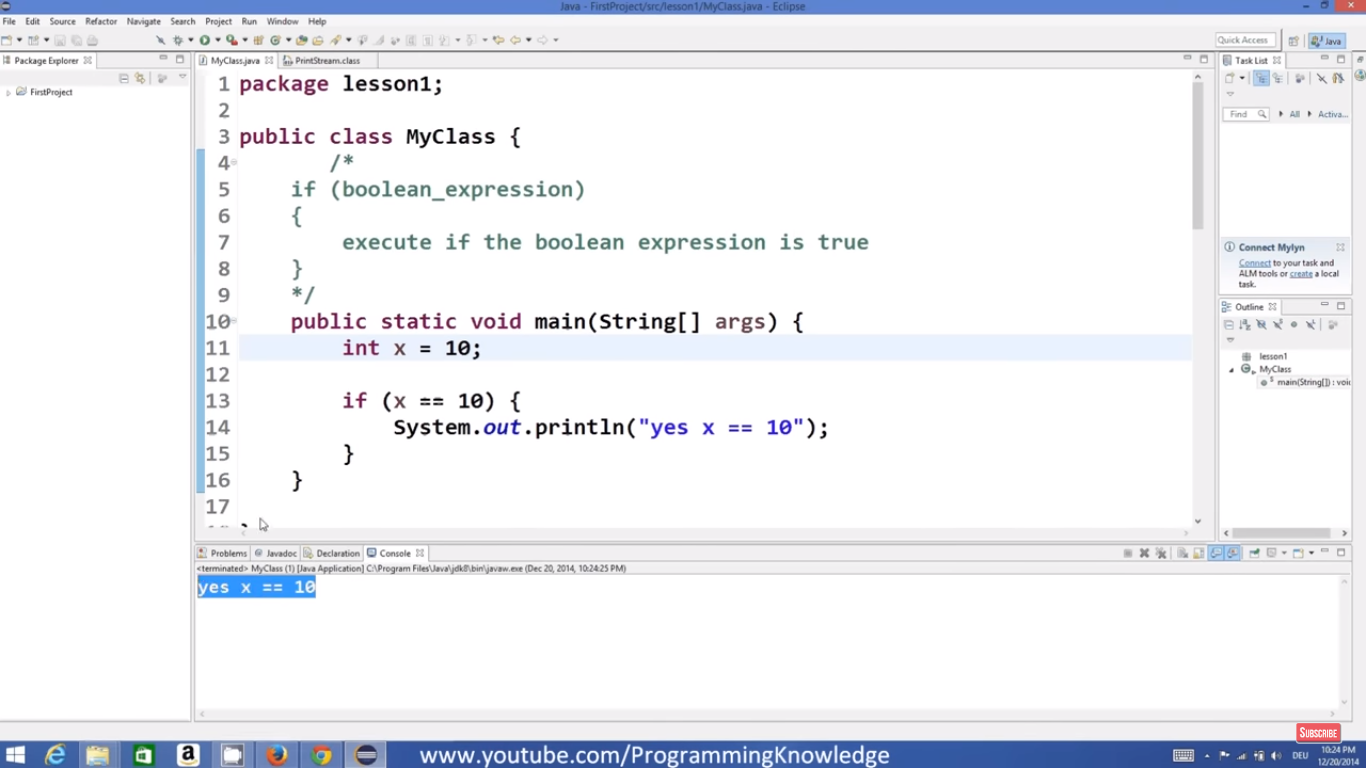
}

else {

// block of code to be executed if the condition is false

}

Example Program:



Switch Statement:

* Whenever you have to make then multiple condition checking then we use switch statement.
* Syntax:

switch(expression) {

case x:

// code block

break;

case y:

// code block

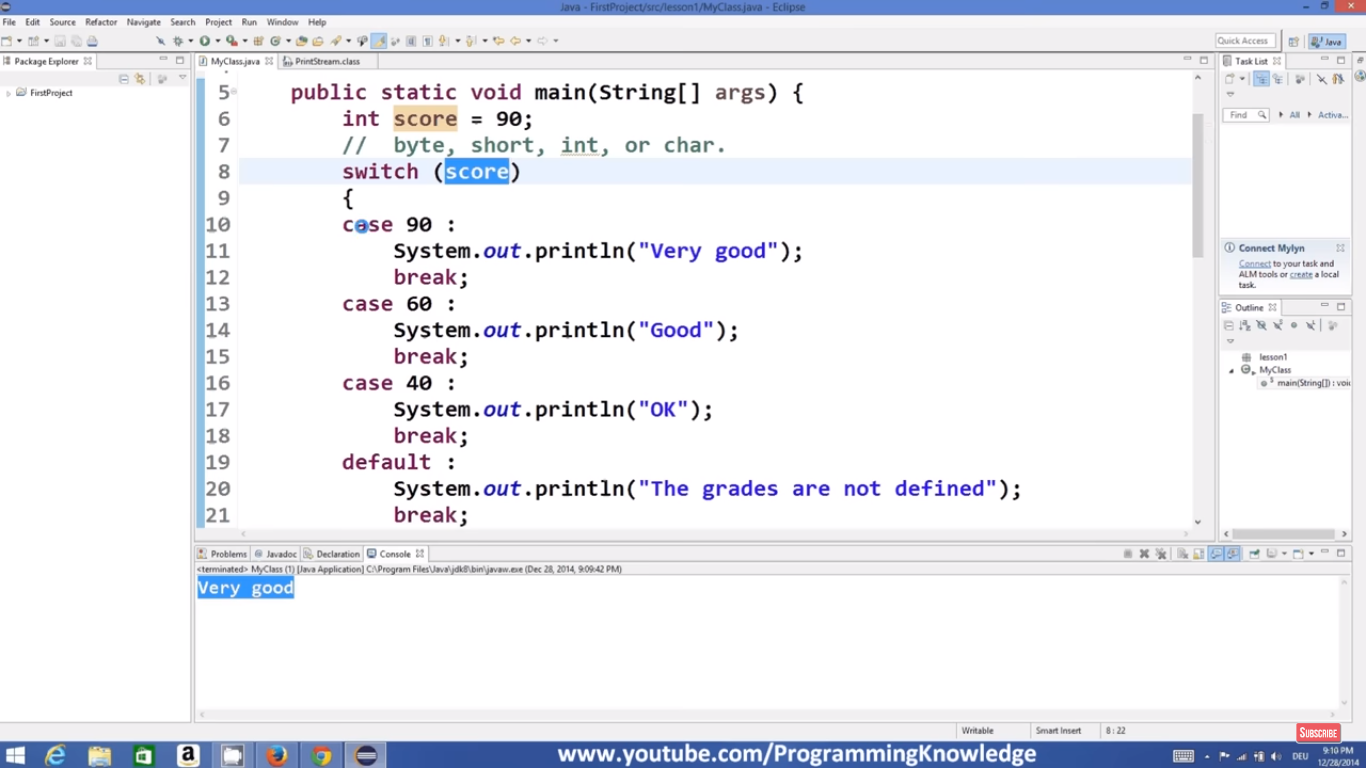
break;

default:

// code block

}

Example Program:

****

Looping Statements:

**While loop Statement:**

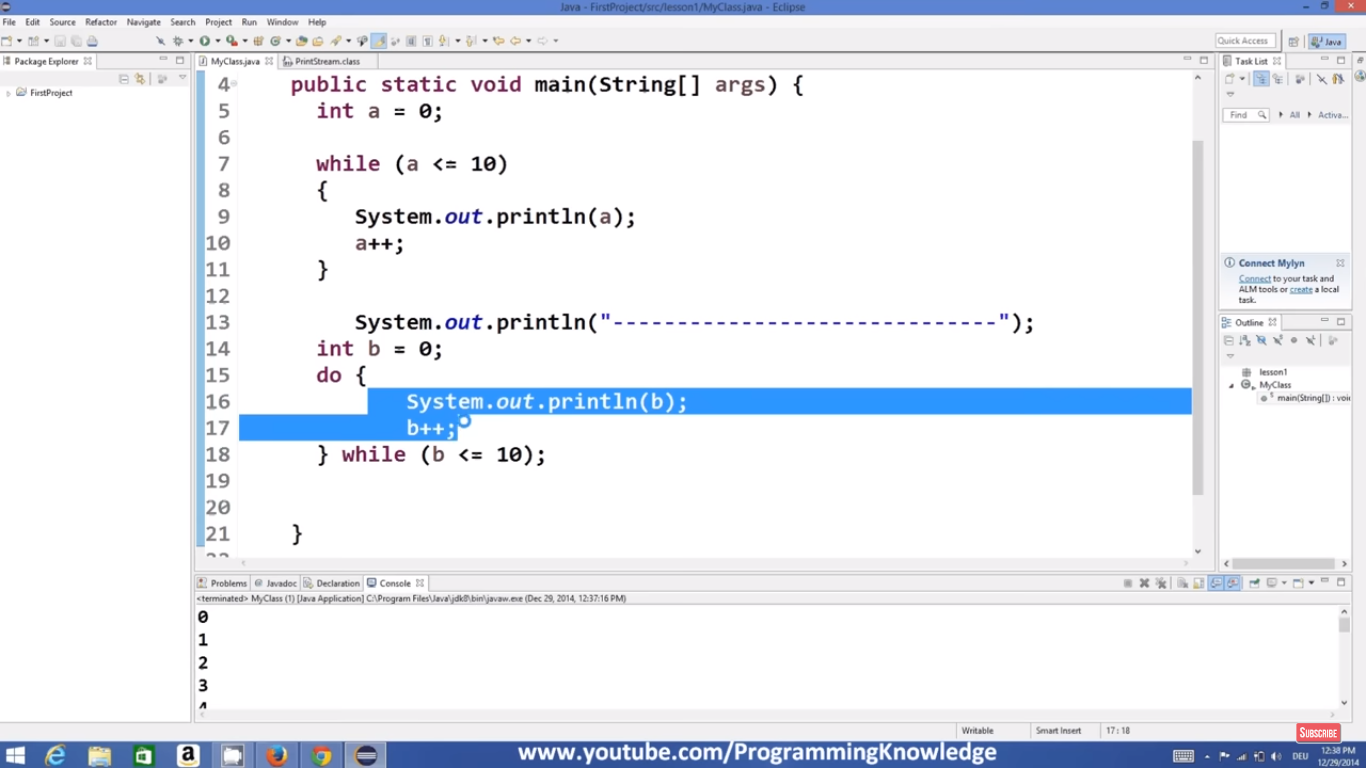
* A loop is a piece of code and statement that executes some block of code again and again until some condition is met or true without rewriting it.
* It is most basic kind of loop statement.
* Syntax:

while (condition) {

*// code block to be executed*

}

Example Program:



**Do-While loop Statement:**

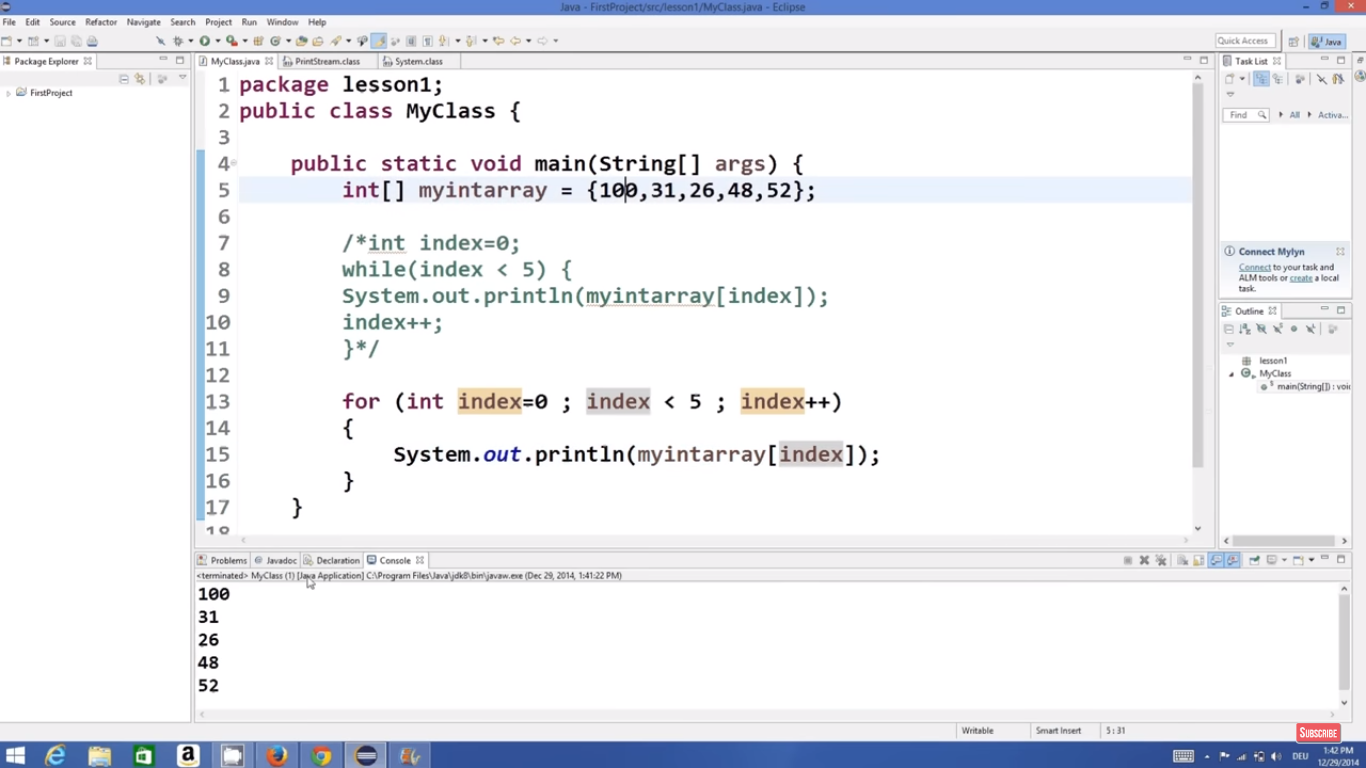
* It is also a kind of loop.
* But it’s slightly different from while loop.
* End of the curly braces we just write while and then in the round brackets comes your expression which you want to evaluate and this will be the Boolean expression.
* Keyword: “Do”
* Syntax:

do (condition) {

*// code block to be executed*

} while (condition);

Example Program:



**For loop Statement:**

* For loop is same as while loop.
* In for loop, there are 3 types of expressions.

1. Init (or) Initialization
2. Expression
3. Increment Value

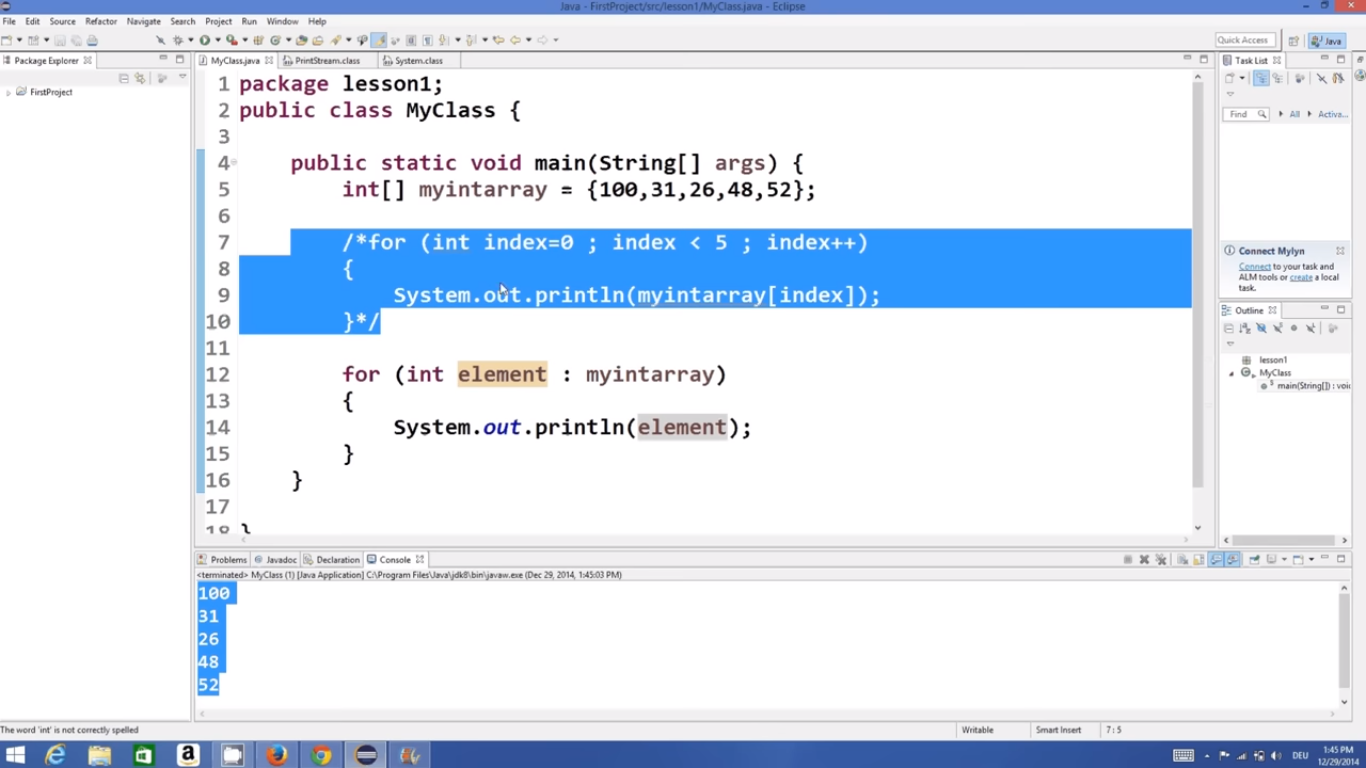
* Syntax:

for (*statement 1*; *statement 2*; *statement 3*) {

*// code block to be executed*

}

Example Program:



Buffered Reader in Java:

// Java Program for taking user

// input using BufferedReader Class

import java.io.\*;

class GFG {

// Main Method

public static void main(String[] args)

throws IOException

{

// Creating BufferedReader Object

// InputStreamReader converts bytes to

// stream of character

BufferedReader bfn = new BufferedReader(

new InputStreamReader(System.in));

// String reading internally

String str = bfn.readLine();

// Integer reading internally

int it = Integer.parseInt(bfn.readLine());

// Printing String

System.out.println("Entered String : " + str);

// Printing Integer

System.out.println("Entered Integer : " + it);

}

}

### Using Buffer Reader Class To Read the Input

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.io.BufferedReader;

import java.io.InputStreamReader;

class Easy {

public static void main(String[] args)

{

// creating the instance of class BufferedReader

BufferedReader reader = new BufferedReader(

new InputStreamReader(System.in));

String name;

try {

System.out.println("Enter your name");

name = reader.readLine(); // taking string input

System.out.println("Name=" + name);

}

catch (Exception e) {

}

}

}

## 2. Using Scanner Class for Taking Input in Java

// Java Program to show how to take

// input from user using Scanner Class

import java.util.\*;

class GFG {

public static void main(String[] args)

{

// Scanner definition

Scanner scn = new Scanner(System.in);

// input is a string ( one word )

// read by next() function

String str1 = scn.next();

// print String

System.out.println("Entered String str1 : " + str1);

// input is a String ( complete Sentence )

// read by nextLine()function

String str2 = scn.nextLine();

// print string

System.out.println("Entered String str2 : " + str2);

// input is an Integer

// read by nextInt() function

int x = scn.nextInt();

// print integer

System.out.println("Entered Integer : " + x);

// input is a floatingValue

// read by nextFloat() function

float f = scn.nextFloat();

// print floating value

System.out.println("Entered FloatValue : " + f);

}

}

**Output :**

Entered String str1 : Geeks  
Entered String str2 : Geeks For Geeks  
Entered Integer : 123  
Entered FloatValue : 123.090

// Java Program to implement

// Scanner Class to take input

import java.io.\*;

import java.util.Scanner;

// Driver Class

class Easy {

// main function

public static void main(String[] args)

{

// creating the instance of class Scanner

Scanner obj = new Scanner(System.in);

String name;

int rollno;

float marks;

System.out.println("Enter your name");

// taking string input

name = obj.nextLine();

System.out.println("Enter your rollno");

// taking integer input

rollno = obj.nextInt();

System.out.println("Enter your marks");

// taking float input

marks = obj.nextFloat();

// printing the output

System.out.println("Name=" + name);

System.out.println("Rollno=" + rollno);

System.out.println("Marks=" + marks);

}

}

**Output**

Enter your name  
Geeks   
Enter your rollno  
5  
Enter your marks  
84.60  
Name=Geeks   
Rollno=5  
Marks=84.60

### i). For Number Formatting

// Java Program to demonstrate

// Use of printf to

// Formatting Integer

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main (String[] args) {

int a=10000;

//System.out.printf("%.d%n",a);

System.out.printf("%,d%n",a);

}

}

Operators in Java:

In Java, operators are symbols that perform operations on operands. Operands can be variables, literals, method calls, or expressions. Java provides a variety of operators for arithmetic, relational, logical, bitwise, assignment, and other operations.

### 1. Arithmetic Operators:

Arithmetic operators are used to perform basic arithmetic operations like addition, subtraction, multiplication, division, and modulus.

* Addition: **+**
* Subtraction: **-**
* Multiplication: **\***
* Division: **/**
* Modulus (Remainder): **%**

Example:

int a = 10;

int b = 3;

int sum = a + b; // sum = 13

int difference = a - b; // difference = 7

int product = a \* b; // product = 30

int quotient = a / b; // quotient = 3

int remainder = a % b; // remainder = 1

### 2. Relational Operators:

Relational operators are used to compare values.

* Equal to: **==**
* Not equal to: **!=**
* Greater than: **>**
* Less than: **<**
* Greater than or equal to: **>=**
* Less than or equal to: **<=**

Example:

int x = 5;

int y = 10;

boolean isEqual = (x == y);

boolean isNotEqual = (x != y);

boolean isGreater = (x > y);

boolean isLess = (x < y);

boolean isGreaterOrEqual = (x >= y);

boolean isLessOrEqual = (x <= y);

### 3. Logical Operators:

Logical operators are used to perform logical operations on boolean expressions.

* Logical AND: **&&**
* Logical OR: **||**
* Logical NOT: **!**

Example:

boolean p = true;

boolean q = false;

boolean logicalAnd = (p && q);

boolean logicalOr = (p || q);

boolean logicalNotP = !p;

boolean logicalNotQ = !q;

### 4. Assignment Operators:

Assignment operators are used to assign values to variables.

* Assignment: **=**
* Add and assign: **+=**
* Subtract and assign: **-=**
* Multiply and assign: **\*=**
* Divide and assign: **/=**
* Modulus and assign: **%=**

Example:

int num = 10;

num += 5; // num = num + 5 = 15

num -= 3; // num = num - 3 = 12

num \*= 2; // num = num \* 2 = 24

num /= 4; // num = num / 4 = 6

num %= 5; // num = num % 5 = 1

Strings in Java:

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

Example: String name = “Oviya”;

Example Program:

public class StringExample {

public static void main(String args[])

{

String str = new String("example");

System.out.println(str);

}

}

Output:

example

Immutable Strings in Java:

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

Example Program:

// Java Program to demonstrate Immutable String in Java

import java.io.\*;

class GFG {

public static void main(String[] args)

{

String s = "Sachin";

// concat() method appends

// the string at the end

s.concat(" Tendulkar");

// This will print Sachin

// because strings are

// immutable objects

System.out.println(s);

}

}

Output:

Sachin

Arrays in Java:

* In Java, all arrays are dynamically allocated. (discussed below)
* Arrays may be stored in contiguous memory [consecutive memory locations].
* Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++, where we find length using sizeof.
* A Java array variable can also be declared like other variables with [] after the data type.
* The variables in the array are ordered, and each has an index beginning with 0.
* Java array can also be used as a static field, a local variable, or a method parameter.

Jagged Array:

A jagged[array](https://en.wikipedia.org/wiki/Jagged_array) is an array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.

Example Program:

// Program to demonstrate 2-D jagged array in Java

class Main {

public static void main(String[] args)

{

// Declaring 2-D array with 2 rows

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

// Making the above array Jagged

// First row has 3 columns

arr[0] = new int[3];

// Second row has 2 columns

arr[1] = new int[2];

// Initializing array

int count = 0;

for (int i = 0; i < arr.length; i++)

for (int j = 0; j < arr[i].length; j++)

arr[i][j] = count++;

// Displaying the values of 2D Jagged array

System.out.println("Contents of 2D Jagged Array");

for (int i = 0; i < arr.length; i++) {

for (int j = 0; j < arr[i].length; j++)

System.out.print(arr[i][j] + " ");

System.out.println();

}

}

}

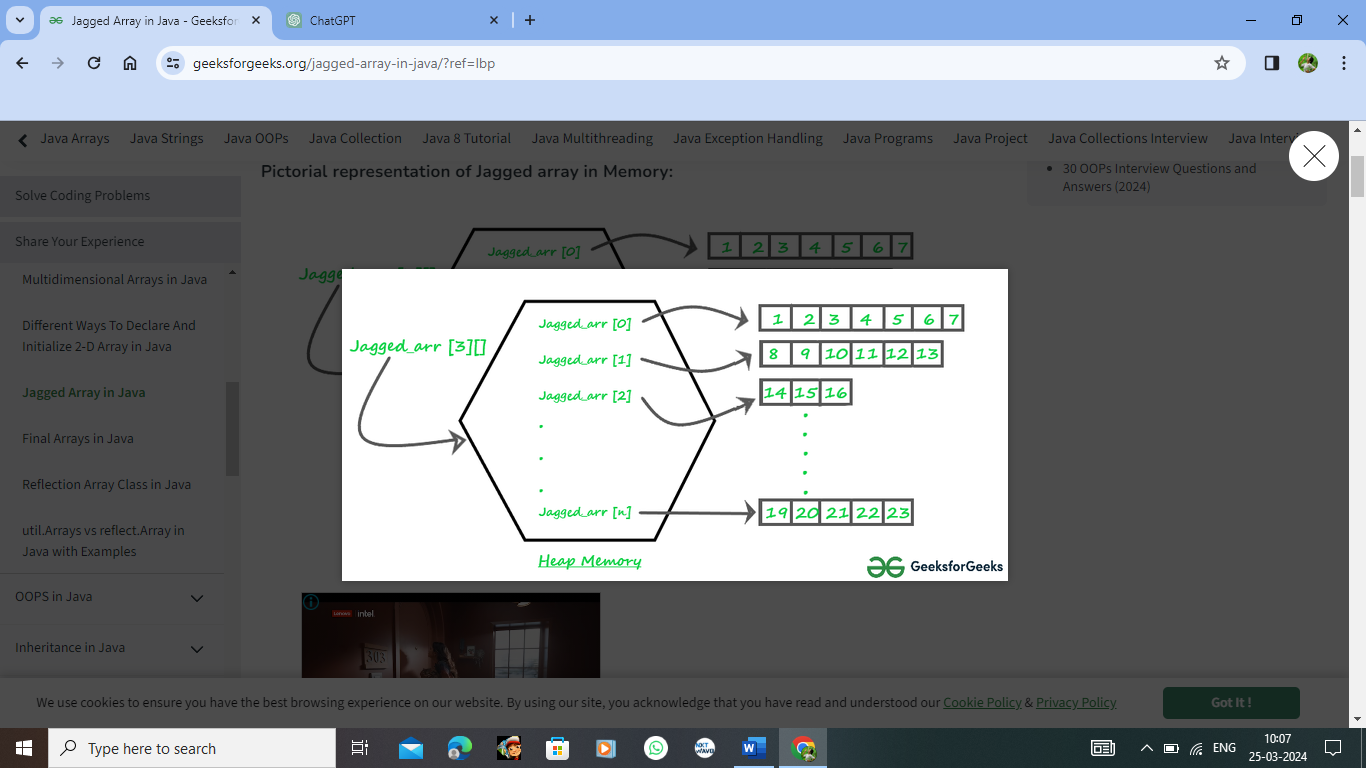
Output:

Contents of 2D Jagged Array

0 1 2

3 4

Diagram:



Object-Oriented Programming Concept in Java:

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

Diagram Representation:



OOPS concepts are as follows:

1. [Class](https://www.geeksforgeeks.org/classes-objects-java/)
2. [Object](https://www.geeksforgeeks.org/classes-objects-java/)
3. [Method](https://www.geeksforgeeks.org/methods-in-java/)and [method passing](https://www.geeksforgeeks.org/message-passing-in-java/)
4. Pillars of OOPs
   * [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/)
   * [Encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java/)
   * [Inheritance](https://www.geeksforgeeks.org/inheritance-in-java/)
   * [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java/)
     + Compile-time polymorphism
     + Runtime polymorphism

Classes and Objects in Java:

In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities. The class represents a group of objects having similar properties and behavior. For example, the animal type **Dog** is a class while a particular dog named **Tommy**is an object of the **Dog** class.

### Properties in Java:

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

### Class Declaration in Java:

access\_modifier **class** <class\_name>

{

data member;

method;

constructor;

nested class;

interface;

}

Java Objects:

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

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

Example of an object: dog

**class and objects one simple java program :**

public class GFG {

static String Employee\_name;

static float Employee\_salary;

static void set(String n, float p) {

Employee\_name = n;

Employee\_salary = p;

}

static void get() {

System.out.println("Employee name is: " +Employee\_name );

System.out.println("Employee CTC is: " + Employee\_salary);

}

public static void main(String args[]) {

GFG.set("Rathod Avinash", 10000.0f);

GFG.get();

}

}

**Output**

Employee name is: Rathod Avinash

Employee CTC is: 10000.0

Inheritance:

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

**Syntax :**

class DerivedClass extends BaseClass   
{   
 //methods and fields   
}

//base class or parent class or super class

class A{

//parent class methods

void method1(){}

void method2(){}

}

//derived class or child class or base class

class B extends A{ //Inherits parent class methods

//child class methods

void method3(){}

void method4(){}

}

Inheritance Types:

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

Abstraction:

**Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

//abstract class

abstract class GFG{

//abstract methods declaration

abstract void add();

abstract void mul();

abstract void div();

}

Encapsulation:

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield. Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data hiding. Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

//Encapsulation using private modifier

//Employee class contains private data called employee id and employee name

class Employee {

private int empid;

private String ename;

}

Polymorphism in Java:

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

Types:

* Compile-time Polymorphism
* Runtime Polymorphism

Eg:

sleep(1000) //millis

sleep(1000,2000) //millis,nanos

// Java program to Demonstrate Polymorphism

Eg:

// This class will contain

// 3 methods with same name,

// yet the program will

// compile & run successfully

public class Sum {

// Overloaded sum().

// This sum takes two int parameters

public int sum(int x, int y)

{

return (x + y);

}

// Overloaded sum().

// This sum takes three int parameters

public int sum(int x, int y, int z)

{

return (x + y + z);

}

// Overloaded sum().

// This sum takes two double parameters

public double sum(double x, double y)

{

return (x + y);

}

// Driver code

public static void main(String args[])

{

Sum s = new Sum();

System.out.println(s.sum(10, 20));

System.out.println(s.sum(10, 20, 30));

System.out.println(s.sum(10.5, 20.5));

}

}

Constructors in Java:

Java constructors or constructors in Java is a terminology used to construct something in our programs. 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. It can be used to set initial values for object attributes.

Example Program:

// Java Program to demonstrate

// Constructor

import java.io.\*;

// Driver Class

class Geeks {

// Constructor

Geeks()

{

super();

System.out.println("Constructor Called");

}

// main function

public static void main(String[] args)

{

Geeks geek = new Geeks();

}

}

Output:

Constructor Called

### **How Java Constructors are Different From Java Methods?**

* Constructors must have the same name as the class within which it is defined 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.

Instance Methods:

Instance methods are methods that require an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class in which the method is defined.

Syntax:

public void geek(String name)

{

// code to be executed....

}

// Return type can be int, float String or user defined data type.

## **Java Static Methods:**

Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the **class name itself** or reference to the Object of that class.

Syntax:

public static void geek(String name)

{

// code to be executed....

}

// Must have static modifier in their declaration.

// Return type can be int, float, String or user defined data type.

## Java Abstract Method:

The abstract Method is used for creating blueprints for classes or interfaces. Here methods are defined but these methods don’t provide the implementation. Abstract Methods can only be implemented using subclasses or classes that implement the interfaces.

These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the super-class. Thus, a subclass must [override](https://www.geeksforgeeks.org/overriding-in-java/) them to provide a method definition.

## Overriding in Java:

In Java, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Functional Interface in Java:

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for integrity and simplicity. There are no solely functions present in a programming language called Java. Functions in the Java programming language are part of a class, and if someone wants to use them, they have to use the class or object of the class to call any function.

Nested Interface:

We can declare interfaces as members of a class or another interface. Such an interface is called a member interface or nested interface. **Interface in a class** Interfaces (or classes) can have only public and default access specifiers when declared outside any other class. Refer to [Access modifiers for classes or interfaces in the Java](https://www.geeksforgeeks.org/access-modifiers-for-classes-or-interfaces-in-java/) article for details. This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as**c\_name.i\_name** where **c\_name** is the name of the class in which it is nested and **i\_name** is the name of the interface itself.

Marker Interface in Java:

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces. 

public interface Serializable

{

// nothing here

}

Wrapper Class:

A wrapper class wraps (encloses) around a data type and gives it an object appearance. Wrapper classes are final and immutable. Two concepts are there in the wrapper classes namely autoboxing and unboxing.

Autoboxing:

Autoboxing is a procedure of converting a primitive value into an object of the corresponding [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/). For example, converting int to Integer class.

Unboxing:

Unboxingis a procedure of converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int.

Keywords in Java:

In Java, 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.

Important Keywords in Java:

1. **abstract:**It is a non-access modifier applicable for classes and methods. It is used to achieve abstraction. For more, refer to [abstract keyword in java](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. **enum:**It is used to define [enum in Java](https://www.geeksforgeeks.org/enum-in-java/)
3. **instanceof:**It is used to know whether the object is an instance of the specified type (class or subclass or interface).
4. **private:**It is an access modifier. Anything declared private cannot be seen outside of its class.
5. **protected:**If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
6. **public:**Anything declared public can be accessed from anywhere. For more on Access Modifiers, refer to [Access Modifiers in Java](https://www.geeksforgeeks.org/access-modifiers-java/)
7. **static:**It is used to create a member(block, method, variable, nested classes) that can be used by itself, without reference to a specific instance. For more, refer [static keyword in java](https://www.geeksforgeeks.org/static-keyword-java/)
8. **strictfp:**It is used for restricting floating-point calculations and ensuring the same result on every platform while performing operations in the floating-point variable. For more refer [strictfp keyword in Java](https://www.geeksforgeeks.org/strictfp-keyword-java/)
9. **synchronized:**Applicable for blocks methods. It is used to get synchronization in java. For more, refer to [Synchronized in Java](https://www.geeksforgeeks.org/synchronized-in-java/)
10. **transient:**transient is a variables modifier used in [serialization](https://www.geeksforgeeks.org/serialization-in-java/). At the time of serialization, if we don’t want to save the value of a particular variable in a file, then we use the transient keyword. For more refer [transient keyword in Java](https://www.geeksforgeeks.org/transient-keyword-java/)
11. **volatile**: The volatile modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program. For more, refer to [volatile keyword in java](https://www.geeksforgeeks.org/volatile-keyword-in-java/)

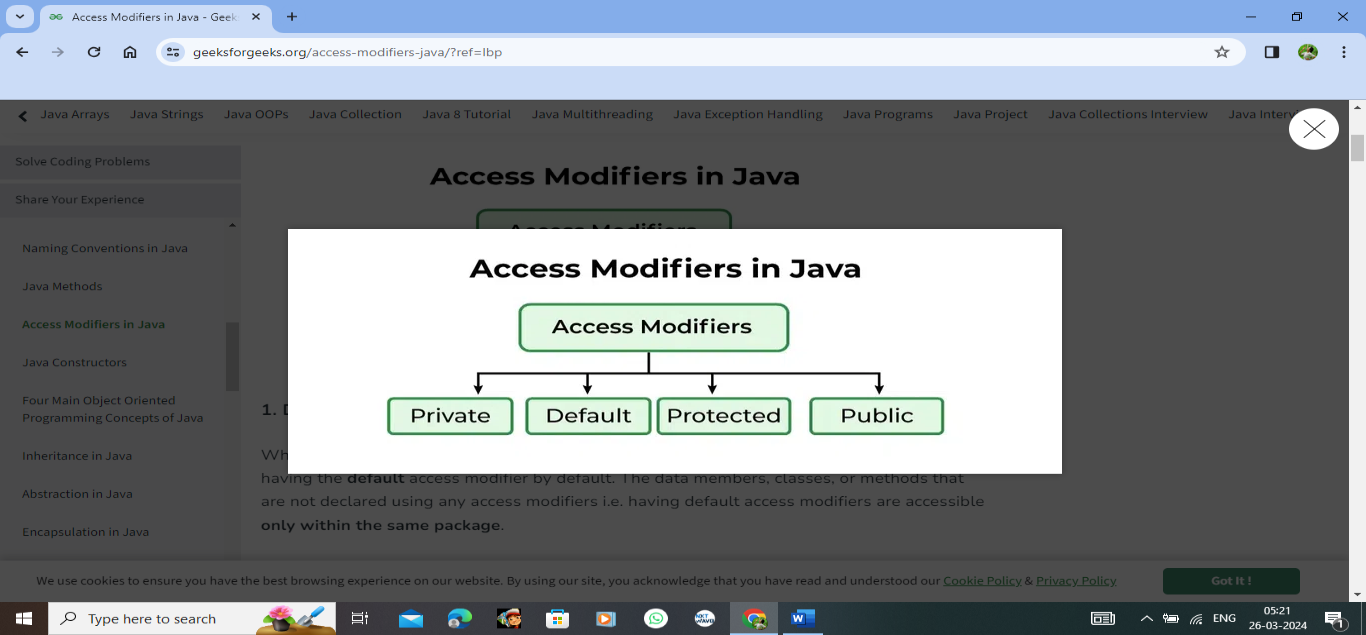
**Access Modifiers:**

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Types:

There are four types of access modifiers available in Java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



**Memory Allocation in Java:**

**In Java, memory allocation is managed by the Java Virtual Machine (JVM) and is handled dynamically during runtime.**

**1.Heap Memory: In Java, objects are allocated memory on the heap. The heap is a region of memory that is shared among all threads running inside the JVM and is where objects are allocated. When you create an object using the `new` keyword, memory is allocated for that object on the heap.**

**2.Stack Memory: Each thread in a Java application has its own stack memory. Stack memory is used for storing local variables and references to objects. When a method is invoked, a new stack frame is created for that method call, which contains information about the method's local variables and its partial execution state. When the method returns, its stack frame is removed from the stack.**

**3.Garbage Collection: Java has automatic garbage collection, which means that the JVM automatically reclaims memory that is no longer in use by objects. When an object is no longer referenced by any part of the program, it becomes eligible for garbage collection. The JVM periodically runs a garbage collector to reclaim memory occupied by these unreferenced objects.**

**4.Memory Management in the JVM: The JVM manages memory allocation and deallocation using various memory management techniques, such as generational garbage collection, which divides the heap into multiple generations (young generation, old generation, etc.) and applies different garbage collection algorithms to each generation based on their characteristics.**

**5. Explicit Memory Management: Unlike languages like C or C++, Java does not allow direct memory manipulation through pointers. Instead, memory allocation and deallocation are handled implicitly by the JVM. This helps prevent common memory-related errors like buffer overflows and memory leaks.**

Classes in Java:

In Java, classes are the fundamental building blocks of object-oriented programming. A class is a blueprint or template for creating objects.

1.Definition: A class is defined using the `class` keyword followed by the class name. It typically contains data members (fields) and methods.

Example:

public class MyClass {

// Fields (data members)

private int myField;

// Methods

public void myMethod() {

// Method implementation

}

}

2.Objects: An object is an instance of a class. You create objects using the `new` keyword followed by the class name and optional constructor arguments.

Example:

MyClass obj = new MyClass();

3.Encapsulation: Classes support encapsulation, which means that they encapsulate data (fields) and behavior (methods) together. Access to the data is controlled through access modifiers such as `public`, `private`, `protected`, and package-private.

4.Inheritance: Java supports inheritance, allowing classes to inherit fields and methods from other classes. This facilitates code reuse and the creation of class hierarchies.

Example:

public class SubClass extends MyClass {

// Additional fields and methods

}

5.Polymorphism: Java supports polymorphism, which means that a reference variable of a superclass type can refer to an object of a subclass type. This allows methods to be invoked dynamically at runtime based on the actual object type.

6.Constructor: A constructor is a special method used for initializing objects. It has the same name as the class and is invoked using the `new` keyword when an object is created.

Example:

public class MyClass {

private int myField;

// Constructor

public MyClass(int initialValue) {

myField = initialValue;

}

}

7.Methods: Methods define the behavior of a class. They can manipulate the class's fields and perform various operations.

8.Static Members: A class can have static fields and methods, which belong to the class itself rather than to individual objects. They are accessed using the class name.

Example:

public class MyClass {

private static int staticField;

public static void staticMethod() {

// Static method implementation

}

}

9.Access Modifiers: Java provides access modifiers such as `public`, `private`, `protected`, and package-private (`no modifier`) to control the visibility of fields, methods, and constructors.

10.Classes form the basis of Java's object-oriented programming paradigm, providing a mechanism for modeling real-world entities and organizing code into reusable and maintainable components.

Packages in Java:

In Java, a package is a way to organize classes into namespaces for better management and modularization of code. Here are some key points about packages in Java:

1. Definition: A package is a directory that contains a collection of related Java classes and interfaces. It provides a way to group related classes and avoid naming conflicts.

2.Package Declaration: At the beginning of each Java source file, there can be a package declaration that specifies which package the file belongs to. This declaration must be the first non-comment statement in the file.

Example:

package com.example.myapp;

3.Package Structure: Packages can be organized hierarchically, with sub-packages containing further classes and sub-packages. For example, the package `com.example.myapp` might contain sub-packages like `util`, `models`, and `controllers`.

4. Import Statement: To use classes from another package, you need to import them using the `import` statement. This informs the compiler about the classes that your code depends on.

Example:

import com.example.myapp.util.UtilityClass;

5. Default Package: If no package is specified in a source file, the classes in that file belong to the default package. However, it's generally considered a good practice to organize classes into packages to avoid naming conflicts and to provide better code organization.

6. Access Modifiers: Classes, interfaces, constructors, methods, and fields can have access modifiers (`public`, `protected`, `private`, or no modifier). These modifiers control the visibility of the class members within the same package or across different packages.

7.Classpath: Java uses the classpath to locate compiled classes and resources. When running Java programs, you need to ensure that the classpath includes the directories and JAR files containing the packages and classes your program depends on.

8.Java Standard Library: Java provides a vast standard library divided into packages, such as `java.lang`, `java.util`, `java.io`, etc. These packages contain classes and interfaces for common programming tasks like string manipulation, collections, file I/O, networking, etc.

9. Packages are essential for organizing and managing large Java projects. They promote code reusability, modularity, and maintainability by providing a structured way to group related classes and avoid naming conflicts.

Collection Framework in Java:

The Java Collections Framework is a set of classes and interfaces that provide reusable data structures to store and manipulate groups of objects. It offers high-performance implementations of commonly used data structures and algorithms. Here are the key components of the Java Collections Framework:

1. Interfaces:

- Collection: The root interface in the collection hierarchy. It defines basic operations such as adding, removing, and querying elements. Subinterfaces include List, Set, Queue, and Deque.

- List: An ordered collection (sometimes called a sequence). Allows duplicate elements and maintains insertion order. Implementations include ArrayList, LinkedList, and Vector.

- Set: A collection that does not allow duplicate elements. Implementations include HashSet, TreeSet, and LinkedHashSet.

- Queue: A collection used to hold elements prior to processing. Follows the FIFO (First-In-First-Out) order. Implementations include LinkedList, PriorityQueue, and ArrayDeque.

- Deque: A double-ended queue that supports insertion and removal of elements at both ends. Implementations include ArrayDeque and LinkedList.

- Map: An object that maps keys to values. A key uniquely identifies a value. Implementations include HashMap, TreeMap, LinkedHashMap, and ConcurrentHashMap.

2. Classes:

- ArrayList: Resizable array implementation of the List interface. Provides dynamic resizing and fast random access.

- LinkedList: Doubly linked list implementation of the List interface. Provides fast insertion and deletion at any position.

- HashSet: Hash table-based implementation of the Set interface. Provides constant-time performance for the basic operations.

- TreeSet: Red-black tree-based implementation of the Set interface. Maintains elements in sorted order.

- HashMap: Hash table-based implementation of the Map interface. Provides constant-time performance for the basic operations.

- TreeMap: Red-black tree-based implementation of the Map interface. Maintains key-value pairs in sorted order.

- LinkedHashMap: Hash table and linked list-based implementation of the Map interface. Maintains insertion order.

-PriorityQueue: Heap-based implementation of the Queue interface. Maintains elements in natural order or according to a specified comparator.

-ArrayDeque: Resizable-array implementation of the Deque interface. Provides dynamic resizing and fast insertion and removal at both ends.

3. Utilities:

- Collections: Contains static methods that operate on or return collections. Includes methods for sorting, searching, shuffling, etc.

- Arrays: Contains static methods for manipulating arrays, such as sorting and searching.

Java Collections Framework provides a unified architecture for representing and manipulating collections of objects. It promotes code reuse, simplifies programming, and improves performance by offering efficient implementations of common data structures and algorithms.

Lists of Java:

ArrayList: Resizable-array implementation of the List interface, allowing dynamic resizing of the array as elements are added or removed.

Syntax:

ArrayList<Type> listName = new ArrayList<>();

Example:

ArrayList<String> fruits = new ArrayList<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

Vector Class:

Similar to ArrayList but synchronized, making it thread-safe but potentially slower in concurrent environments.

Syntax:

import java.util.Vector;

Vector<DataType> vectorName = new Vector<>();

Vector<DataType> vectorName = new Vector<>(initialCapacity);

Vector<DataType> vectorName = new Vector<>(initialCapacity, capacityIncrement);

Example:

import java.util.Vector;

public class VectorExample {

public static void main(String[] args) {

// Create a Vector

Vector<String> vector = new Vector<>();

// Adding elements to the Vector

vector.add("Java");

vector.add("Python");

vector.add("C++");

// Accessing elements by index

System.out.println("Element at index 0: " + vector.get(0));

System.out.println("Element at index 1: " + vector.get(1));

// Removing an element

vector.remove("Python");

System.out.println("After removing 'Python': " + vector);

// Checking if the Vector contains an element

System.out.println("Contains 'C++'? " + vector.contains("C++"));

// Getting the size of the Vector

System.out.println("Size of Vector: " + vector.size());

// Iterating through the Vector

System.out.println("Iterating through the Vector:");

for (String element : vector) {

System.out.println(element);

}

// Clearing the Vector

vector.clear();

System.out.println("Vector after clearing: " + vector);

}

}

Stack Class:

Subclass of Vector, implements a last-in, first-out (LIFO) stack of objects.

Syntax:

import java.util.Stack; Stack<DataType> stackName = new Stack<>();

LinkedList:

Doubly-linked list implementation of the List and Deque interfaces, providing efficient insertion, deletion, and traversal.

Syntax:

import java.util.LinkedList; LinkedList<DataType> linkedListName = new LinkedList<>();

AbstractList:

Abstract base class for List implementations, providing skeletal implementations for common List operations.

Syntax:

import java.util.AbstractList; AbstractList<DataType> abstractListName = new AbstractList<>();

AbstractSequentialList:

Abstract base class for sequential access List implementations, extending AbstractList to support efficient sequential access.

Syntax:

import java.util.AbstractSequentialList; AbstractSequentialList<DataType> abstractSeqListName = new AbstractSequentialList<>();

CopyOnWriteArrayList:

Thread-safe variant of ArrayList, ensuring thread safety by creating a fresh copy of the underlying array on modification operations.

Syntax:

import java.util.concurrent.CopyOnWriteArrayList; CopyOnWriteArrayList<DataType> copyOnWriteArrayListName = new CopyOnWriteArrayList<>();

Custom ArrayList:

Implementing a custom ArrayList involves creating a class that encapsulates an array, provides methods to add, remove, and access elements, and handles resizing as needed, following the principles of dynamic arrays.

Synatx:

import java.util.ArrayList; ArrayList<DataType> customArrayListName = new ArrayList<>();

Abstract Queue:

In Java, `AbstractQueue` is an abstract class defined in the `java.util` package that provides a skeletal implementation of the `Queue` interface. It extends the `AbstractCollection` class and implements some of the methods of the `Queue` interface. This class is intended to be subclassed to implement custom queue data structures.

Syntax:

import java.util.AbstractQueue; AbstractQueue<DataType> abstractQueueName = new AbstractQueue<>();

Array Blocking Queue:

ArrayBlockingQueue is a blocking queue implementation in Java, available in the **java.util.concurrent** package. It represents a bounded blocking queue backed by an array.

Here are some key points about ArrayBlockingQueue:

**Bounded Capacity**: It has a fixed capacity specified at the time of creation, meaning it can only hold a certain number of elements.

**Blocking Operations**: When trying to add an element to a full queue or remove an element from an empty queue, operations will block until there is space to add elements or until an element becomes available for removal.

**FIFO Order**: It maintains the order of elements based on the First-In-First-Out (FIFO) principle.

**Thread-Safe**: It is thread-safe, meaning it can be safely accessed and modified by multiple threads without additional synchronization.

Syntax:

import java.util.concurrent.ArrayBlockingQueue;

ArrayBlockingQueue<DataType> queue = new ArrayBlockingQueue<>(capacity);

Concurrent Linked Queue:

In Java, **ConcurrentLinkedQueue** is a concurrent, unbounded queue implementation available in the **java.util.concurrent** package. It's an implementation of the **Queue** interface and follows a non-blocking algorithm for its operations, making it suitable for concurrent access from multiple threads.

Here are some key features of **ConcurrentLinkedQueue**:

1. **Unbounded**: Unlike **ArrayBlockingQueue**, **ConcurrentLinkedQueue** does not have a fixed capacity and can dynamically grow to accommodate elements.
2. **Non-Blocking**: Operations such as **add**, **offer**, **poll**, **peek**, etc., are non-blocking, meaning they do not involve locking or blocking threads.
3. **FIFO Order**: It maintains the order of elements based on the First-In-First-Out (FIFO) principle.
4. **Thread-Safe**: It is designed to be thread-safe and suitable for concurrent access from multiple threads without additional synchronization.
5. **Weakly Consistent Iterators**: Iterators returned by **ConcurrentLinkedQueue** provide weakly consistent iteration, meaning they reflect the state of the queue at the time of their creation and may not reflect subsequent changes made by other threads.

Syntax:

import java.util.concurrent.ConcurrentLinkedQueue;

ConcurrentLinkedQueue<DataType> queue = new ConcurrentLinkedQueue<>();

LinkedBlockingQueue:

An optionally bounded blocking queue backed by linked nodes.Blocking operations like **put()** and **take()** are supported.Suitable for producer-consumer scenarios with potentially unbounded queues.

LinkedTransferQueue:

An unbounded transfer queue based on linked nodes.Supports both FIFO and LIFO processing modes.Provides efficient handoff between producers and consumers.

PriorityBlockingQueue:

An unbounded blocking queue that uses a priority heap.Elements are ordered according to their natural ordering or a specified comparator.Suitable for scenarios requiring ordering based on element priority.

Deque in Java (Double Ended Queue):

A linear collection that supports insertion and removal at both ends.Common implementations include **ArrayDeque** and **LinkedList**.

ArrayDeque:

Resizable array-based implementation of the Deque interface.Offers constant amortized time for add/remove operations at both ends.

ConcurrentLinkedDeque:

An unbounded concurrent double-ended queue based on linked nodes.Supports thread-safe insertion and removal operations at both ends.

LinkedBlockingDeque:

An optionally bounded blocking double-ended queue backed by linked nodes.Supports blocking operations like **putFirst()**, **putLast()**, **takeFirst()**, and **takeLast()**.

PriorityQueue in Java:

An unbounded priority queue based on a priority heap.Elements are ordered according to their natural ordering or a specified comparator.

EnumMap:

* + Specialized implementation of **Map** for use with enum type keys.
  + Provides high performance and type safety when using enum keys.

HashMap:

* + Implements **Map** interface using a hash table.
  + Offers constant-time performance for most operations.
  + Not synchronized, so not thread-safe by default.

WeakHashMap:

* + An implementation of **Map** with weak keys.
  + Allows keys to be garbage collected when not referenced elsewhere.

LinkedHashMap:

* + Extends **HashMap** with predictable iteration order.
  + Maintains a doubly-linked list of entries to preserve insertion order.

IdentityHashMap:

* + Implements **Map** with reference-equality in place of object-equality.
  + Suitable for specialized use cases where reference equality is needed.

ConcurrentHashMap:

* + Concurrent, thread-safe implementation of **Map**.
  + Offers high concurrency for read and write operations.

Dictionary:

* + Abstract class representing a key-value store.
  + Superseded by the **Map** interface.

HashTable:

* + Legacy implementation of **Map** that is synchronized.
  + Generally slower than **ConcurrentHashMap**.

SortedMap:

* + Interface extending **Map** to provide a sorted view of key-value pairs.

TreeMap:

* + Implements **SortedMap** using a Red-Black tree.
  + Maintains keys in sorted order.

Stack:

* + Legacy class representing a last-in, first-out (LIFO) stack of objects.

Vector:

* + Legacy class similar to **ArrayList** but synchronized.
  + Generally slower than **ArrayList** due to synchronization.

AbstractSet:

* + Abstract base class for set implementations.
  + Provides skeletal implementations of the **Set** interface.

EnumSet:

* + Specialized implementation of **Set** for use with enum type elements.
  + Backed by a bit vector for memory-efficient representation.

HashSet:

* + Implements **Set** using a hash table.
  + Offers constant-time performance for most operations.

TreeSet:

* + Implements **SortedSet** using a Red-Black tree.
  + Maintains elements in sorted order.

SortedSet:

* + Interface extending **Set** to provide a sorted view of elements.

LinkedHashSet:

* + Extends **HashSet** with predictable iteration order.
  + Maintains a doubly-linked list of entries to preserve insertion order.

NavigableSet:

* + Interface extending **SortedSet** with navigation methods.
  + Allows navigation of elements based on their order.

ConcurrentSkipListSet:

* + Concurrent, thread-safe implementation of **NavigableSet**.
  + Provides high concurrency for read and write operations.

CopyOnWriteArraySet:

* + Concurrent, thread-safe implementation of **Set**.
  + Maintains a copy of the underlying array on modification operations.

Exception Handling:

In Java, exceptions are unexpected events that occur during the execution of a program, disrupting the normal flow of instructions. They can occur due to various reasons such as invalid input, hardware failure, or programming errors. Java provides robust mechanisms for handling exceptions to prevent program crashes and ensure graceful error recovery.

Types of Exceptions:

1.Checked Exceptions: These are exceptions that are checked at compile-time. They are subclasses of `Exception` but not subclasses of `RuntimeException`. Examples include `IOException`, `SQLException`, etc.

2.Unchecked Exceptions (Runtime Exceptions):These are exceptions that are not checked at compile-time. They are subclasses of `RuntimeException`. Examples include `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, etc.

Difference between Checked and Unchecked Exceptions:

- Checked exceptions must be either caught or declared in the method signature using the `throws` keyword, while unchecked exceptions do not need to be explicitly handled.

- Checked exceptions are generally used for conditions that are outside the control of the program (e.g., I/O errors), while unchecked exceptions typically indicate programming errors (e.g., dividing by zero).

- Checked exceptions help in writing robust code by forcing developers to handle exceptional conditions explicitly.

Try, Catch, Finally, Throw, and Throws:

- try: It defines a block of code in which exceptions may occur.

- catch:It is used to handle the exceptions that occur in the try block. Multiple catch blocks can be used to handle different types of exceptions.

- finally:It defines a block of code that is always executed, regardless of whether an exception occurs or not. It is typically used for cleanup tasks such as closing resources.

- throw: It is used to explicitly throw an exception within a method.

- throws: It is used in method signatures to declare that the method may throw certain types of exceptions.

Flow control in Try-catch block:

- When an exception occurs within the try block, the control flow immediately transfers to the corresponding catch block.

- If no exception occurs, the catch block is skipped, and the code after the try-catch block is executed.

- The finally block, if present, is executed after the try-catch block, regardless of whether an exception occurred or not.

Throw vs Throws:

- `throw` is used to explicitly throw an exception within a method.

- `throws` is used in method signatures to declare that the method may throw certain types of exceptions.

Final vs Finally vs Finalize:

- final:It is a keyword used to declare constants, prevent method overriding, or prevent inheritance.

- finally: It is a block of code used in exception handling to ensure that certain code is always executed, regardless of whether an exception occurs or not.

- finalize: It is a method in the `Object` class that is called by the garbage collector before reclaiming an object's memory.

User-defined Custom Exception:

You can create custom exceptions by extending the `Exception` class or one of its subclasses. This allows you to define your own exception types tailored to specific scenarios in your application.

Chained Exceptions:

Chained exceptions allow you to associate one exception with another. This is useful when you want to provide more information about the cause of an exception. You can chain exceptions using the constructor of an exception class.

Null Pointer Exceptions:

A `NullPointerException` occurs when you attempt to access or call a method on an object that is `null`. It is one of the most common runtime exceptions in Java and typically indicates a programming error.

Exception Handling with Method Overriding:

When overriding a method that declares checked exceptions in its superclass, the overriding method can:

- Declare the same exceptions.

- Declare subtypes of the exceptions declared in the superclass method.

- Choose not to declare any exceptions (only applicable to unchecked exceptions).

Multithreading in Java:

Multithreading in Java allows multiple tasks to execute concurrently within a single program. This concurrency can improve performance and responsiveness, especially in applications that involve heavy I/O or CPU-bound tasks.

Introduction to Multithreading in Java:

Multithreading allows multiple threads to exist within the context of a single process, enabling concurrent execution of tasks. Each thread has its own stack and runs independently, allowing for parallelism and asynchronous behavior.

Lifecycle and Stages of a Thread:

1. New: When a thread is created but not yet started.

2. Runnable: After calling the `start()` method, the thread enters the runnable state, waiting for the scheduler to allocate processor time.

3. Blocked/Waiting: When a thread is waiting for some condition to occur, such as I/O operations or waiting for a lock.

4. Timed Waiting: Similar to the blocked state, but with a specified waiting time.

5. Terminated: When a thread completes its execution or is stopped forcefully.

Thread Priority in Java:

Thread priority is an integer value that determines the scheduling priority of a thread. Higher priority threads are scheduled to run before lower priority threads, but this behavior is not guaranteed across different platforms and JVM implementations.

Main Thread in Java:

The main thread is the entry point of every Java application. It is created by the JVM to execute the `main()` method of the main class.

Thread Class and Runnable Interface:

- Thread Class: The `Thread` class in Java represents a thread of execution. You can extend this class to create your own threads.

- Runnable Interface: The `Runnable` interface defines a single method, `run()`, that contains the code executed by the thread. It can be implemented by any class whose instances are intended to be executed by a thread.

Naming a Thread:

You can name a thread by either passing the desired name as an argument to the `Thread` constructor or by calling the `setName()` method on the thread object.

start() Method in Thread:

The `start()` method is used to start the execution of a thread. It allocates necessary resources and calls the `run()` method of the thread.

Difference between run() and start() Method:

- run(): Contains the code that defines the task to be executed by the thread. You should not call this method directly to start a thread.

- start(): Initiates the execution of the thread, including calling the `run()` method. It allocates resources and starts the thread's execution.

sleep() Method:

The `sleep()` method is a static method of the `Thread` class used to pause the execution of the current thread for a specified amount of time.

Daemon Thread:

A daemon thread is a low-priority thread that runs in the background and provides services to other threads. The JVM exits when all non-daemon threads have finished executing.

Thread Pool in Java:

A thread pool is a collection of pre-initialized threads that are available to perform tasks. It helps in managing and reusing threads efficiently, reducing the overhead of thread creation.

Thread Group in Java:

A thread group is a way of organizing threads into a hierarchical structure. It provides a convenient way to manage and manipulate multiple threads as a single unit.

Thread Safety in Java:

Thread safety refers to ensuring that shared resources are accessed in a safe and synchronized manner to prevent data corruption and race conditions in a multithreaded environment.

ShutdownHook:

A shutdown hook is a thread registered with the JVM that gets executed when the JVM is shutting down. It allows performing cleanup tasks or releasing resources before the application exits.

Synchronization in Java:

Java Synchronization:

Java synchronization is a technique used to control access to shared resources by multiple threads to prevent data corruption and ensure consistency in a multithreaded environment.

Importance of Thread Synchronization in Java:

Thread synchronization is crucial in Java to avoid race conditions, where multiple threads try to access and modify shared resources simultaneously, leading to unpredictable behavior and data corruption. Synchronization ensures that only one thread can access a synchronized block or method at a time, maintaining data integrity.

Method and Block Synchronization in Java:

1. Method Synchronization: In Java, you can use the `synchronized` keyword to synchronize entire methods. When a thread invokes a synchronized method, it acquires the intrinsic lock associated with the object on which the method is being called.

```java

public synchronized void synchronizedMethod() {

// Synchronized method body

}

```

2. Block Synchronization: Alternatively, you can use synchronized blocks to synchronize specific sections of code. This allows for finer-grained control over synchronization.

```java

synchronized (object) {

// Synchronized block body

}

```

Local Frameworks vs Thread Synchronization:

Local frameworks typically refer to libraries or frameworks that provide concurrency utilities specific to a particular application or domain. Thread synchronization, on the other hand, is a language-level feature provided by Java to manage concurrent access to shared resources across all applications.

Difference between Atomic, Volatile, and Synchronized in Java:

- Atomic: Classes in the `java.util.concurrent.atomic` package provide atomic operations on underlying variables without the need for synchronization. Examples include `AtomicInteger`, `AtomicLong`, etc.

- Volatile: The `volatile` keyword ensures that changes to a variable are immediately visible to other threads. However, it does not provide atomicity or synchronization like `synchronized` or `Atomic` classes.

- Synchronized: Synchronization in Java ensures that only one thread can access a synchronized block or method at a time. It provides both mutual exclusion and visibility guarantees, ensuring thread safety.

Deadlock in Multithreading:

Deadlock occurs when two or more threads are blocked indefinitely, each waiting for a resource that the other thread holds. This situation leads to a stalemate where neither thread can proceed, causing the entire application to hang.

Deadlock Prevention and Avoidance:

Deadlock prevention techniques include:

- Avoiding circular waits by imposing a total ordering of resources.

- Using timeouts and resource allocation hierarchies.

Deadlock avoidance involves analyzing and predicting potential deadlocks at runtime and taking preventive measures to avoid them.

Difference between Lock and Monitor in Concurrency:

- Lock:Locks are explicit objects provided by the `java.util.concurrent.locks` package for exclusive access to shared resources. They offer more flexibility and control compared to intrinsic locks provided by Java synchronization.

- Monitor:Monitors are higher-level synchronization constructs provided by Java, where synchronized methods and blocks use monitors internally. Monitors provide mutual exclusion and coordination among threads.

Reentrant Lock:

A reentrant lock is a type of lock provided by the `java.util.concurrent.locks` package that allows a thread to acquire the same lock multiple times. This feature enables reentrant synchronization, where a thread can enter synchronized blocks it already holds the lock for, without blocking itself.

File Handling in Java:

In Java, the `File` class is used to represent files and directories in the file system. It provides methods for creating, reading, writing, and deleting files, as well as for checking file permissions.

1. Creating Files:

You can create a new file using the `createNewFile()` method of the `File` class.

```java

File file = new File("example.txt");

try {

if (file.createNewFile()) {

System.out.println("File created successfully.");

} else {

System.out.println("File already exists.");

}

} catch (IOException e) {

System.err.println("Error creating file: " + e.getMessage());

}

```

2.Reading Files:

You can read from a file using `FileReader` or other input streams such as `BufferedReader`.

```java

try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage());

}

```

3.Writing to Files:

You can write to a file using `FileWriter` or other output streams such as `BufferedWriter`.

```java

try (BufferedWriter writer = new BufferedWriter(new FileWriter("example.txt"))) {

writer.write("Hello, world!");

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

```

4. Deleting Files:

You can delete a file using the `delete()` method of the `File` class.

```java

File file = new File("example.txt");

if (file.delete()) {

System.out.println("File deleted successfully.");

} else {

System.out.println("Failed to delete the file.");

}

```

5. File Permissions:

You can check file permissions using methods such as `canRead()`, `canWrite()`, and `canExecute()` of the `File` class.

```java

File file = new File("example.txt");

System.out.println("Readable: " + file.canRead());

System.out.println("Writable: " + file.canWrite());

System.out.println("Executable: " + file.canExecute());

```

6. FileReader:

`FileReader` is a class used for reading character files. It reads the content of the file character by character.

7. FileWriter:

`FileWriter` is a class used for writing character files. It writes characters to the file sequentially.

8. FileDescriptor class:

`FileDescriptor` is a unique identifier for an open file or socket. It provides a handle to the underlying machine-specific structure representing an open file.

9. RandomAccessFile class:

`RandomAccessFile` class allows you to read from and write to a file randomly. It supports both reading and writing at the same time, and you can move the file pointer to any position within the file.

Java Regex:

Regular expressions, often abbreviated as regex or regexp, are sequences of characters that define a search pattern. They are widely used in programming for pattern matching, validation, and text manipulation tasks. In Java, regular expressions are supported through the `java.util.regex` package.

1. Writing Regex Expressions:

Regex expressions are written as strings in Java. They consist of normal characters and special characters that have specific meanings. For example, `\\d` represents a digit, `\\w` represents a word character, and `\\s` represents whitespace.

```java

String regex = "\\d+"; // Matches one or more digits

```

2. Matcher Class:

The `Matcher` class is used to perform matching operations on text using patterns. It's created by calling the `matcher()` method on a `Pattern` object.

```java

Pattern pattern = Pattern.compile("\\d+"); // Compile the regex pattern

Matcher matcher = pattern.matcher("123 Java"); // Create a matcher for the input string

```

3. Pattern Class:

The `Pattern` class represents a compiled version of a regular expression. It's used to define regex patterns and provides methods to obtain `Matcher` objects.

```java

Pattern pattern = Pattern.compile("\\d+"); // Compile the regex pattern

```

4. Quantifiers:

Quantifiers specify the number of occurrences of a character or group in a regex pattern. Some common quantifiers include `\*` (zero or more), `+` (one or more), `?` (zero or one), `{n}` (exactly n occurrences), `{n,}` (at least n occurrences), and `{n,m}` (between n and m occurrences).

```java

String regex = "\\d{2,4}"; // Matches 2 to 4 digits

```

5. Character Class:

Character classes represent a set of characters that can match any single character in that set. They are denoted by enclosing characters within square brackets `[]`. For example, `[abc]` matches either 'a', 'b', or 'c'.

```java

String regex = "[aeiou]"; // Matches any vowel

```

These are just some basics to get started with regular expressions in Java. Regular expressions can become quite complex and powerful, allowing you to perform intricate pattern matching operations on text data. The Java regex API provides various methods and features to work with regular expressions efficiently.

Java IO:

Java I/O (Input/Output) is a mechanism provided by Java to perform input and output operations, such as reading from and writing to files, streams, and other I/O devices. It is a fundamental part of Java programming and is used extensively in various applications.

Here's an introduction to Java I/O and some key classes and concepts:

1. Reader Class:

The `Reader` class is an abstract class for reading character streams. It is the superclass of all classes representing readers of character streams.

2. Writer Class:

The `Writer` class is an abstract class for writing character streams. It is the superclass of all classes representing writers of character streams.

3. FileInputStream:

`FileInputStream` is a class used for reading binary data from files. It reads bytes of data from a file in a sequential manner.

4. FileOutputStream:

`FileOutputStream` is a class used for writing binary data to files. It writes bytes of data to a file in a sequential manner.

5. BufferedReader Input Stream:

`BufferedReader` is a class used for reading character input streams. It reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines.

6. BufferedWriter Output Stream:

`BufferedWriter` is a class used for writing character output streams. It writes text to a character-output stream, buffering characters to provide efficient writing of characters, arrays, and lines.

7. BufferedReader vs Scanner:

`BufferedReader` and `Scanner` are both used for reading input from the console or files, but they have different characteristics. `BufferedReader` is generally more efficient for reading character-based input, while `Scanner` is more convenient for parsing different types of input (e.g., integers, doubles) from the input stream.

8. Fast I/O in Java:

In competitive programming or situations where performance is critical, you may need to use fast I/O techniques to improve the efficiency of input/output operations in Java. This typically involves using `BufferedReader` and `BufferedWriter` along with methods like `readLine()` and `println()` for faster input/output operations.

Java I/O provides a powerful and flexible framework for performing input and output operations in Java applications. Understanding and effectively using Java I/O classes and techniques is essential for building robust and efficient Java programs.

Lambda Expressions:

1. Lambda Expressions: Lambda expressions introduce a concise syntax to represent anonymous functions in Java. They allow you to treat functionality as a method argument, or to create instances of functional interfaces. Lambda expressions can significantly reduce the verbosity of your code.

2. Streams API: Streams API is a new abstraction in Java that allows you to process sequences of elements in a functional manner. Streams provide a fluent, functional-style API to manipulate collections of data, enabling developers to perform operations such as filter, map, reduce, and collect in a more expressive way.

3. New Date/Time API: Java 8 introduced a new Date and Time API (`java.time` package) that addresses the shortcomings of the old `java.util.Date` and `java.util.Calendar` classes. The new API provides immutable, thread-safe classes for representing dates, times, instants, durations, and periods, and it offers better support for time zones and formatting.

4. Default Methods: Default methods allow you to add new methods to interfaces in Java without breaking existing implementations. They provide a mechanism for extending interfaces in a backward-compatible way. Default methods have an implementation in the interface itself, which can be overridden by implementing classes.

5. Functional Interfaces: Functional interfaces are interfaces that contain exactly one abstract method. They serve as the basis for lambda expressions and method references in Java. Java provides several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`.

6. Method References: Method references provide a shorthand syntax for creating lambda expressions that call an existing method. They allow you to reuse existing methods as lambda expressions, improving readability and conciseness. Method references are often used when passing methods as arguments to higher-order functions or when implementing functional interfaces.

7. Optional class: The `Optional` class is a container object that may or may not contain a non-null value. It is designed to prevent null pointer exceptions and to provide a more expressive way of handling potentially absent values. `Optional` encourages you to explicitly handle the presence or absence of a value, making your code more robust and self-explanatory.

8. Stream Filter: The `filter` method is one of the intermediate operations provided by the Stream API. It allows you to select elements from a stream based on a specified predicate (a boolean-valued function). Elements that match the predicate are retained in the resulting stream, while those that do not are filtered out.

9. Type Annotations: Type annotations allow you to apply annotations to types, in addition to the traditional locations such as declarations and type uses. Type annotations can be used for a variety of purposes, such as nullability checking, resource management, and program analysis.

10. String Joiner: `StringJoiner` is a utility class introduced in Java 8 that is used to construct a sequence of characters separated by a delimiter. It provides a flexible way to concatenate strings with a specified prefix, suffix, and delimiter. `StringJoiner` is particularly useful when dealing with collections of strings or building comma-separated lists.

Java Date and Time:

In Java, the **Date** class, located in the **java.util** package, was one of the original classes for representing dates and times. However, it has been largely supplanted by the more comprehensive **java.time** API introduced in Java 8.

### Date Class in Java:

The **Date** class represents a specific instant in time, with millisecond precision.

### Methods of the Date class:

Some of the important methods of the **Date** class include:

1. **Date()**: Constructs a new **Date** object representing the current date and time.
2. **Date(long milliseconds)**: Constructs a **Date** object initialized to represent the specified number of milliseconds since the standard base time known as "the epoch," namely January 1, 1970, 00:00:00 GMT.
3. **getTime()**: Returns the number of milliseconds since the epoch represented by this **Date** object.
4. **after(Date when)**: Checks if this date is after the specified date **when**.
5. **before(Date when)**: Checks if this date is before the specified date **when**.
6. **equals(Object obj)**: Compares this date to another object to determine if they represent the same instant in time.

### Java Current Date and time:

To obtain the current date and time in Java, you can use the **java.util.Date** class or the **java.time.LocalDateTime** class from the **java.time** package introduced in Java 8.

Using **java.util.Date**:

Date currentDate = new Date(); // Creates a new Date object representing the current date and time

System.out.println(currentDate);

Using **java.time.LocalDateTime**:

LocalDateTime currentDateTime = LocalDateTime.now(); // Gets the current date and time in the system default time-zone

System.out.println(currentDateTime);

### Compare dates in Java:

To compare dates in Java, you can use methods such as **before()**, **after()**, and **equals()** provided by the **Date** class.

Example:

Date date1 = new Date(); // Current date and time

Date date2 = new Date(System.currentTimeMillis() + 1000); // One second later than current date and time

if (date1.after(date2)) {

System.out.println("date1 is after date2");

} else if (date1.before(date2)) {

System.out.println("date1 is before date2");

} else {

System.out.println("date1 is equal to date2");

}