**Basics of Java**

**Java why?**

**JRE, JVM, JDK**

Step1:

Install Eclipse.

Location:

<https://www.eclipse.org/downloads/packages/release/2025-06/r/eclipse-ide-java-developers>?

Click Download -> Select One time \_> eclipse will download

Extract the folder -> You can find eclipse ready

**(Optional)Run the installer** (eclipse-inst-jre-win64.exe) and select **Eclipse IDE for Java Developers** when prompted.

**(Optional)Install a 64-bit JDK**, if you haven't already, to ensure compatibility.

**Launch Eclipse**, choose or create a workspace, and you're ready to start coding!

**Class 3:**

**Basic Java Program Structure**

Here’s a simple program that prints "Hello World":

class HelloWorld {

public static void main(String[] args) {

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

}

}

**Explanation:**

* class HelloWorld → Defines a class named HelloWorld.
* public static void main(String[] args) → Starting point of the program.
* System.out.println() → Prints output to the screen.

### Program 1: Print "Hello World"

class HelloWorld {

public static void main(String[] args) {

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

}

}

### Program 2: Print your name

class PrintName {

public static void main(String[] args) {

System.out.println("My name is Suresh");

}

}

### Program 3: Print multiple lines

class PrintLines {

public static void main(String[] args) {

System.out.println("Java is powerful.");

System.out.println("It is platform independent.");

System.out.println("Let's start learning!");

}

}

**Java Data Types**

Java is **strongly typed**, meaning every variable must have a type.

**Primitive Data Types**

| **Data Type** | **Size** | **Example** |
| --- | --- | --- |
| byte | 1 byte | 127 |
| short | 2 bytes | 32767 |
| int | 4 bytes | 2147483647 |
| long | 8 bytes | 9223372036854775807 |
| float | 4 bytes | 3.14f |
| double | 8 bytes | 3.14159 |
| char | 2 bytes | 'A' |
| boolean | 1 bit | true/false |

**Non-Primitive Data Types**

* Strings
* Arrays
* Classes
* Interfaces

**Variables**

Variables store data.

### ****Types of Variables:****

1. **Local Variables** – Inside methods.
2. **Instance Variables** – Inside a class, but outside methods.
3. **Static Variables** – Shared among all objects.

**Example:**

int age = 25;

String name = "Suresh";

System.out.println("My name is " + name + " and I am " + age + " years old.");

### Program 1: Display different data types

class DataTypesDemo {

public static void main(String[] args) {

int age = 25;

float height = 5.9f;

char grade = 'A';

boolean isJavaFun = true;

String name = "Suresh";

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

System.out.println("Age: " + age);

System.out.println("Height: " + height);

System.out.println("Grade: " + grade);

System.out.println("Java Fun? " + isJavaFun);

}

}

### Program 2: Swap two numbers

class SwapNumbers {

public static void main(String[] args) {

int a = 10, b = 20;

System.out.println("Before swapping: a = " + a + ", b = " + b);

int temp = a;

a = b;

b = temp;

System.out.println("After swapping: a = " + a + ", b = " + b);

}

}

**Operators**

| **Type** | **Example** |
| --- | --- |
| Arithmetic | + - \* / % |
| Relational | > < >= <= == != |
| Logical | `&& |
| Assignment | = += -= \*= |
| Increment/Decrement | ++ -- |

**Example:**

int a = 5, b = 3;

System.out.println(a + b); // Output: 8

### Program 1: Arithmetic operations

class ArithmeticDemo {

public static void main(String[] args) {

int a = 15, b = 4;

System.out.println("Addition: " + (a + b));

System.out.println("Subtraction: " + (a - b));

System.out.println("Multiplication: " + (a \* b));

System.out.println("Division: " + (a / b));

System.out.println("Modulus: " + (a % b));

}

}

### Program 2: Relational operators

class RelationalDemo {

public static void main(String[] args) {

int a = 10, b = 20;

System.out.println(a == b); // false

System.out.println(a != b); // true

System.out.println(a > b); // false

System.out.println(a < b); // true

}

}

### Program 3: Logical operators

class LogicalDemo {

public static void main(String[] args) {

boolean x = true, y = false;

System.out.println("x && y: " + (x && y));

System.out.println("x || y: " + (x || y));

System.out.println("!x: " + (!x));

}

}

**Control Structures in Java**

Control structures in Java are used to control the **flow of execution** of a program. They allow you to make decisions, repeat tasks, and control how and when certain parts of the code run.

There are **three main types** of control structures in Java:

| **Control Structure Type** | **Examples** |
| --- | --- |
| Conditional Statements | if, if-else, if-else-if, switch |
| Looping Statements | for, while, do-while |
| Jump Statements | break, continue, return |

**1. Conditional (Decision-Making) Statements**

These are used to **make decisions** in the program based on conditions.

**a) if Statement**

Executes a block of code **only if** a given condition is true.

**Syntax:**

if (condition) {

// Code executes if condition is true

}

**Example:**

public class IfExample {

public static void main(String[] args) {

int age = 20;

if (age >= 18) {

System.out.println("You are eligible to vote.");

}

}

}

**b) if-else Statement**

Executes one block if the condition is true, otherwise executes another block.

**Syntax:**

if (condition) {

// Code if condition is true

// Code if condition is true

// Code if condition is true

}

else {

// Code if condition is false

}

**Example:**

public class IfElseExample {

public static void main(String[] args) {

int number = 5;

if (number % 2 == 0) {

System.out.println("Even number");

} else {

System.out.println("Odd number");

}

}

}

**c) if-else-if Ladder (Nested if)**

Used when there are **multiple conditions** to check.

**Syntax:**

if (condition1) {

// Code if condition1 is true

} else if (condition2) {

// Code if condition2 is true

} else {

// Code if none of the above is true

}

**Example:**

public class IfElseIfExample {

public static void main(String[] args) {

int marks = 85;

if (marks >= 90) {

System.out.println("Grade A+");

} else if (marks >= 75) {

System.out.println("Grade A");

} else {

System.out.println("Grade B");

}

int marks2 = 100;

if (marks2 >= 90) {

System.out.println("Grade A+");

} else if (marks2 >= 75) {

System.out.println("Grade A");

} else {

System.out.println("Grade B");

}

int marks3 = 70;

if (marks3 >= 90) {

System.out.println("Grade A+");

} else if (marks3 >= 75) {

System.out.println("Grade A");

} else {

System.out.println("Grade B");

}

}

}

**d) switch Statement**

Used to **replace multiple if-else statements** when comparing a single variable with multiple values.

**Syntax:**

switch (variable) {

case value1:

// Code block

break;

case value2:

// Code block

break;

default:

// Code if none match

}

**Example:**

public class SwitchExample {

public static void main(String[] args) {

int day = 3;

switch (day) {

case 1: System.out.println("Monday"); break;

case 2: System.out.println("Tuesday"); break;

case 3: System.out.println("Wednesday"); break;

default: System.out.println("Invalid day");

}

}

}

**2. Looping (Iteration) Statements**

Used to **repeat a block of code** multiple times.

**a) for Loop**

When the **number of iterations is known**.

**Syntax:**

for (initialization; condition; increment/decrement) {

// Code to be executed

}

**Example:**

public class ForLoopExample {

public static void main(String[] args) {

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

System.out.println("Count: " + i);

}

}

}

Nested loop

Multiplication

public class ForLoopExample {

public static void main(String[] args) {

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

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

System.out.print(i\*j + “\t”);

}

System.out.print(“”);

}

}

Pyramid:

public class ForLoopExample {

public static void main(String[] args) {

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

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

System.out.print(j + “\t”);

}

System.out.print(“”);

}

}

**b) while Loop**

Used when the **number of iterations is not known** in advance.

**Syntax:**

while (condition) {

// Code to be executed

}

**Example:**

public class WhileLoopExample {

public static void main(String[] args) {

int i = 1;

while (i <= 5) {

System.out.println("Count: " + i);

i++;

}

}

}

**c) do-while Loop**

Executes the code block **at least once**, and then repeats while the condition is true.

**Syntax:**

do {

// Code to be executed

} while (condition);

**Example:**

public class DoWhileExample {

public static void main(String[] args) {

int i = 1;

do {

System.out.println("Count: " + i);

i++;

} while (i <= 5);

}

}

**3. Jump Statements**

Used to **alter the normal flow of execution**.

**a) break**

* Exits the loop or switch immediately.

**Example:**

public class BreakExample {

public static void main(String[] args) {

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

if (i == 3) {

break; // Exit loop when i is 3

}

System.out.println(i);

}

}

}

**b) continue**

* **Skips the current iteration** and moves to the next one.

**Example:**

public class ContinueExample {

public static void main(String[] args) {

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

if (i == 3) {

continue; // Skip when i is 3

}

System.out.println(i);

}

}

}

**c) return**

* Exits from a method and optionally returns a value.

**Example:**

public class ReturnExample {

static int sum(int a, int b) {

return a + b;

}

public static void main(String[] args) {

System.out.println("Sum is: " + sum(5, 10));

}

}

**Arrays**

**What is Array?**

Array is the collection of similar data types

**1D:**

Example:

Int a[] = new int[5];

A[] -> reference of an array

Int[6] -> object

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |

A 0 1 2 3 4

|  |
| --- |
|  |

Here we have to assign value,

A[0] = 1;

Every array has length(length is a property, not a method), to print the length,

S.o.p (a.length);

(Or)

Directly to mention the values,

Example:

Int a[] = {1,2,3,4,5,6}

|  |
| --- |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |

A

How to access?

s.o.p(a[0])

s.o.p(a[1])

**Example:**

public class arrayExample {

public static void main(String[] args) {

int a[] = {2,4,6,8,10}

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

System.out.println(a[i]);

}

}

}

public class arrayExample {

public static void main(String[] args) {

int a[] = {2,4,6,8,10}

for (int i = a.length -1; i >=0; i--) {

System.out.println(a[i]);

}

}

}

X -> x takes the value directly, a[i] means it traverse thr index’s.

Only forward direction

public class arrayExample {

public static void main(String[] args) {

int a[] = {2,4,6,8,10}

for (int x : a) {

System.out.println(x);

}

}

}

public class arrayPractise {

public static void main(String[] args) {

int a[] = new int[10];

int b[] = {1,2,3,4,5}

b[2] = 15;

int c[];

c= new int[10];

int[] d = {2,4,6,8,10}

int []e= {3,6,9,}

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

{

System.out.println(a[i]);🡪 default value will be o if not initialized

System.out.println(b[i]);

}

}

}

Program:

1. Find the sum of all the numbers
2. Searching element
3. Maximum element
4. Second largest element
5. copy array A to B

//program 1

public class arrayExample {

public static void main(String[] args) {

Int a[] = {3,9,7,8,12,6,15,5,4,10}

Int sum = 0;

For ( int i=0; i<=a.length; i++}

{

Sum = sum + a[i];

}

s.o.p(“sum is ”+ sum);

}

}

//program2

Int key = 12;

For ( int i=0; i<=a.length; i++}

{

If(key == a[i]}

{

s.o.p(“found the element in index number ”+ i);

System.exit(0);

}

}

s.o.p(“not found in array”);

//program3

Int max = a[0];

For ( int i=0; i<a.length; i++)

{

If (a[i] > max)

{

Max = a[i];

}

}

s.o.p( “max number is” +max)

program4:

Int a[] = {3,9,7,8,12,6,15,5,4,10}

Rough:

Max1 = 3,9,12,15 max2 = 3,3,7,8,9,12

Int max1, max2 ;

Max1=max2 = a[0];

For ( int i=0; i<a.length; i++)

{

If (a[i] > max1)

{

Max2 = max1;

Max1 = a[i];

}

Else if (a[i] > max2)

{

Max2 = a[i];

}

}

s.op.( “second max element is ” +max2)

}

Program5:

public static void main(String[] args)

{

int A[]={8,6,10,9,2,15,7,13,14,11};

int B[]=new int[10];

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

{

B[i]=A[i];

}

for(int x:B)

{

System.out.print(x+",");

}

}

}

**2D**

Collection of arrays or array of arrays

Int a[][] = new int[3][4]

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |

|  |
| --- |
| 0 |
| 1 |
| 2 |
|  |

a

|  |
| --- |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |

Int a[][]= {{1,2,3,4},{2,4,6,8},{3,5,7,9}}

Int a[][] = new int[3][4]

A[] – reference to an array of reference

(2nd)[] – reference to an array

|  |
| --- |
| 0 |
| 1 |
| 2 |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

a

|  |
| --- |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 6 | 8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 5 | 7 | 9 |

Can be also declared as,

Int a[][];= new int[5][5];

Int b[][] = {{1,2,3,4},{2,4,6,8},{3,5,7,9}}

Int c[][] ;

C= new int[5][5];

Int[] d[] = new int[5][5];

Int[] E,F[];

E 🡪 1D array

F 🡪 2Darray

**Program:**

**to print the values in 2D array**

|  |
| --- |
| No. of rows |

For (int i=0; i<a.length;i++) no of columns

{ for (int j= 0; j<a[0].length ;j++)

{

s.o.p(a[i][j]);

}

s.o.p(“\n”);

**to access each loop**

For (int x[] :a) 🡪 x[] reference to array

{ for (int y : x) 🡪 Y takes the element

{

s.o.p(y) }

s.o.p(“\n”);

**Program**:

public class arrayExample {

public static void main(String[] args) {

Int b[][] = {{1,2,3},{2,4,6},{1,3,5}};

For ( int i=0; i<b.length; i++}

{

For ( int j=0; j<b[0].length; j++}

{

s.o.print(b[i][j]+””);

} s.o.pln(“”);

} } }

(OR)

For(int x[]:B)

{

For(int y:x)

{

s.o.print(y);

}

s.o.pln(“”):

} } }

(OR)

**What will happen if I directly print B?**

**s.o.pn(B);**

**Guess the result ?**

**how to access the jagged array 🡪 means of different size**

0 1

|  |
| --- |
| 0 |
| 1 |
| 2 |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 |  |  |

a

0 0 1 2 3

|  |
| --- |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 4 | 6 | 8 |

0 1 2

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 5 | 7 | 9 |

Int a[][];

A = new int [3][];

A[0] = new int[2];

A[1]= new int[4];

A[2] = new int[3];

Example:

public class arrayExample {

public static void main(String[] args) {

Int A[][] = new int[3][];

A[0] = new int[5];

A[1]= new int[3];

A[2] = new int[8];

For ( int i=0; i<A.length; i++}

{

For ( int j=0; j<A[i].length; j++}

{

s.o.print(A[i][j]+””);

} s.o.pln(“”);

} } }

(OR)

public class arrayExample {

public static void main(String[] args) {

Int A[][] = new int[3][];

A[0] = new int[5];

A[1]= new int[3];

A[2] = new int[8];

For ( int x[] : A)

{

For ( int Y : x)

{

s.o.print(y +””);

} s.o.pln(“”);

} } }

Program : Addition of 2 arrays and store it in 3rd array.

public class arrayExample {

public static void main(String[] args) {

Int a[][] = {{3,5,9},{7,6,2},{4,3,5}}

Int b[][]= {{1,5,2},{6,8,4},{3,9,7}}

Int c[][]= new int[3][3]

For ( int i = 0; i<a.length; i++)

{

For ( int j=0; j<a[0].length; j++)

{

C[i][j] = a[i][j]+b[i][j];}

}

For ( x[] :c)

{

For ( y : x)

{

s.o.pln(y+””)

} } }}

Methods: **Methods** are **blocks of code that perform a specific task** and are executed when they are **called or invoked**

**In c, C++ - called as functions but in java we call as Method**

Data

Method 1 Method 2 Method 3 Method n

returnType methodName (Parameter list)

{

// body

//

//

}

returnType -> signature of the method / header of the method (In C or C++ 🡪 prototype)

Example: print the max number

class Test

{

static int max( int x, int y) 🡪 defining method / formal parameter

{

If( x > y)

Return x;

Else

Return y;

}

Public static void main (String args[])

{

Int a=10,b=15,c;

C = max(a,b); 🡪 calling method / actual parameter

s.o.pln(c);

}

}

Main method defining method

a b x y

10 15 10 15

X++ 🡪 value will change to 11 only in defining method

Program:

Static method:

public class MethodPractice {

static int max(int x,int y) //static method

{

if(x>y)

return x;

else

return y;

}

public static void main(String[] args) {

int a=10,b=15;

System.out.println(max(a,b)); 🡪 static method

}

}

Non static method:

public class MethodPractice {

int max(int x, inty) // non static method

{ if(x>y)

return x;

else

return y;

}

public static void main(String[] args) {

int a=10,b=15;

//non static method – create object for class then call the method

MethodPractice mp=new MethodPractice();

System.out.println(mp.max(a,b));

}

}

Difference between actual and formal parameter values:

public class MethodInc {

static void inc(int x)

{

X++;

s.o.pl(“formal parmeter value” +x)

}

public static void main(String[] args) {

int a=10;

inc(a)

s.o.pl(“actual parmeter value” +a)

}

}

**Passing object as Parameter:**

public class Test {

static void update(int a[])

{ a[1] = 25;

}

public static void main(String[] args) {

int a[]={2,4,6,8};

update(a);

System.out.println(a[0])

}

}

Array will be modified to {25,4,6,8} since the reference is passed to change the object, the object itself changing ( different from basic data types like int)

Program for object passing as parameter:

public class MethodPractice1 {

static void change(int A[],int index,int value)

{

A[index]=value;

}

static void change2(int x,int value)

{

x=value;

}

public static void main(String[] args) {

int A[]={2,4,6,8,10};

change(A,2,20);

for(int x:A)

{

System.out.println(x);

}

int x=10;

change2(x,20);

System.out.println("Value of "+x); 🡪 value of x remains as 10 because its not an array/ basic primitive data type.

}

}

**Parameter passing in Java**

Content of actual parameter is passed to formal parameter

Example:

1. A passed to X, B passed to Y -🡪 Z passed to C after addition

X Y Z

Int add (int x, inty) 10 15 25

{int z;

Z=x+y;

Return z; }

P.S.V.M(String args[]) A B C

{ int a=10,b=15,c; 10 15 25

C = add(int a,b)

s.o.pln(C);

} }

1. Name & N both methods will point to same **object** because string is object not primitive data type like int

Void welcome(String n) n

{ S.o.pln(“ Welcome to this “+ n);

}

P.S.V.M(String args[]) name

{ String name = “Victor”; Victor

welcome(name);

} }

Program:

Find prime number:

public class primeNumber

{

static boolean isPrime(int n)

{

for(int i=2;i<n/2;i++)

{

if(n%i==0)

return false;

}

return true;

}

public static void main(String[] args)

{

System.out.println(isPrime(19));

//check 91

} }

**Method Overloading :**

Having more than one method which allows to use same name but there should be difference in parameter list or data types.

1. Overlaod method to calculate areas
2. To reverse a int or array

**Program:**

import java.util.Arrays; // For printing arrays

public class SCMethod2 {

// Validate that the name contains only alphabets and spaces

static boolean validate(String name) {

return name.matches("[a-zA-Z\\s]+");

}

// Validate that the age is between 3 and 15

static boolean validate(int age) {

return age >= 3 && age <= 15;

}

// Reverse an integer

static int reverse(int n) {

int rev = 0;

while (n > 0) {

rev = rev \* 10 + n % 10;

n = n / 10;

}

return rev;

}

// Reverse an array

static int[] reverse(int A[]) {

int B[] = new int[A.length];

for (int i = A.length - 1, j = 0; i >= 0; i--, j++) {

B[j] = A[i];

}

return B;

}

// Calculate area of a circle

static double area(double radius) {

return Math.PI \* radius \* radius;

}

// Calculate area of a rectangle

static double area(double length, double breadth) {

return length \* breadth;

}

// Main method

public static void main(String[] args) {

// 1. Validate Name

String name = "John Doe";

System.out.println("Validating name: " + name);

if (validate(name)) {

System.out.println("Name is valid ✅");

} else {

System.out.println("Name is invalid ❌");

}

System.out.println();

// 2. Validate Age

int age = 10;

System.out.println("Validating age: " + age);

if (validate(age)) {

System.out.println("Age is valid ✅");

} else {

System.out.println("Age is invalid ❌");

}

System.out.println();

// 3. Reverse an Integer

int number = 12345;

int reversedNumber = reverse(number);

System.out.println("Original Number: " + number);

System.out.println("Reversed Number: " + reversedNumber);

System.out.println();

// 4. Reverse an Array

int[] originalArray = {1, 2, 3, 4, 5};

int[] reversedArray = reverse(originalArray);

System.out.println("Original Array: " + Arrays.toString(originalArray));

System.out.println("Reversed Array: " + Arrays.toString(reversedArray));

System.out.println();

// 5. Area of Circle

double radius = 7.0;

double circleArea = area(radius);

System.out.println("Area of Circle with radius " + radius + " = " + circleArea);

System.out.println();

// 6. Area of Rectangle

double length = 5.0, breadth = 3.0;

double rectangleArea = area(length, breadth);

System.out.println("Area of Rectangle with length " + length + " and breadth " + breadth + " = " + rectangleArea);

}

}

**Variable Arguments**

Use single method to handle multiple parameters.

Show()

Show(10)

Show(10,20)

Show(10,20,30)

Show(10,20,30,40)

Show(10,20,30,40,…….,n)

Void show(int …x) 🡪 similar to array (int x[]) but not same.

{

For (a:x)

s.o.pln(a);

}

Also,

Void show(int x, int …y)

Show 🡪 not valid, min one parameter is required because of X

Show(10) 🡪 x=10, y will be null

Show(10,20,30) 🡪 x=10, y = 20,30

Practice 1:

Public class varargsTest

{

Static void show(int …A)

{

For(int x : A)

{

s.o.pln(x);

} }

Public static void main(String[] args)

{

Show();

Show(10,20,30);

Show(new int[]{3,5,7,9,11,13,15});

} }

Practice 2:

If method is taking var parameter, that parameter must be last one.

Public class varargsString

{

Static void showList(int start, String …S)

{

For(int i=0; i<S.length; i++)

{

s.o.pln(i+1+”. “+S[i]);

start++;

} }

Public static void main(String[] args)

{

ShowList(1, “abc”, “def”,”ghi”,”klm”, “nop”);

} }

**Program:**

Find the max number using variable arguments:

public class SCMethod3

{

static int max(int ...A)

{

if(A.length==0)return Integer.MIN\_VALUE;

int max=A[0];

for(int i=1;i<A.length;i++)

if(A[i]>max)max=A[i];

return max;

}

public static void main(String[] args)

{

System.out.println(max());

System.out.println(max(10));

System.out.println(max(10,20));

System.out.println(max(10,20,30));

}

}

**Assignment:**

1. Sum of numbers:

static int sum(int ...A)

{

int s=0;

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

s+=A[i];

return s;

}

1. Calculate Discounts

static double discount(double ...P)

{

double sum=0;

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

sum+=P[i];

if(sum<500) return sum\*0.10;

else if(sum>=500 && sum<1000) return sum\*0.15;

else return sum\*0.20;

}

**Command Line Arguments**

Cd\

dir

Cls

Cd Windows

Dir v\*.\* 🡪 this is argument which will return the V related files.

Cd..

C:\MyJava> Notepad

C:\MyJava> Notepad commandTest.java 🡪 click enter 🡪 it will as for to create new one -> create new.

Enter DIR > it will show the .java is created.

Javac commandTest.java

Java commandTest

1. Create program in commadTest.java

Import java.lang.\*;

Class commandTest{

public static void main(String args[])

{ for (String s : args)

{ s.o.pln(s) }}}

* Javac commandTest.java
* Java commandTest hello world how are you 🡪 enter and observe the result.

1. Dfdf

Import java.lang.\*;

Class commandTest{

public static void main(String args[])

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

{ s.o.pln(args[i] ) }}}

Homework:

public class MySum {

public static void main(String[] args) {

double s=0;

for(String x:args)

{

if(x.matches("[0-9\\.]+"))

s=s+Double.parseDouble(x);

}

System.out.println("Sum is "+s);

}

}

**Recursion**

Method calling itself is called recursive method

public class Recursion

{

static void fun(int n)

{

if(n>0)

{

fun(n-1);

System.out.println(n);

}

}

public static void main(String[] args)

{

fun(3);

}

}

🡪

fun(n-1);

System.out.println(n);

O/P : 1 2 3

🡪

System.out.println(n);

fun(n-1);

O/P : 3 2 1

**OOPS**

**Principles of OOPS:**

Abstraction – Hiding internal details and showing only required features.

Example : Television

Encapsulation - : that refers to the bundling of data (variables) and the methods (functions) that operate on that data into a single unit

Example: Television, capsule, car

Inheritance: one class can use the features of another class – **resuing**

**Specializion –** part of inheritance, enhancing the feature (Ex: car, iphone)

Polymorphism – means "many forms"

**Generalisation** - part of polymorphism, means grouping the similar terms (Ex: java students, sql students etc, TV, Car)

**Classes & Objects:**

An **object** is an **instance** of a class.

A **class** is like a **blueprint** or **template** for creating objects.

Program:

class Student

{

public int roll;

public String name; Properties

public String course;

public int m1,m2,m3;

public int total()

{ return m1+m2+m3;

}

public float average()

{ return (float)total()/3;

}

public char grade()

{ if(average()>=60)

return 'A';

else

return 'B'; }

public String toString() 🡪 no need to call toString as method, we can access this as object \*

{ return "Roll No:"+roll+"\n"+"Name:"+name+"\n"+"Course:"+course+"\n";

} }

public class StudentTest

{ public static void main(String[] args)

{ Student s=new Student();

s.roll=1;

s.name="John";

s.course="CS";

s.m1=70;

s.m2=80;

s.m3=65;

System.out.println("Total :"+s.total());

System.out.println("Average :"+s.average());

System.out.println("Details:\n "+ s );

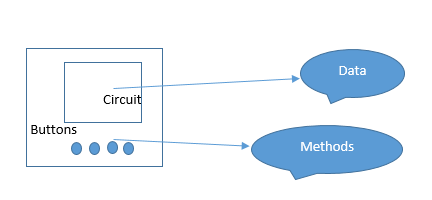
} }

**Data Hiding**

How the data is hidden?

Example:

Television: we operate the TV using the buttons but we do not know the internal operations which are performing. Similarly, methods will access the data but the data is hidden.



Class Rectangle{

Public/private int length;

Public/private int breadth;

Public int area(){

{ return length \* breadth; }

P s v m(String args[])

{

Rectangle r = new Rectangle();

r.length = 10;

r.breadth = 15; } 🡪 this wll work for public (access outside the class, but not for private, can not access outside the class)

so, to overcome this, we can have to use getters(to read) and setters(to write). HOW?

Getters and setters we calls as property methods

**Program:**

class Rectangle

{

private double length;

private double breadth;

public double getLength()

{

return length;

}

public double getBreadth()

{

return breadth;

}

public void setLength(double l)

{

if(l>=0)

length=l;

else

length=0;

}

public void setBreadth(double b)

{

if(b>=0)

breadth=b;

else

breadth=0;

}

public double area()

{

//return length\*breadth;

return getLength()\*getBreadth();

}

public double perimeter()

{

return 2\*(length+breadth);

}

public boolean isSquare()

{

if(length==breadth)

return true;

else

return false;

}

}

public class DataHidingtest {

public static void main(String[] args) {

Rectangle r=new Rectangle();

r.setLength(10.5);//check with negative values.

r.setBreadth(5.5);

System.out.println("Area "+r.area());

System.out.println("Perimeter "+r.perimeter());

System.out.println("Is Square "+r.isSquare());

System.out.println("Length "+r.getLength());

System.out.println("Breadth "+r.getBreadth());

}

}

**Type of Properties:**

1. Read & write – previous program
2. Read – example: rollno, bank accounts
3. Write – example charity account(class1) <🡨 🡪 donor(class2) – very rare case

**Read:**

public class Rectangle {

private double length = 10.0; // Default length value assigned directly

// getLength method

public double getLength() {

return length; }

public static void main(String[] args) {

Rectangle rect = new Rectangle(); // No-argument constructor (implicit)

System.out.println("The length of the rectangle is: " + rect.getLength());

}

}

**Write:**

public class Rectangle {

private double length; // Declare a private variable to store the length

// Method to set the value of length

public void setLength(double len) {

length = len; // Assign the given value to the length variable

System.out.println("Length has been set to: " + length);

}

public static void main(String[] args) {

Rectangle rect = new Rectangle(); // Create a Rectangle object

rect.setLength(12.5); // Set the length using setLength method

}

}

**Constructors**

A constructor **must have the same name as the class**, constructor will not have return type.

**automatically called when an object is created**. It is mainly used to **initialize the object’s state** (assign initial values to instance variables).

Default constructor - Java automatically provides a **default constructor** with **no arguments** and **empty body**

**Example: create the object first, then assign values separately.**

**class Rectangle {**

**int length; int width; }**

**public class Main {**

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

**Rectangle r = new Rectangle();**

**r.length = 10; // setting values manually**

**r.width = 5;**

**System.out.println("Length: " + r.length);**

**System.out.println("Width: " + r.width); } }**

**Parametrized constructor - takes one or more parameters (arguments)** to initialize an object with specific values at the time of its creation.

**Example:** Here, the values 10 and 5 are **passed directly to the constructor** when creating the object.

class Rectangle {

int length; int width;

// Parameterized constructor

Rectangle(int l, int w) {

length = l; width = w; }}

public class Main {

public static void main(String[] args) { // Pass values directly at object creation

Rectangle r = new Rectangle(10, 5);

System.out.println("Length: " + r.length); // 10

System.out.println("Width: " + r.width); // 5 }}

**Why Use Parameterized Constructor?**

1. Cleaner code – No need to set each field manually after object creation.
2. Initialization in one step – Object gets ready-to-use values immediately.
3. Avoids inconsistent state – Ensures required data is provided when the object is created.

**Example: Multiple Objects with Different Data :** Parameterized constructors let you create multiple objects with different initial values easily:

class Student {

String name; int age;

Student(String n, int a) {

name = n; age = a; } }

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Arun", 20);

Student s2 = new Student("Meera", 22);

System.out.println(s1.name + " - " + s1.age); // Arun - 20

System.out.println(s2.name + " - " + s2.age); // Meera - 22

}

}

**Constructor Overloading with Parameters:** You can have multiple constructors with different parameter lists.

class Rectangle {

int length;

int width;

// No-argument constructor

Rectangle() {

length = 0; width = 0;

System.out.println("No-argument constructor called: length = " + length + ", width = " + width); }

// Parameterized constructor

Rectangle(int l, int w) {

length = l; width = w;

System.out.println("Parameterized constructor called: length = " + length + ", width = " + width); }}

public class Main {

public static void main(String[] args) {

Rectangle r1 = new Rectangle(); // calls default constructor

Rectangle r2 = new Rectangle(15, 8); // calls parameterized constructor

// Print final values explicitly

System.out.println("Final values of r1 -> Length: " + r1.length + ", Width: " + r1.width);

System.out.println("Final values of r2 -> Length: " + r2.length + ", Width: " + r2.width); } }

Arrays of Objects:

**Assignment:**

class Subject

{

private String subID;

private String name;

private int maxMarks;

private int marksObtains;

public Subject(String subID,String name,int maxMarks)

{ this.subID=subID; **this** 🡪 It's used to differentiate between instance variables and local variables (parameters) when both have the same name.

this.name=name;

this.maxMarks=maxMarks; }

public String getSubID(){return subID;}

public String getName(){return name;}

public int getMaxMarks(){return maxMarks;}

public int getMarksObtains(){return marksObtains;}

public void setMaxMarks(int mm)

{

maxMarks=mm; }

public void setMarksObtain(int m)

{

marksObtains=m; }

boolean isQualified()

{

return marksObtains>=maxMarks/10\*4; }

public String toString()

{

return"\n SubjectID: "+subID+"\n Name "+name+"\n MarksObtained "+marksObtains; } }

public class SubjectArray {

public static void main(String[] args)

{

Subject subs[]=new Subject[3];

subs[0]=new Subject("s101","DS",100);

subs[1]=new Subject("s102","Algorithms",100);

subs[2]=new Subject("s103","Operating Systems",100);

for(Subject s:subs)

System.out.println(s);

} }

Set value for marksobtained,

public class SCLoops {

public static void main(String[] args) {

Subject subs[] = new Subject[3];

// Create subjects

subs[0] = new Subject("s101", "DS", 100);

subs[1] = new Subject("s102", "Algorithms", 100);

subs[2] = new Subject("s103", "Operating Systems", 100);

// Set marks for each subject

subs[0].setMarksObtain(85); // manually setting marksObtains

subs[1].setMarksObtain(72);

subs[2].setMarksObtain(90);

// Display subject details

for (Subject s : subs) {

System.out.println(s);

System.out.println("Qualified: " + s.isQualified());

System.out.println("------------");

}

}

}

**Inheritance:**

Inheritance: one class can use the features of another class – **resuing**

**Inheriting its properties**

**Generalization** - part of polymorphism, means grouping the similar terms (Ex: java students, sql students etc, TV, Car)

Polymorphism – means "many forms"

Smartphone 🡪 iphone,Samsung,vivo

Vehicle 🡪 car,bike, bus

Students 🡪 java, sql, excel,python

Derivation is from bottom-up

**Specialization –** part of inheritance, enhancing the feature (Ex: car, iphone)

Iphone 🡪 14 🡪15 🡪16 🡪17

Java 🡪 core java 🡪 advanced java

Derivation is from top-down

Interitance is the process of acquiring the features of an existing class into new class

Features -> properties ,method

Example:

Class circle() 🡪 Base class, super class, parent class

{ provate double radius;

Public circle() { radius =0; } 🡪 constructor

Public double area();

Public double perimeter();

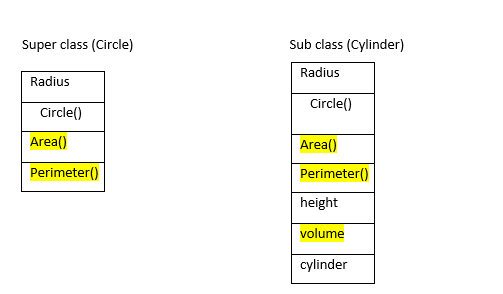
Class cylinder extends circle() { -🡪 derived class, sub class, child class

Private double height;

Public cylinder() ( height = 10.0; }

Public double volume() { }

}



Class Test

{ p.s.v.m ()

{ Circle c1 = new circle();

cylinder c2 = new cylinder(); 🡪 no need to call inheritance methods because by default that will be called.

c1. Area();

c2.are();

c2.volume(); }}

Program 1:

// No constructor – inheritance concept

class Circle

{ public double radius;

public double area() {

return Math.PI \* radius \*radius; }

public double perimeter()

{ return 2\*Math.PI\*radius; }

public double circumference()

{ return perimeter();

} }

class Cylinder extends Circle

{

public double height;

public double volume()

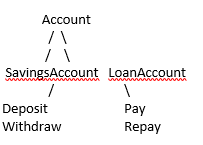
{

return area()\*height;

}

}

Program 2: Example for inheritance 🡪 accounts program



class Account {

private String accNo;

private String name;

private String address;

private String phno;

private String dob;

protected long balance;

public Account(String acc, String n, String add, String phno, String dob) {

accNo = acc;

name = n;

address = add;

this.phno = phno;

this.dob = dob;

balance = 0;

}

public String getAccNo() {

return accNo;

}

public String getName() {

return name;

}

public String getAddress() {

return address;

}

public String getPhno() {

return phno;

}

public String getDOB() {

return dob;

}

public long getBalance() {

return balance;

}

public void setAddress(String add) {

address = add;

}

public void setPhno(String phno) {

this.phno = phno;

}

}

class SavingsAccount extends Account {

public SavingsAccount(String acc, String n, String add, String phno, String dob) {

super(acc, n, add, phno, dob); 🡪 Why super keyword? - in SavingsAccount, you want to **use that constructor** to initialize those fields. Without super(...), Java would try to call a **no-arg constructor** of Account

}

public void deposit(long amt) {

if (amt > 0) {

balance += amt;

System.out.println("Deposited ₹" + amt + " to savings account.");

}

}

public void withdraw(long amt) {

if (amt > 0 && amt <= balance) {

balance -= amt;

System.out.println("Withdrew ₹" + amt + " from savings account.");

} else {

System.out.println("Insufficient balance or invalid amount.");

}

}

}

class LoanAccount extends Account {

public LoanAccount(String acc, String n, String add, String phno, String dob) {

super(acc, n, add, phno, dob);

balance = 100000; // assume loan amount is 1,00,000 by default

}

public void payEMI(long amt) {

if (amt > 0 && amt <= balance) {

balance -= amt;

System.out.println("Paid EMI of ₹" + amt + ". Remaining loan: ₹" + balance);

} else {

System.out.println("Invalid EMI payment.");

}

}

public void repay(long amt) {

if (amt == balance) {

balance = 0;

System.out.println("Loan fully repaid.");

} else {

System.out.println("Repayment amount does not match the remaining balance.");

}

}

}

public class SCInherit {

public static void main(String[] args) {

// Creating Savings Account

SavingsAccount sa = new SavingsAccount("SA1001", "Alice", "Bangalore", "9876543210", "1990-01-01");

sa.deposit(5000);

sa.withdraw(2000);

System.out.println("Savings Balance for " + sa.getName() + ": ₹" + sa.getBalance());

System.out.println();

// Creating Loan Account

LoanAccount la = new LoanAccount("LA2001", "Bob", "Chennai", "9123456780", "1985-05-05");

la.payEMI(20000);

la.repay(80000); // this should repay the rest of the loan

System.out.println("Loan Balance for " + la.getName() + ": ₹" + la.getBalance());

}

}

**Constructors in Inheritance**

Class Parent

{

parent{

s.o.pln (“Parent constructor”); }}

Class Child

{

Child extends Parent{

s.o.pln (“Child constructor”); }}

Class Parent

{

Class Grandchild extends Child

{

Public GrandChild{

s.o.pln (“GrandChild constructor”); }}

public class Inheritconstructor{

{

Grandchild C = new GrandChild(); }}

o/p: Parent Constructor, Child Constructor, GrandChild Constructor.

Why is works like that? 🡪 Example: car, many cars will use the same platform (Base engile,CC,torque etc.., working model can be developed, later it can be transformed to different models – Swift,Swift desire.

**Parameterised Constructors in Inheritance**

class Parent

{

Parent()

{

System.out.println("Non-Param of parent");

}

Parent(int x)

{

System.out.println("Param of parent "+x);

}

}

class Child extends Parent

{

Child()

{

System.out.println("Non-Param of child");

}

Child(int y)

{

System.out.println("Param of child");

}

Child(int x,int y)

{

super(x); 🡪 to initialize the super class method, else parent class method will not be called.

System.out.println("2 param of child :" + x + “, “+y);

}

}

public class SuperConstr {

public static void main(String[] args) {

//Child c=new Child(); 🡺 O/p: Non param of parent,Non Param of child

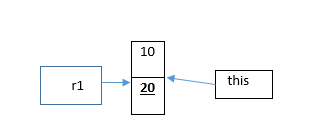
//Child c=new Child(20); 🡺 O/p: Non param of parent, Param of child

Child c=new Child(10,20); 🡺 O/p: param of parent 10, Param of child 20

}

}

**This and super**

**This** 

Class Rectangle{

Int length,breadth;

Rectangle(int length, int breadth){

This.length = length or length = l; if (parameter is (int l, int b))

This.breadth = breadth or breadth = b; if (parameter is (int l, int b)) }

Void display(){

s.o.pln(“length is”+this.length);

s.o.pln(“breadth is”+this. breadth); }}

public class this{ {

p.s.v.m(string args[])

{ Rectangle r1 = new Rectangle(10,20) }}

**Super:** super is reference to the super class

class Rectangle{

int length;

int breadth;

int x=10;

Rectangle(int length,int breadth) {

this.length=length;

this.breadth=breadth;

} }

class Cuboid extends Rectangle {

int height;

int x=20;

Cuboid(int l,int b,int h)

{

super(l,b);

height=h;

}

void display()

{

System.out.println(super.x);

System.out.println(x);

} }

public class ThisSuper

{

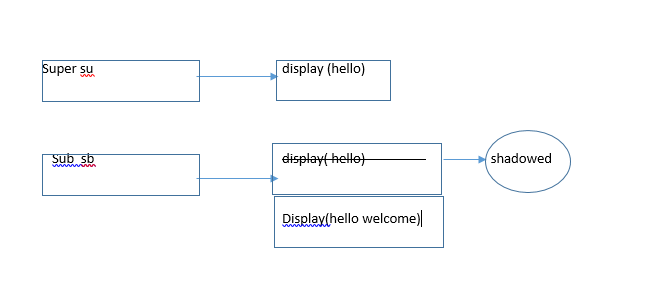
public static void main(String[] args)

{

Cuboid c=new Cuboid(10,5,15);

c.display(); } }

**Method Overriding:**

Redefining the method of super class in subclass.

Example 1:

Class super{

Public void display(){

s.o.pln(“hello”) }

class sub extends super {

Public void display(){

s.o.pln(“hello welcome”) }

**Class Test{**

P.s.v.m(Str ar){

Super su = new super();

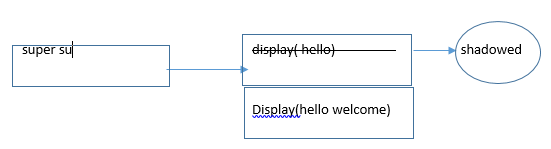
Su.display(); 🡪 hello

Sub sb = new sub();

Sb.display(); 🡪 hello welcome

Example 2:

When a Super class reference holding an object of sub class and overridden method is called Dynamic method dispatch.



Super su = new sub(); 🡨 🡪 not allowed Sub sb = new super();

Su. display(); ? 🡪 hello welcome

Program:

class Car{

public void start(){System.out.println("Car Started");}

public void accelerate(){System.out.println("Car is Accelerated");}

public void changeGear(){System.out.println("Car Gear Changed"); } }

class LuxaryCar extends Car{

public void changeGear(){System.out.println("Automatic Gear");}

public void openRoof(){System.out.println("Sun Roof is Opened"); } }

public class OverridingExample {

public static void main(String[] args) {

LuxaryCar c=new LuxaryCar();

c.start();

c.accelerate();

c.changeGear();

c.openRoof(); } }

**To remember:**

1. Over loading is used for compile time polymorphism
   * because the compiler decides which method to call based on parameter types.
2. Over loading is used for run time polymorphism
   * because the JVM decides which method to call when the program runs.
3. dynamic method dispatch 🡪 super s = new sub()
4. static and final methods cannot override
5. Constructors cannot be overridden.

| **Feature** | **Overloading** | **Overriding** |
| --- | --- | --- |
| Type | Compile-time polymorphism | Runtime polymorphism |
| Location | Same class | Subclass |
| Parameters | Must differ | Must be same |
| Return type | Can differ | Must be same |
| Decided by | Compiler | JVM at runtime |

Program to overload and override:

class Animal {

// Overloaded methods — compile-time polymorphism

void sound() {

System.out.println("Animal makes a sound");

}

void sound(String type) {

System.out.println("Animal sound type: " + type);

}

}

class Dog extends Animal {

// Overriding method — runtime polymorphism

@Override

void sound() {

System.out.println("Dog barks");

}

// Overloaded method (different parameter)

void sound(int times) {

System.out.println("Dog barks " + times + " times");

}

}

public class PolymorphismExample {

public static void main(String[] args) {

Animal a = new Animal();

Animal b = new Dog();

Dog d = new Dog();

System.out.println("--- Overloading (Compile-time) ---");

d.sound(); // Dog barks

d.sound("Loud"); // Animal sound type: Loud (from parent)

d.sound(3); // Dog barks 3 times

System.out.println("\n--- Overriding (Runtime) ---");

a.sound(); // Animal makes a sound

b.sound(); // Dog barks (runtime polymorphism)

}

}

**Abstract Class:**

An **abstract class** in Java is a **class that cannot be instantiated** (you can’t create objects of it) and is declared using the **abstract keyword**.

2 types of classes:

1. Abstract Class: if abstract keyword is used in the class (Ex: abstract Class test {})
2. Concrete class: nothing is used, just starts with the class (Ex: Class test {})

What is the diff?

You cannot create objects for abstract class (not allowed: super s1 = new super())

You can create object for concrete class (allowed : super s1 = new super())

reference can be created in abstract class ( Super s1)

reference can be created in concrete class ( Super s1)

**Abstract class and Abstract Methods:**

**Abstract Class:**

* if class is having one abstract method, then definitely class should be used as abstract class.
* Class can also be used as abstract class without having abstract method as well.

**Abstract Method:**

* abstract method will not have body (no flower brackets, semi colon should be mentioned
* Also called undefined method, concretemethods called as defined methods

Demonstration:

abstract class Super

{

//constructor

public Super() { System.out.println("Super Constructor"); }

// defined method

public void meth1()

{

System.out.println("Meth1 of Super");

}

//abstract method / undefined method

abstract public void meth2();

}

class Sub extends Super

{

@Override

public void meth2()

{

System.out.println("Sub meth2");

}

}

public class AbstractExample

{

public static void main(String[] args)

{

Super s=new Sub();

s.meth1();

s.meth2();

//Also possible

sub sb = new sub();

Super s = sb;

s.meth1()

}

}

Question:

If sub class is inheriting super abstract class, sub class also become abstract sub class ???

**NO** 🡪 sub class will inherit all the abstract super class , also method2 is **overridden** above example.

Example:

1. Hospital
2. KFC

* Abstract classes are used for defining or imposing the standards.
* super class(HOSPITAL) – imposing the standards
* sub class (MY HOSPITAL) – following the standards

1. **Hospital scenario:**

abstract Class hospital

{ 🡪 they don’t have their own hospital, but they give only standards, which can be related that objects cannot be created, but reference(standards here) can be created.

abstract void emergency();

abstract void appointment();

abstract void admit();

abstract void billing(); }

class Myhospital extends hospital {

myhsopital() {

}

void emergency() {}

void appointment() } } }

* as discussed, object cannot be created but reference can be created.
* super class reference can hold the object of sub class

hospital h = new Myhospital()

h.appointment(); 🡪 sub class method will be called.

**Program:**

abstract class Shape

{

abstract public double perimeter();

abstract public double area();

}

class Circle extends Shape

{

double radius;

public double perimeter()

{

return 2\*Math.PI\*radius;

}

public double area()

{

return Math.PI\*radius\*radius;

}

}

class Rectangle extends Shape

{

double length;

double breadth;

public double perimeter()

{

return 2\*(length+breadth);

}

public double area()

{

return length\*breadth;

}

}

public class SCAbstract1

{

public static void main(String[] args)

{

Rectangle r=new Rectangle();

r.length=10;

r.breadth=5;

Shape s=r;

→ This is **upcasting**:  
The reference variable s is of **type Shape**, but it actually **refers to a Rectangle object**.

this is run time polymorphism or dynamic method dispath

System.out.println(s.area());

}

}

Rules:

| **Rule** | **Description** |
| --- | --- |
| 1 | Declared using abstract keyword |
| 2 | Cannot be instantiated (object cannot be created) |
| 3 | Can have abstract and non-abstract methods |
| 4 | If one method is abstract, class must be abstract |
| 5 | Subclass must implement abstract methods else sub class will become abstract |
| 6 | Can have constructors |
| 7 | Can have variables and static methods |
| 8 | Abstract methods cannot be final, static. |
| 9 | Can inherit from another abstract class |

**Interfaces:**

* It defines **what** a class must do — not **how** it does it.
* A collection of abstract methods
* Interfaces are implemented
* Interface is an abstract class of all abstract method. the purpose of achieving interface is polymorphism.
* A class **uses the implements keyword** to provide code for an interface’s methods.
* one interface can extend another interface
* you cannot create object for interface
* interface inherit for one class but can implement more than one interfaces.
* default interface method can be created inside interface.

Demonstration:

interface Test{

void method1();

void method2();

}

class My implements Test {

public void method1(){

s.o.pln (“method1 of class my”);

}

public void method2(){

s.o.pln (“method2 of class my”);

}

public void method3(){

s.o.pln (“method3 of class my”);

}

public class interfacepractise{

p.s.v.m(String{} arg) {

Test t = new Test() } } // not allowed in interface

Test t = new My() // This is allowed – reference to sub class object

t.method1(); 🡪 called

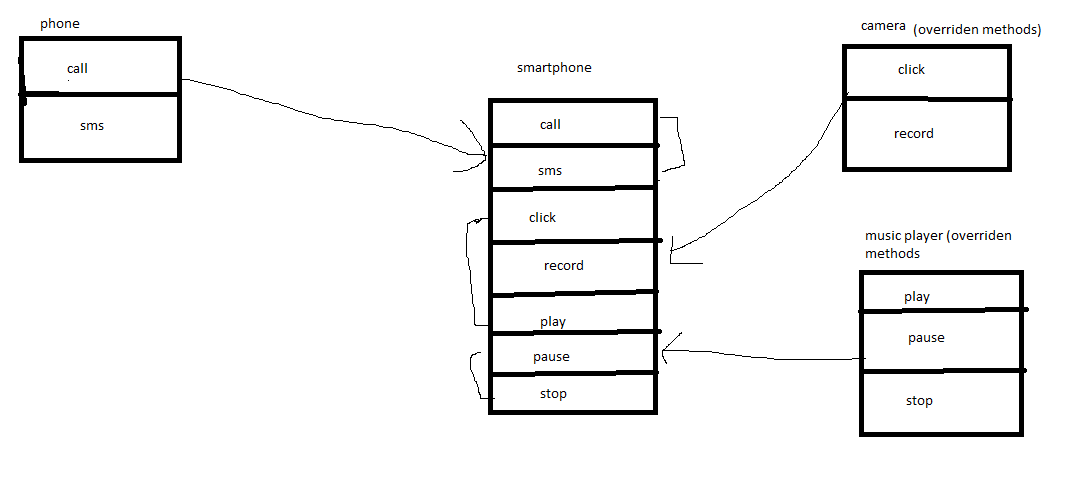
t.method2(); 🡪 called

t.method3(); will not be called because only overriding methods will be called.

} }

* interfaces are meant only for overriding methods
* interfaces achieve run time polymorphism (overriding methods)

Diagram:



**Program1:**

class Phone

{

public void call() { System.out.println("Phone call"); }

public void sms() { System.out.println("Phone sending SMS"); }

}

interface ICamera

{

void click();

void record();

}

interface IMusicPlayer

{

void play();

void stop();

}

class SmartPhone extends Phone implements ICamera,IMusicPlayer

{

public void videoCall() { System.out.println("Smart Phone video calling"); }

public void click() { System.out.println("Smart Phone Clicking Photo"); }

public void record() { System.out.println("Smart Phone recording video"); }

public void play() { System.out.println("Smart Phone playing music"); }

public void stop() { System.out.println("Smart Phone stopped playing music"); }

}

public class InterfaceExample

{

public static void main(String[] args)

{

phone sp=new SmartPhone(); // allowed (phonecall/phone sending dms)

Icamera sp=new SmartPhone(); //allowed

IMusicPlayer sp=new SmartPhone(); //allowed

sp.play();

sp.stop();

}

}

**Program2: same achieved using multiple inheritance via interface**

// Interface 1

interface Camera {

void click();

}

// Interface 2

interface MusicPlayer {

void play();

}

// Class implementing both interfaces

class SmartPhone implements Camera, MusicPlayer {

public void click() {

System.out.println("Clicking photo...");

}

public void play() {

System.out.println("Playing music...");

}

}

// Main class

public class MultipleInheritanceExample {

public static void main(String[] args) {

SmartPhone sp = new SmartPhone();

sp.click(); // From Camera

sp.play(); // From MusicPlayer

}

}

**Program3:**

interface Test

{

final static int X=10;

public abstract void meth1();

public abstract void meth2();

public static void meth3()

{

System.out.println("Meth3 of Test");

}

}

interface Test2 extends Test

{

void meth4();

}

class My implements Test2

{

public void meth1(){System.out.println("Meth1");}

public void meth2(){System.out.println("Meth2");}

public void meth4(){System.out.println("Meth4");}

}

public class InterfacePractice

{

public static void main(String[] args)

{

System.out.println(Test.X);

**//Scenario1**

Test.meth3();

}

}

**//Scenario2**

**Change from public to static**

public static void meth3() 🡪 default void meth3()

change the main class as,

My obj = new My();

obj.meth3(); // ✅ Now allowed (default method call)

**//Scenario3**

private void method3()

{ S.O.Pln( ‘’method3 of test”); }

default void method5(){

method3();

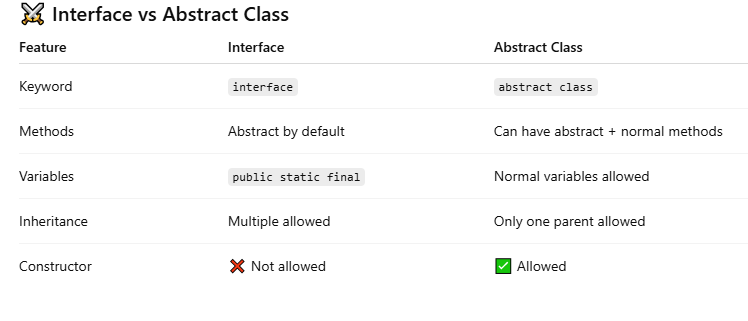
}

class My implements Test2

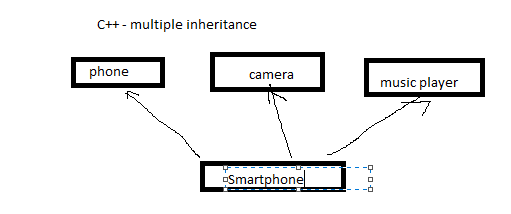
{

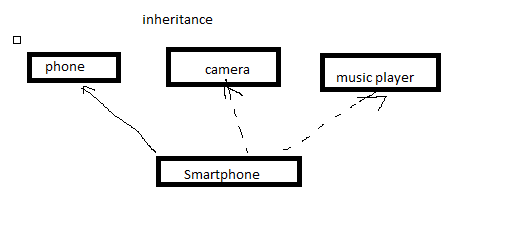
// add this code;

public void meth4(){System.out.println("Meth3");}



**Interface vs multiple interface (C++):**





**Inner class**

We can define a class inside another class, means outer class can have an inner class inside.

In simple term 🡪 class inside another class.

What is the need of inner class?

If you have a class and it is becoming very big and very complex and you want to make it a little simple, anything is repeating in that one, then you can write it in the form of a class itself. Even you can do an interface also. You can define an interface inside a class.

There are different types of inner classes.

1. **Nested inner class:**

class Outer {

// static int x=10;

int x =10;

Inner i=new Inner(); 🡺 object can be created here also

class Inner {

int y=20;

public void innerDisplay() {

System.out.println(x+" "+y); } }

public void outerDisplay() {

// Inner i=new Inner(); 🡺 object can be created here also

i.innerDisplay();

System.out.println(i.y); } }

public class NestedInner {

public static void main(String[] args) {

Outer.Inner oi=new Outer().new Inner(); // Create inner object

oi.innerDisplay();

Outer o = new Outer(); // Create Outer object

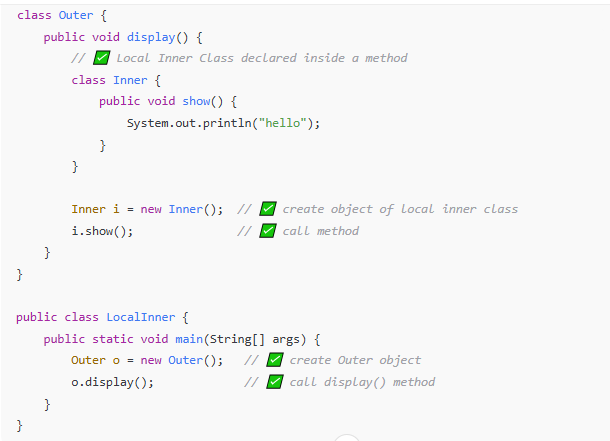
o.outerDisplay(); } }

Inner class can access the members of outer class directly, and outer class cannot access

the members of inner class directly unless it has created an object.

1. Local inner class:

define a class inside a method. outside this method this class is not visible.



1. Anonymous inner class :

can be defined at the time of creation of object itself. You can define the class as well as create an object. Usually these are useful for interfaces and abstract classes.

Program:

abstract class My { // same achieved with interface

public abstract void display(); }

class Outer {

public void meth() {

// Anonymous inner class extending abstract class My

My m = new My() {

public void display() {

System.out.println("Hello"); } };

m.display(); } }

public class Test {

public static void main(String[] args) {

Outer o = new Outer();

o.meth();

}

}

1. Static inner class:

static classes are similar to nested class, only difference is they can be directly accessed outside the outer class.

class Outer

{

int x=10;

static int y=20;

static class inner {

public void show() {

System.out.println(y); } } }

public class LocalInner {

public static void main(String[] args) {

// Outer.Inner oi=new Outer().new Inner(); // nested class

Outer.inner m=new Outer.inner(); // static class

m.show(); } }

**Static and Final:**

**Static:**

Static keyword is used for representing metadata. Metadata means data about data.

representing the information of a class, not about the objects.

Static variables - are useful for representing data related to a class

Static members are useful for representing information or data related to a class.

Static class: if you have a lot of data and that can be grouped together and made as a class.

If you have a small information just some value, go for variable.

some processing required like addition or condition based, go for methods.

If you have a lot of methods, group them in a class.

Static variable and methods will be created inside the method area independent of the objects. that's why without creating the object, we can access it just by using a class name when the class is loaded.

Example:

Static variable price is representing the class so value of price belongs to both the objects h1 & h2

Class Hondacity{

static price =10; // data member , static variable

int a,b // non static variable method area

static double onRoadPrice() { // static method

switch(city) {

Case1 “bangalore” :

return price\*price\*0.1;

Case1 “mumbai” :

return price\*price\*0.5; }}

Program:

class Test {

static int x=10;

int y=20;

void show() {

System.out.println(x+" "+y); }

static void display() {

System.out.println(x); 🡪 Y cannot be accessed since y is not static.

} }

public class StaticPractice {

public static void main(String[] args) {

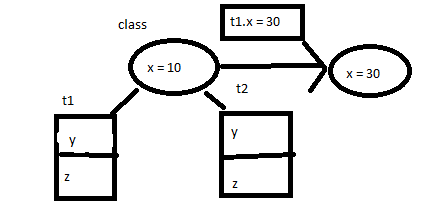
Test t1=new Test();

t1.show(); 🡪 10 20

t1.x=30; t1.y=50;

Test t2=new Test();

t2.show(); } } o/p: xx & YY???



**Static Blocks:**

There can be more than one static block in the class but they will execute in the order in the program mentioned.

this is not commonly used 🡪 only static variables and members are used.

Program:

class Test

{ // this static block will execute when the class is loaded ( Test t = new test)

static {

System.out.println("Block 1"); }

static {

System.out.println("Block 2");

} }

public class StaticPractice1 {

public static void main(String[] args)

{

// executes the block as per the order

//Test t=new Test();

System.out.println("Main");

Test t=new Test();

}

}

**Final:**

is **constant, unchangeable, or non-over ridable**, depending on where it is used.

**Final Variables:**

* these variables are constants, means the values are fixed, modification not allowed.
* final variables are written in capitals. notations followed in JAVA.
* 3 ways to initialize:
  1. initialize while declaring
  2. initialize inside static block
  3. initialize inside constructor

Example:

class Test {

1. final int X = 10;

x = 20; // ERROR – cannot assign a value to final variable

final int y; // blank final variable

final int z; // blank final variable

(2) static {

y = 100; }

(3) Test() {

Z = 100; // allowed: initialized once in constructor

}

}

**Final Methods:**

A **final method** in Java is a method that **cannot be overridden** by a subclass

class Test {

public final void show() {

System.out.println(); } }

class Test1 extends Test {

//public void show(){}

//gives error as final methods cannot be overloaded.

}

public void show1(){}

//this is possible

}

public class FinalDemo {

public static void main(String[] args) {

} }

**Final Class:**

A **final class** is a class that **cannot be extended (inherited)** by any other class.

In other words, **no subclass** can be created from it.

final class Parent {

void show() {

System.out.println("Inside final Parent class"); } }

// ❌ This will cause a compile-time error

/\*

class Child extends Parent {

void show() {

System.out.println("Inside Child class"); } } \*/

public class FinalClassDemo {

public static void main(String[] args) {

Parent p = new Parent();

p.show(); } }

**Conclusion:**

If 2 organizations working for same clients, each organization can restrict to override to write but just to read. Final is used in such cases.

**Singleton Class:** A **Singleton class** is a class that **allows only one instance** (object) of itself to be created and provides a **global point of access** to that instance.

Example:

class CoffeeMachine {

private float coffeQty;

private float milkQty;

private float waterQty;

private float sugarQty;

static private CoffeeMachine my=null;

private CoffeeMachine() // object can be created like (coffeeMachine m1=new CoffeeMachine)

{ coffeQty=1;

milkQty=1;

waterQty=1;

sugarQty=1; }

public void fillWater(float qty) {

waterQty=qty; }

public void fillSugar(float qty) {

sugarQty=qty; }

public float getCoffee() {

return 0.15f; }

static CoffeeMachine getInstance() {

if(my==null) // instance is null

my=new CoffeeMachine();

return my; } }

public class Singleton {

public static void main(String[] args) {

// All three variables reference the **same instance**.

CoffeeMachine m1=CoffeeMachine.getInstance();

CoffeeMachine m2=CoffeeMachine.getInstance();

CoffeeMachine m3=CoffeeMachine.getInstance();

System.out.println(m1+" "+m2+" "+m3);

if(m1==m2 && m1==m3)

System.out.println("Same"); } } 🡪 🡪 O/P : (Same)

**Program:** Program to assign rollno based on year.

import java.util.Date;

(1)class Student {

private String rollNo;

private static int count = 1;

// Assign roll number based on current year and static count

(3) private String assignRollNo() {

Date d = new Date();

String rno = "Univ-" + (d.getYear() + 1900) + "-" + count;

count++;

return rno; }

(2) Student() { // Constructor initializes roll number

rollNo = assignRollNo(); }

// Getter for roll number

public String getRollNo() {

return rollNo; } }

public class SCStatic1 {

public static void main(String[] args) {

Student s1 = new Student();

Student s2 = new Student();

Student s3 = new Student();

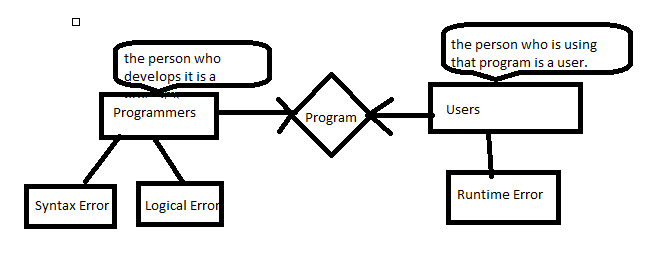
System.out.println(s1.getRollNo());

System.out.println(s2.getRollNo());

System.out.println(s3.getRollNo()); } } 🡪 🡪 O/P : (Univ-2025-1 Univ-2025-2 Univ-2025-3)

**Exception Handling:**

****



Exceptions are run time errors.

1. Syntax Error 🡪 syntactic mistake, or grammatical mistake,

or spelling mistakes,

Example:

int x ,y, (extra , but no semicolon)

x=10 (no semicolon)

z = x+y(Z is undefined)

compiler error 🡪 compiler cannot translate that to byte code,

1. Logical Error 🡪 no syntax error but bug in a program.

a = -b / 2 \* c; 🡪 a = -b /( 2 \* c); 🡪 bracket is imp

* Program executes and gives the result but it is not expected results.
* Logical errors are removed with the help of **debugger**

1. Runtime Error

* faced by users ->mishandling the programming
  + invalid input ( input age 🡪 -10, Printer, File missing in location)
  + Example : Car ( engine – programmers, fuel –users)
  + These are handled by **Exception Handling**

**How to handle Execeptions?**

Class Test {

P. S.V.M (Str args) {

int a, b, c;

int aa[] = {10,20,30,40,50}

a = 5; b= 0;

c = 5/0 🡪 infinite but for computers it is **undefined**. Fails and stops the execution – rest of the steps wont be executed.

S.o.p ( C) 🡪 Arithmetic Exception

S.o.p(aa[10]) 🡪 Arrayindexoutofboundexecetion

S.o.p ( “end pf prg”) }} 🡪 this can be achieved using Multiple catch blocks Nested try and catch blocks

Program:

1. **Try catch block**

import java.util.Scanner;

public class ExceptionDemo {

public static void main(String[] args) {

int a, b, c;

Scanner sc = new Scanner(System.in);

System.out.println("Enter 2 numbers");

a = sc.nextInt(); b = sc.nextInt();

try {

c = a / b; // ❗ May throw ArithmeticException if b = 0

System.out.println("Division is " + c);

} catch (ArithmeticException e) {

System.out.println("Denominator should not be 0, try again" + e.getmessage() or e ); }

System.out.println("Bye"); }}

1. **Multiple catch blocks & 3. Nested try and catch**

public class NestedCatch {

public static void main(String[] args) {

int A[] = {30, 20, 10, 40, 0};

try {

int c = A[0] / A[4]; // Outer try 🡪(SC1) As soon as this exception occurs, control jumps to the **outer catch (ArithmeticException e)** block:

System.out.println("Division is " + c);

try {

System.out.println(A[5]); // Inner try (SC2)

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Index is Invalid"); } }

catch (ArithmeticException e) {

System.out.println("Denominator should not be 0"); }

System.out.println("Bye"); } }

SC1 :

O/P :

Denominator should not be 0

Bye

SC2:

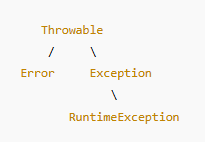
Division is 3

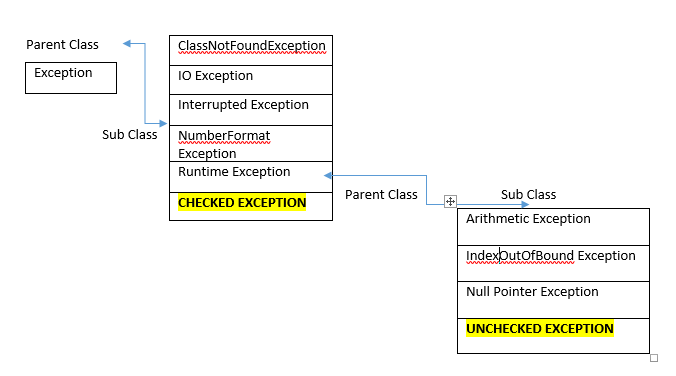
Index is Invalid

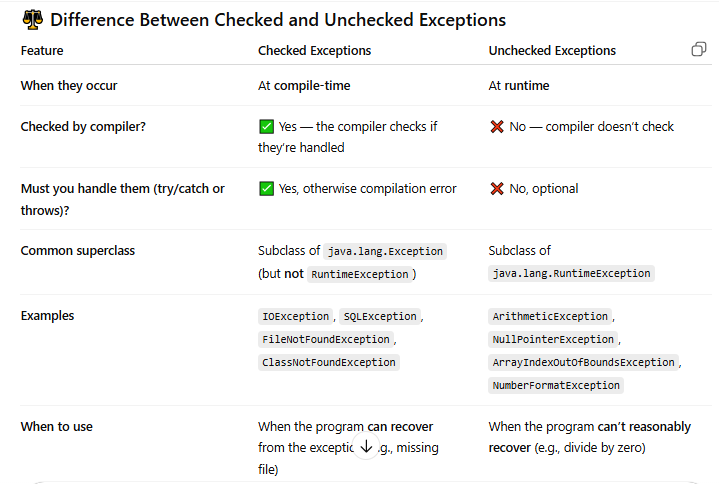
Bye

**Class Execption:**

Exception is the parent class for all the exceptions







Program:

import java.util.Scanner;

public class MultipleCatchDemo {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

try {

System.out.println("Enter two numbers:");

int a = sc.nextInt();

int b = sc.nextInt();

int c = a / b; // May cause ArithmeticException

System.out.println("Division is: " + c);

int[] arr = {10, 20, 30};

System.out.println("Enter index to access:");

int index = sc.nextInt();

System.out.println("Value at index " + index + " is " + arr[index]);

}

catch (ArithmeticException e) { 🡪 Sub class ( handles specific exceptions)

System.out.println("❗ Arithmetic Exception: Cannot divide by zero");

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("❗ Array Index Exception: Invalid index");

}

catch (Exception e) { 🡪 Super class (handles all exceptions)

System.out.println("❗ General Exception: " + e.getMessage()); e.getstring(),e.printStackTrace()

}

System.out.println("Program continues normally...");

}

}

Assignment:

Program for **custom checked exception**

class LowBalanceException extends Exception

{

public String toString()

{

return "Balance Should not be less than 5000";

}

}

public class CheckedUnchecked

{

static void fun1()

{

try

{

throw new LowBalanceException();

}

catch(LowBalanceException e)

{

System.out.println(e); } }

static void fun2()

{

fun1();

}

public static void main(String[] args)

{

fun2();

} }

O/P : Balance Should not be less than 5000

**Throw vs Throws:**

throw is used **inside a method or block** to **actually throw an exception object**.

throws is used in the **method signature** to **declare** that a method might throw one or more exceptions.

Example:  
Class Test {

P.S.V.M(Str arg) {  **(3) 🡪 use try catch block here**

meth1();

}

int meth1() { (2)🡪

try{

int a = area(10,5) or (-10,5)

}catch(Exception e)

S.o.p(e); }

int area(int l ,b) **throws** Exception { 🡪(1) calling method should handle the exception or

if (l <9 || b||b)

**throw** new Exception(“value can not be negativity”); 🡪 check exception(built-in class)

int a = l \*b;

return a ;

}

**Program:**

// Custom Exception Class

class NegativeDimensionException extends Exception {

public String toString() {

return "Dimensions cannot be Negative";

}

}

public class ThrowThrowsDemo {

// Method to calculate area

static int area(int l, int b) throws NegativeDimensionException {

if (l < 0 || b < 0)

throw new NegativeDimensionException();

return l \* b;

}

// Method that calls area()

static void meth1() throws NegativeDimensionException {

System.out.println("Area is " + area(10, 5));

}

// Main method

public static void main(String[] args) {

try {

meth1();

} catch (NegativeDimensionException e) {

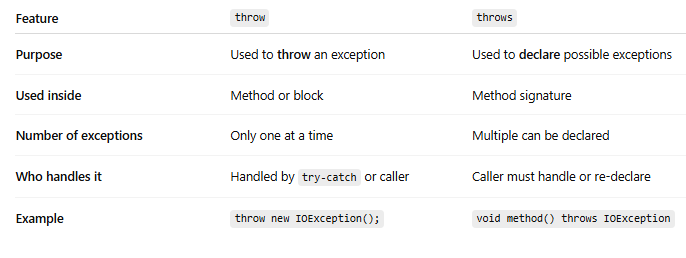
System.out.println(e);

} } }

O/P 🡪

Area is 50

**Difference throw vs throws:**



**Finally Block:**

the **finally block** is a block of code that **always executes** after a try and catch block — **whether an exception is thrown or not**. The finally block is **optional**

**finally**, is not for handling exceptions — it’s for **cleaning up resources or performing final actions** that must happen **regardless** of success or failure.

**Example1:**

public class FinallyDemo {

public static void main(String[] args) {

try {

int a = 10 / 2;

System.out.println("Result: " + a);

} catch (ArithmeticException e) {

System.out.println("Exception: " + e);

} finally {

System.out.println("This will always execute."); } } }

**Example2:**

public class FinallyDemo

{ static void meth1()throws Exception

{

try

{

throw new Exception();

}

finally

{

//this message willdefinitly execute

System.out.println("Final Message");

}

}

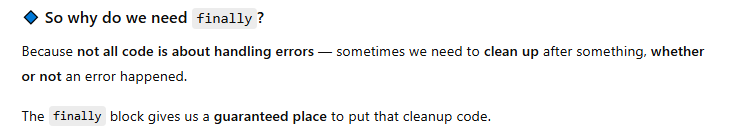
public static void main(String[] args) throws Exception

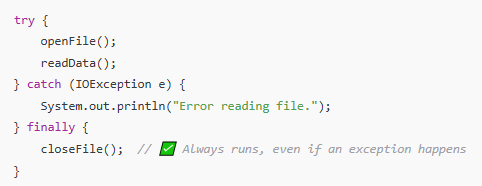
{

meth1();

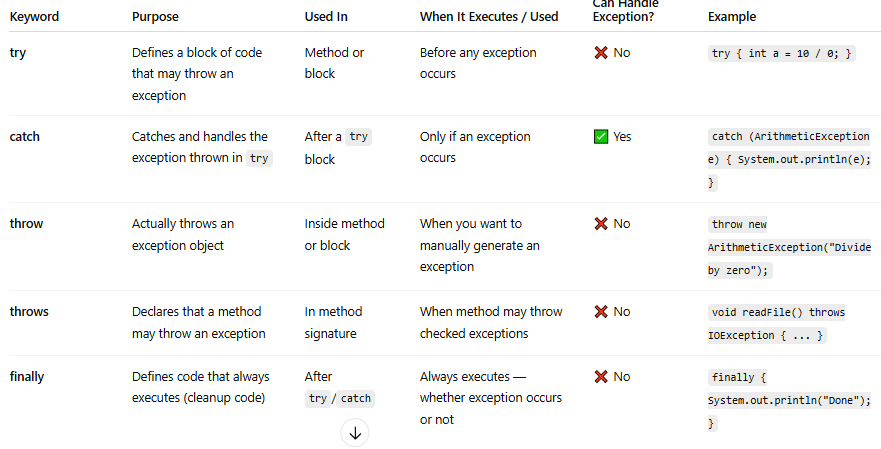
}

}





**Summary Table: Try – catch – throw – throws - finally**



**Multithreading:**

**Multithreading** allows a program to perform **multiple tasks simultaneously** by executing multiple threads within a single process.

For example, in a web browser:

* One thread handles user input,
* Another loads web pages,
* Another plays a video.

All these run at the same time using multithreading.

**Multithreading** means executing two or more parts of a program concurrently. Each part is called a **thread**, and Java provides the Thread class and Runnable interface to create and manage threads

**Control flow:**

All programs seen till now is single thread or single flow program. calling and called methods.

**how to achieve using multithreading?**

* Thread Class
* Runnable Interface

**Thread Class:**

* Class must extend Thread class
* run() 🡪 override the method which exists in Thread class
* Create object for thread and use start() method 🡪 calls the run method as thread
* one thread prints “Hello” and other thread prints “world” based on CPU time.

/\*class MyThread extends Thread

{

public void run()

{

int i=1;

while(true)

{

System.out.println(i+"Hello");

i++;

}

}

}\*/

class MyRunnable implements Runnable

{

public void run()

{

int i=1;

while(true)

{

System.out.println(i+"Hello");

i++; } } }

public class ThreadTest //implements Runnable //extends Thread

{

/\*public void run()

{

int i=1;

while(true)

{

System.out.println(i+"Hello");

i++; } }\*/

public static void main(String[] args) {

//MyThread t=new MyThread();

//ThreadTest t=new ThreadTest();

MyRunnable t=new MyRunnable();

Thread th=new Thread(t);

th.start();

int i=1;

while(true)

{ System.out.println(i+"World"); } } }

**Runnable Interface:**

* same like thread class but one change in main method
* Create object and send the object to run the thread.

**Real time example:**

* Video Player or YouTube
* Online Banking System
* Gaming

**States of Therad:**

+--------+

| NEW |

+--------+

|

| start()

v

+-----------+

| RUNNABLE |

+-----------+

/ | \

/ | \

BLOCKED WAITING TIMED\_WAITING 🡪 2 threads trying to access same file

\ | /

\ | /

v

+-----------+

|TERMINATED |

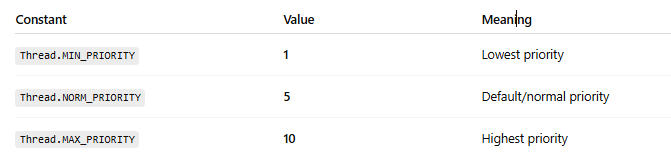
+-----------+

**Wait** : Thread is **waiting indefinitely**

**Blocked:** two threads want to access the same synchronized block — one runs, the other waits → BLOCKED

**Thread Priorities:**

Set priorities to threads



class PriorityDemo extends Thread {

public PriorityDemo(String name) {

super(name); }

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(getName() + " (Priority: " + getPriority() + ") - iteration " + i);

} }

public static void main(String[] args) {

PriorityDemo t1 = new PriorityDemo("Low Priority Thread");

PriorityDemo t2 = new PriorityDemo("Normal Priority Thread");

PriorityDemo t3 = new PriorityDemo("High Priority Thread");

// Set priorities

t1.setPriority(Thread.MIN\_PRIORITY); // 1 or Thread.(3)

t2.setPriority(Thread.NORM\_PRIORITY); // 5

t3.setPriority(Thread.MAX\_PRIORITY); // 10

// Start threads

t1.start();

t2.start();

t3.start(); } }

**Collection Framework:**

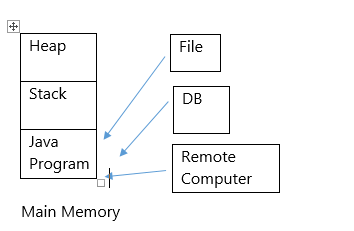
Collection framework is nothing but data structure.

If you're developing any application, then definitely, without data structure or collection framework, you cannot develop any application.

The data structure is building provided by Java and they are called as collections.

The **Java Collection Framework (JCF)** is a **set of classes and interfaces** that provide a **ready-made architecture to store, manipulate, and process groups of objects** efficiently.

It is part of the java.util package.



**Objects** created in **heap memory** and **Reference** created in **stack memory**. (show in diagram)

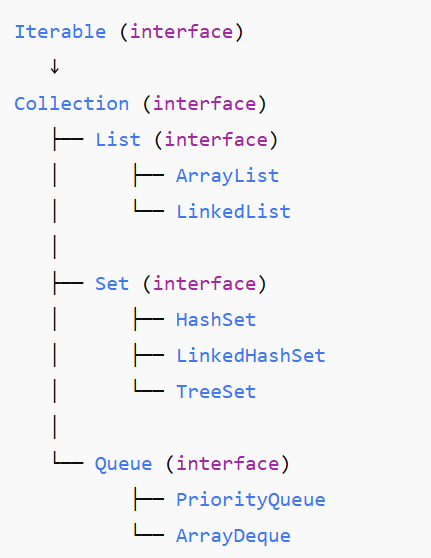
**Data (Java Program)** 🡪 fetches list of values from Db, File, Remote Computer through networking.

Example: Customer, Products, Accounts, Movies, Books, Students (explain)

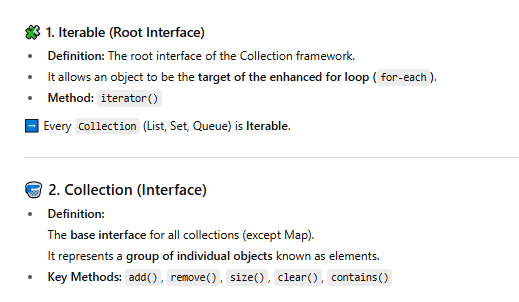
How to arrange all those data? that’s called as Data Structure or Collections.

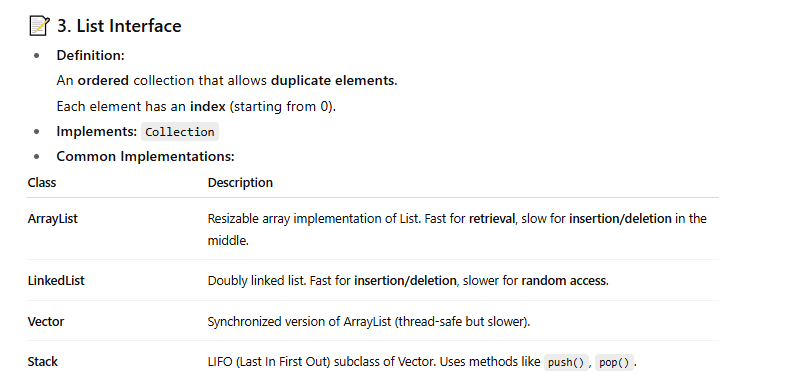
It’s mainly divided into **two branches**:

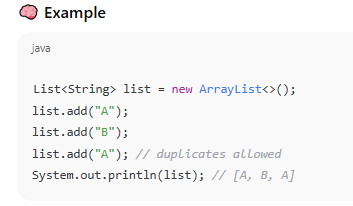
1. **Collection hierarchy** – for grouping elements (like List, Set, Queue)
2. **Map hierarchy** – for key–value pairs

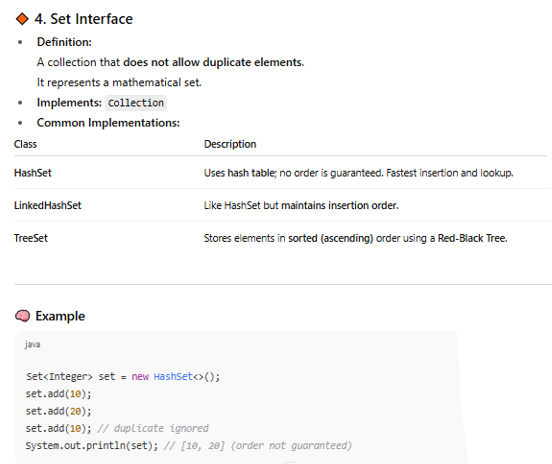


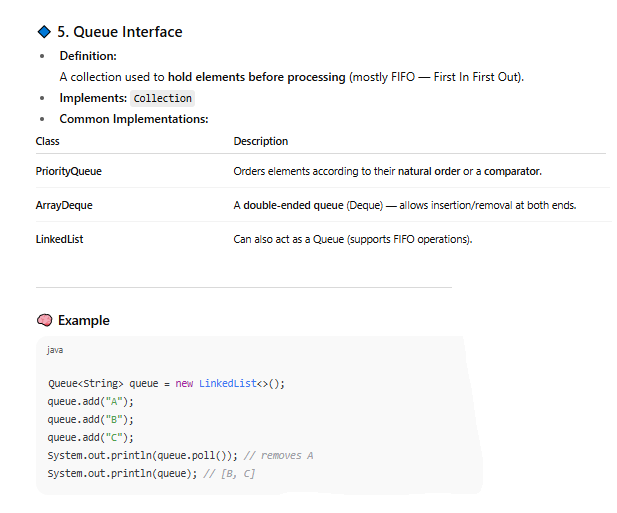
Oval shape 🡪 Interfaces, others are Classes.

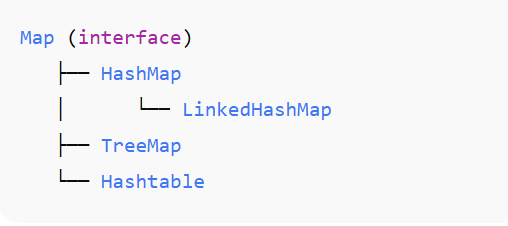


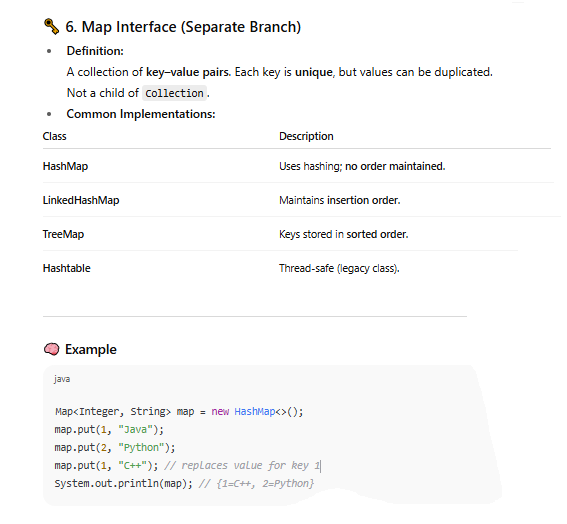


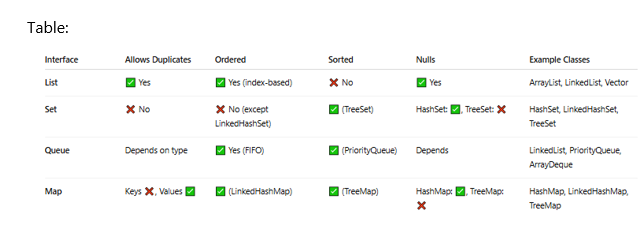










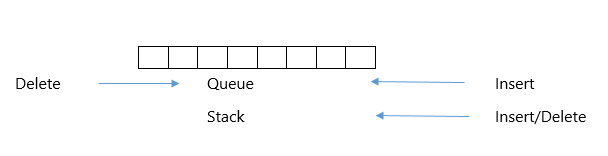


**Queue:**

**ArrayDeque:**

it supports **operations from both ends** (front & rear), and can act as both a **Queue** and a **Stack**.

**Null** is not allowed in ArrayDeque.



Queue 🡪 FIFO

**Main methods:**

* offer() / add() → insert (enqueue)
* poll() / remove() → remove (dequeue)
* peek() → view the front element

Stack 🡪 LIFO

**Main methods:**

* push() → add (insert)
* pop() → remove (from top)
* peek() → view the top element

**Reference for more methods:**

<https://www.geeksforgeeks.org/java/arraydeque-in-java/>

| **Category** | **Method** | **Action** | **End Affected** | **Returns / Behavior** |
| --- | --- | --- | --- | --- |
| **Add elements** | offerFirst(E e) | Adds element at the **front** | Head | Returns true if added |
|  | offerLast(E e) | Adds element at the **end** | Tail | Returns true if added |
|  | addFirst(E e) | Same as offerFirst but throws exception if full | Head | Throws exception if fails |
|  | addLast(E e) | Same as offerLast but throws exception if full | Tail | Throws exception if fails |
| **Remove elements** | pollFirst() | Removes and returns **first** element | Head | Returns null if empty |
|  | pollLast() | Removes and returns **last** element | Tail | Returns null if empty |
|  | removeFirst() | Removes and returns **first** element | Head | Throws exception if empty |
|  | removeLast() | Removes and returns **last** element | Tail | Throws exception if empty |
| **View elements** | peekFirst() | Returns **first** element without removing | Head | Returns null if empty |
|  | peekLast() | Returns **last** element without removing | Tail | Returns null if empty |
|  | getFirst() | Same as peekFirst() but throws exception if empty | Head | Throws exception if empty |
|  | getLast() | Same as peekLast() but throws exception if empty | Tail | Throws exception if empty |
|  |  |  |  |  |

**Program:** OfferFirst(2) OfferLast(1) OfferLast(3)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | **5** | **(10** | **20)** | **30** |  |  |  |

import java.util.ArrayDeque;

public class DequeAllOperationsDemo {

public static void main(String[] args) {

ArrayDeque<Integer> dq = new ArrayDeque<>();

System.out.println("=== ADDING ELEMENTS ===");

dq.offerLast(10); // add at end

dq.offerLast(20);

dq.offerFirst(5); // add at front

dq.offerLast(30);

System.out.println("Deque after offers: " + dq);

System.out.println("\n=== VIEWING ELEMENTS ===");

System.out.println("First element (peekFirst): " + dq.peekFirst());

System.out.println("Last element (peekLast): " + dq.peekLast());

System.out.println("\n=== REMOVING ELEMENTS ===");

dq.pollFirst(); // remove from front

dq.pollLast(); // remove from end

System.out.println("Deque after pollFirst & pollLast: " + dq);

System.out.println("\n=== USING STACK METHODS ===");

dq.push(100); // push at front (like stack)

dq.push(200);

System.out.println("After push: " + dq);

dq.pop(); // pop from front (like stack)

System.out.println("After pop: " + dq);

System.out.println("Top element (peek): " + dq.peek());

System.out.println("\n=== FINAL CONTENTS ===");

dq.forEach(x -> System.out.println(x));

}

}

**Output (with explanation):**

=== ADDING ELEMENTS ===

Deque after offers: [5, 10, 20, 30]

=== VIEWING ELEMENTS ===

First element (peekFirst): 5

Last element (peekLast): 30

=== REMOVING ELEMENTS ===

Deque after pollFirst & pollLast: [10, 20]

=== USING STACK METHODS ===

After push: [200, 100, 10, 20]

After pop: [100, 10, 20]

Top element (peek): 100

=== FINAL CONTENTS ===

100

10

20

**Explanation:**

* offerFirst() / offerLast() → add elements at both ends
* pollFirst() / pollLast() → remove elements from both ends
* push() / pop() / peek() → act like stack (LIFO)

forEach() → prints all current elements

**PriorityQueue:**

Elements are inserted or deleted based on the priority.

**Null** is not allowed in PriorityQueue.

If value is small 🡪 high priority

If value is large 🡪 low priority

Example:

16,25,15,8,26

8 🡪 high priority

26 🡪 low priority

**Reference for more methods:**

<https://www.geeksforgeeks.org/java/priority-queue-in-java/>

**Program: Min priority (default)**

import java.util.PriorityQueue;

public class PriorityDemo {

public static void main(String[] args) {

// Create a PriorityQueue (min-heap by default)

PriorityQueue<Integer> p = new PriorityQueue<>();

// Add elements

p.add(20);

p.add(10);

p.add(30);

p.add(5);

p.add(15);

p.add(3);

// Display the head element (smallest)

System.out.println("Peek (smallest element): " + p.peek());

// Display all elements (heap order, not sorted)

System.out.println("\nAll elements in the PriorityQueue:");

p.forEach((x) -> System.out.println(x));

// Remove the head element (smallest)

p.poll();

// Display after deletion

System.out.println("\nAfter Deletion (after removing smallest element):");

p.forEach((x) -> System.out.println(x));

}

}

**Output:**

Peek (smallest element): 3

All elements in the PriorityQueue:

3 5 30 20 15 10

After Deletion (after removing smallest element):

5 10 30 20 15

**Program:**

**Max priority 🡪 use comparator (used for min priority also)**

import java.util.\*;

// Custom Comparator for descending order (max-heap)

class MyCom implements Comparator<Integer> {

@Override

public int compare(Integer o1, Integer o2) {

if (o1 < o2) return 1; // Reverse natural order

if (o1 > o2) return -1;

return 0;

} }

public class PriorityDemo {

public static void main(String[] args) {

// Create a PriorityQueue with custom comparator (max-heap)

PriorityQueue<Integer> p = new PriorityQueue<>(new MyCom());

// Add elements

p.add(20);

p.add(10);

p.add(30);

p.add(5);

p.add(15);

p.add(3);

// Display the head element (largest due to custom comparator)

System.out.println("Peek (largest element): " + p.peek());

// Display all elements (heap order, not sorted)

System.out.println("\nAll elements in the PriorityQueue:");

p.forEach(x -> System.out.println(x));

// Remove the head element (largest)

p.poll();

// Display after deletion

System.out.println("\nAfter Deletion (after removing largest element):");

p.forEach(x -> System.out.println(x));

}

}

**Output**:

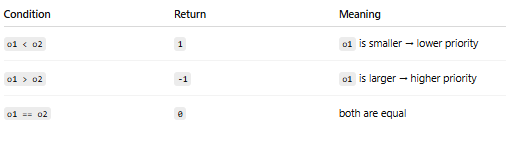
Peek (largest element): 30

All elements in the PriorityQueue:

30 15 20 5 10 3

After Deletion (after removing largest element):

20 15 3 5 10

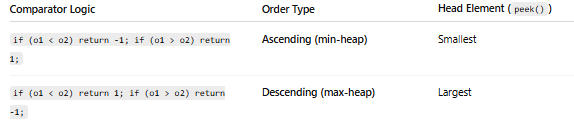


→ if (o1 < o2) return 1; if (o1 > o2) return -1;  
creates a **max-heap (descending order)** PriorityQueue.

→ if (o1 < o2) return -1; if (o1 > o2) return 1;

creates a **min-heap (ascending order)** PriorityQueue.

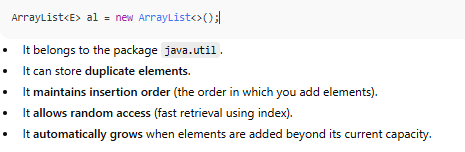
**Comparison:**



**List:**

**Array List:**

**ArrayList** is a **class in Java’s Collection Framework** that provides a **resizable (dynamic) array** implementation of the List interface.





Program:

import java.util.\*;

public class ListDemo {

public static void main(String[] args) {

// Create two ArrayLists

ArrayList<Integer> al1 = new ArrayList<>(20);

ArrayList<Integer> al2 = new ArrayList<>(List.of(50, 60, 70, 80, 90));

// Add elements

al1.add(10);

al1.add(0, 5);

al1.addAll(1, al2);

al1.add(5, 70);

al1.set(6, 100);

System.out.println("Initial al1: " + al1);

System.out.println("al2: " + al2);

// ---------------------------

// 1️⃣ remove(Object o)

// ---------------------------

// Removes the first occurrence of the given element

al1.remove(Integer.valueOf(70)); // remove first 70

System.out.println("\nAfter remove(70): " + al1);

// ---------------------------

// 2️⃣ removeAll(Collection<?> c)

// ---------------------------

// Removes all elements in al1 that are also in al2

al1.removeAll(al2);

System.out.println("After removeAll(al2): " + al1);

// ---------------------------

// 3️⃣ retainAll(Collection<?> c)

// ---------------------------

// Keeps only elements in al1 that are also in al2

al1.addAll(List.of(50, 60, 70, 80, 100)); // add again some from al2

System.out.println("\nBefore retainAll(al2): " + al1);

al1.retainAll(al2);

System.out.println("After retainAll(al2): " + al1);

// ---------------------------

// Print elements using forEach

// ---------------------------

System.out.println("\nFinal elements in al1:");

al1.forEach((x) -> {

System.out.println(x);

});

// Other ways to iterate:

// Using method reference

System.out.println("method reference");

al1.forEach(System.out::println);

//Using show method

System.out.println("show");

al1.forEach(n->show(n));

// Using enhanced for-loop

for (Integer x : al1)

System.out.println("enhanced for-loop" +x);

// Using basic for loop

for (int i = 0; i < al1.size(); i++)

System.out.println("basic for loop" +al1.get(i));

// Using Iterator

for (Iterator<Integer> it = al1.iterator(); it.hasNext(); ) {

Integer x = it.next();

System.out.println("Iterator"+x);

}

// Using ListIterator (Forward + Backward traversal)

// ---------------------------

System.out.println("\nForward traversal using ListIterator:");

ListIterator<Integer> li = al1.listIterator();

while (li.hasNext()) {

System.out.println("Index " + li.nextIndex() + " → " + li.next());

}

System.out.println("\nBackward traversal using hasPrevious():");

while (li.hasPrevious()) {

System.out.println("Index " + li.previousIndex() + " → " + li.previous());

} }

static void show(int n) {

if (n > 60)

System.out.println(n); } }

**Iterator vs ListIterator**

ListIterator is a **specialized iterator for List collections only** (like ArrayList or LinkedList).

**Forward and backward (**bidirectional traversal)

List<String> fruits = new ArrayList<>(List.of("Apple", "Banana", "Cherry"));

// Using Iterator

Iterator<String> it = fruits.iterator();

while (it.hasNext()) {

System.out.println(it.next());

}

// Using ListIterator

ListIterator<String> li = fruits.listIterator();

while (li.hasNext()) {

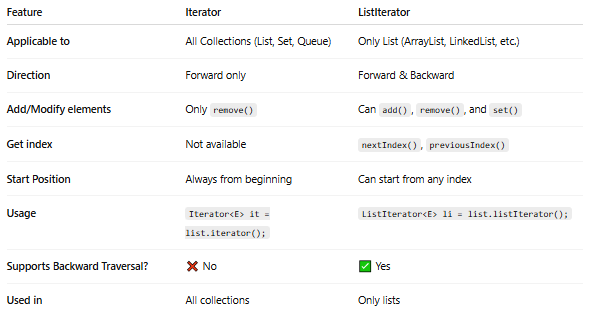
System.out.println(li.next());

}

while (li.hasPrevious()) {

System.out.println(li.previous());

}



**LinkedList**

It stores elements in a **linked sequence of nodes**, where each node contains the **data** and **references (links)** to the **previous** and **next** node.

It is part of the java.util package.

Linked list is a Collection of nodes

Size cannot be mentioned in linked list

**Working:**

* The **LinkedList** in Java is implemented as a **Doubly Linked List**.
* Each node contains three parts:
  1. **Previous node reference**
  2. **Data (element)**
  3. **Next node reference**

So, nodes are linked in both directions:

Head Next Pointer Null

|  |  |  |
| --- | --- | --- |
|  | 2 |  |

|  |  |  |
| --- | --- | --- |
|  | 45 |  |

|  |  |  |
| --- | --- | --- |
|  | 3 |  |

Start Pointer Previous Pointer

null ← [Prev | 10 | Next] ↔ [Prev | 20 | Next] ↔ [Prev | 30 | Next] → null

In Array -> To insert/delete 10 between 45 and 3, elements should be shifted

But in Linkedlist, shifting is not required. pointer is linked.

Program:

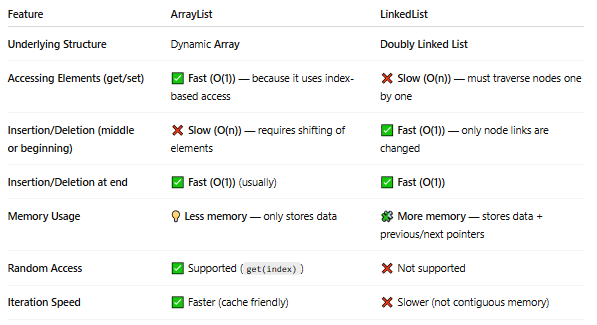
Same program as ArrayList but change only the declaration.

// Create two LinkedLists

LinkedList<Integer> al1 = new LinkedList<>();

LinkedList<Integer> al2 = new LinkedList<>(List.of(50, 60, 70,80, 90));

**ArrayList Vs LinkedList**

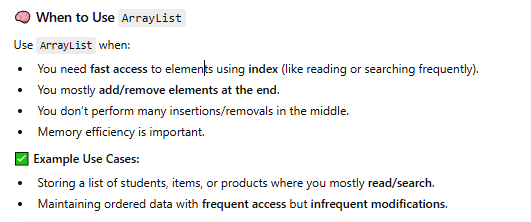


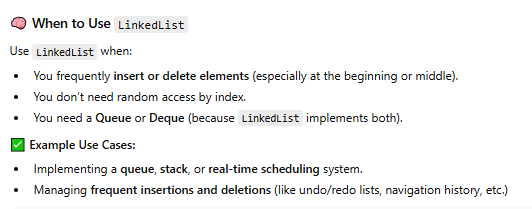
**Note:**

Complexity Meaning Example

O(1) Constant time Accessing by index → list.get(5)

O(n) Linear time Traversing all elements





**Set**

**Hashing Techniques:**

**1. What is Hashing?**

**Hashing** is a technique to **store and retrieve data efficiently**.  
It uses a **hash function** to convert a key (like a string or number) into an **index** in an array called a **hash table**.

index = hash(key) % table\_size

The element is stored at that index.

**2. Load Factor (λ)**

The **load factor** (λ) is a measure of how **full** the hash table is.

λ=Size of hash table / Number of elements​

**Example:**

If your hash table can hold 100 elements, and you’ve inserted 75,

λ=75/100==0.75

* In **Java’s HashSet/HashMap**, the **default load factor** is **0.75**.
* When the load factor exceeds this threshold, the table is **resized (rehashing)** — it’s doubled in size, and all elements are reinserted.

**Why Load Factor Matters:**

* **Low load factor (e.g., 0.5)** → fewer collisions → faster, but more memory used.
* **High load factor (e.g., 0.9)** → more collisions → slower, but memory efficient.

**Collisions**

A **collision** occurs when **two different keys** produce the **same hash index**.

Example:

hash("CAT") % 10 = 2

hash("DOG") % 10 = 2

Both map to index 2 → **collision**!

There are two main strategies to handle collisions:

1. **Chaining**
2. **Open Addressing**

**4. Chaining**

**Chaining** handles collisions by storing **multiple elements in the same index** using a **linked list (or tree)**.

**Structure Example:**

Index | Elements

0 | null

1 | A → B → C

2 | null

3 | X

* Each index in the table points to a **chain** (list) of entries.
* When a new element hashes to an occupied slot, it’s **appended to that chain**.

**Pros:**

* Simple to implement.
* The hash table can grow independently of the number of collisions.

**Cons:**

* Uses extra memory (for linked nodes).
* Lookup time increases if chains get long (O(n) in the worst case).

**Open Addressing**

**Not used in JCF**

In **open addressing**, all elements are stored **directly in the hash table** — **no linked lists**.

When a collision occurs, the algorithm searches for another **empty slot** in the array based on a **probing sequence**.

**Example (Linear Probing):**

Let’s say table size = 10

Insert values with hash results:

hash(25) = 5

hash(35) = 5 // collision!

→ Try next slot: index 6

Final table:

Index: 5 → 25

Index: 6 → 35

**Pros:**

* All data stored in one array (better cache performance).
* No extra memory for linked lists.

**Cons:**

* Performance degrades as the table fills.
* Requires careful deletion handling (to avoid breaking probe chains).

**Hashset:**

Set 🡪 it doesn’t allow duplicates, no order

Hash 🡪 it uses Hashtable, Chaining concept(linked list)

Operations 🡪 add, remove, contains and size

Intial capacity= 16, later on increase

loadFactor = 100, store only 50 elements (.5) by default .75

**Constant time**  means the operation takes the **same amount of time regardless of the number of elements** in the HashSet.

For example:

Set<Integer> numbers = new HashSet<>();

numbers.add(10); // O(1)

numbers.add(20); // O(1)

numbers.contains(10); // O(1)

numbers.remove(20); // O(1)

Even if the set has thousands of elements, these operations generally take about the same amount of time.

**Program:** Complete HashSet Program with All Common Methods

import java.util.\*;

public class SetDemo {

public static void main(String[] args) {

// Create a HashSet with initial capacity 20 and load factor 0.75

HashSet<Integer> hs = new HashSet<>(20, 0.75f);

// 1️⃣ add() — Adds elements (duplicates are ignored)

hs.add(10);

hs.add(20);

hs.add(30);

hs.add(10); // duplicate, will be ignored

System.out.println("Initial HashSet: " + hs);

// 2️⃣ contains() — Check if a value exists

System.out.println("Contains 20? " + hs.contains(20));

System.out.println("Contains 50? " + hs.contains(50));

// 3️⃣ remove() — Remove a specific element

hs.remove(20);

System.out.println("After removing 20: " + hs);

// 4️⃣ size() — Number of elements

System.out.println("Current size: " + hs.size());

// 5️⃣ isEmpty() — Check if set is empty

System.out.println("Is HashSet empty? " + hs.isEmpty());

// 6️⃣ addAll() — Add multiple elements from another collection

HashSet<Integer> hs2 = new HashSet<>();

hs2.add(40);

hs2.add(50);

hs2.addAll(hs); // adds all elements from hs

System.out.println("After addAll(): " + hs2);

// 7️⃣ removeAll() — Remove all elements that exist in another collection

hs2.removeAll(hs);

System.out.println("After removeAll(hs): " + hs2);

// 8️⃣ retainAll() — Keep only common elements between sets

hs.add(50);

hs.add(60);

System.out.println("hs: " + hs);

System.out.println("hs2: " + hs2);

hs.retainAll(hs2);

System.out.println("After retainAll(hs2): " + hs);

// 9️⃣ clear() — Remove all elements

hs.clear();

System.out.println("After clear(): " + hs);

// 🔟 clone() — Shallow copy of HashSet

HashSet<String> names = new HashSet<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

HashSet<String> copy = (HashSet<String>) names.clone();

System.out.println("Original set: " + names);

System.out.println("Cloned set: " + copy);

// 1️⃣1️⃣ iterator() — Traverse elements manually

System.out.print("Iterating with iterator: ");

Iterator<String> it = names.iterator();

while (it.hasNext()) {

System.out.print(it.next() + " "); }

System.out.println();

// 1️⃣2️⃣ forEach() — Java 8+ method for iteration

System.out.println("Using forEach():");

names.forEach(n -> System.out.println("Name: " + n));

}}

**Output:**

Initial HashSet: [20, 10, 30]

Contains 20? True Contains 50? false

After removing 20: [10, 30]

Current size: 2

Is HashSet empty? false

After addAll(): [50, 40, 10, 30] After removeAll(hs): [50, 40]

hs: [50, 10, 30, 60] hs2: [50, 40]

After retainAll(hs2): [50] After clear(): []

Original set: [Alice, Charlie, Bob] Cloned set: [Alice, Charlie, Bob]

Iterating with iterator: Alice Charlie Bob

Using forEach():

Name: Alice

Name: Charlie

Name: Bob

**Treeset:**

 Elements in a TreeSet are **always sorted** (ascending order by default).

 Null elements are **not allowed**

**Program:**

import java.util.\*;

public class SetDemo1 {

public static void main(String[] args) {

TreeSet<Integer> ts=new TreeSet<>(List.of(10,30,50,70,10,40));

ts.add(25);

//ts.ceiling(55); 🡪 Ceiling – nerest to 55 but bigger

System.out.println(ts);

}

}

**Note:**

**Use same as Hashset program and change the syntax as TreeSet.**

**LinkedHashSet:**

**Order:** **Insertion order is preserved** — elements are iterated in the order they were added.

**No duplicates**

**Program:**

import java.util.\*;

public class LinkedHashSetDemo

{

public static void main(String[] args)

{

LinkedHashSet<String> lhs=new LinkedHashSet<>(10);

lhs.add("A");

lhs.add("C");

lhs.add("E");

lhs.add("K");

lhs.add("B");

lhs.add("G");

lhs.add("B");

Iterator<String> itr=lhs.iterator();

while(itr.hasNext())

{

System.out.println(itr.next());

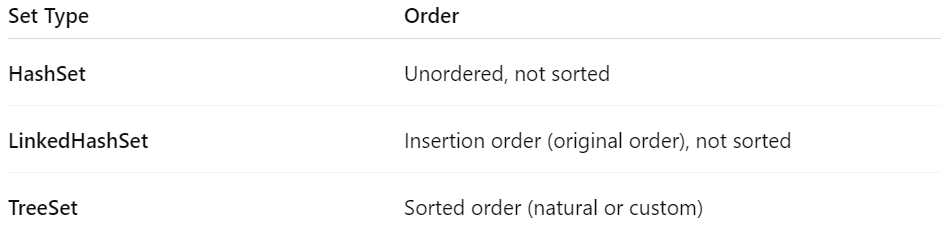
}

}

}

**Note:**

**Use same as Hashset program and change the syntax as LinkedHashSet.**



**Maps:**

**1. What is a Map?**

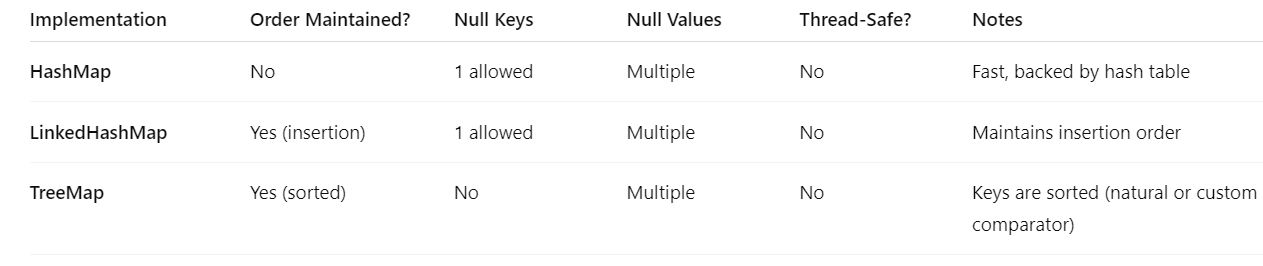
A **Map** is a data structure that stores **key-value pairs**, where:

* **Key**: Unique identifier used to retrieve the value.
* **Value**: Data associated with the key (can be duplicate).

**Key points about Maps:**

1. Keys **must be unique**.
2. Values **can be duplicate**.
3. A Map **does not maintain order** by default (depends on implementation).

**Table**



**TreeMap**

* Implements SortedMap interface.
* **Keys are sorted** naturally
* Cannot have **null key**
* Can have **null values**.

**Program:**

import java.util.Map;

import java.util.TreeMap;

import java.util.Map.Entry;

public class TreeMapDemo {

public static void main(String[] args) {

// Create TreeMap with initial entries

TreeMap<Integer, String> tm = new TreeMap<>(Map.of(0,"A",1,"B",2,"C",3,"D"));

// Add more entries

tm.put(4, "E");

tm.put(6, "F");

// Display all entries (key=value)

System.out.println("TreeMap: " + tm);

// ceilingEntry: first key >= 5

System.out.println("ceilingEntry(5): " + tm.ceilingEntry(5).getValue());

// containsKey

System.out.println("Contains key 3? " + tm.containsKey(3));

// containsValue

System.out.println("Contains value 'E'? " + tm.containsValue("E"));

// entrySet

System.out.println("Entry set:");

for (Entry<Integer, String> entry : tm.entrySet()) {

System.out.println(entry.getKey() + " = " + entry.getValue());

}

// get

System.out.println("Value for key 2: " + tm.get(2));

// firstEntry

Entry<Integer, String> e = tm.firstEntry();

System.out.println("First entry: " + e.getKey() + " " + e.getValue());

// Display TreeMap again

System.out.println("Final TreeMap: " + tm);

}

}

Now firstEntry() returns the entry with the smallest key, which is 0=A.

The map is sorted by keys automatically.

**HashMap**

**Note:** Hashmap is same as TreeMap, first entry and ceiling is used only in Treemap.

Change syntax and use as Hashmap.

* Implements Map interface using **hash table**.
* **Order:** Unordered.
* Can have **one null key**.
* Can have **multiple null values**.

**Example:**

HashMap<Integer, String> map = new HashMap<>();

**Assignment:**

**LinkedHashMap**

* Extends HashMap, **maintains insertion order** (or access order).
* Slightly slower than HashMap due to linked list overhead.

**Example:**

LinkedHashMap<String, Integer> map = new LinkedHashMap<>();

map.put("Alice", 90);

map.put("Bob", 85);

for(String key : map.keySet()) {

System.out.println(key); // Alice, Bob }