JAVA NOTES

Java is a **High-Level Language**(easy for humans to read and write).

***HLL*:**High-Level Language , Human understandable code.

***LLL*:**Low-level language(machine language), contains only 1's and 0's and is directly understood by a computer.

***JDK* :**JAVA DEVELOPMENT KIT

- It is a collection of software tools, libraries, java compiler JRE etc.
- It enables developers to write, compile, and run Java programs.

Compiler

- It translates the entire source code of HLL into LLL or an intermediate code(closer to machine code) in a single step.
- It scans syntax errors.

Interpreter

- The interpreter translates HLL line by line.

WHOLE PROCESS OF RUNNING A JAVA CODE

Java compiler translates source code into byte code.The JVM (DTL) loads and executes the byte code. Optionally, some JVMs may choose to interpret the byte code directly for certain use cases.This combination of compilation and interpretation allows Java programs to be both platform-independent (byte code run on any machine).

|| DAY 1 ||

JDK :https://www.oracle.com/java/technologies/downloads/#jdk21-windows

IDE :https://www.jetbrains.com/idea/download/?section=windows

Boiler-plate || Default Code

```
public class Demo1 {
public static void main(String[] args) {

    }
}
```

Class- Collection of methods and variables. It is concept of OOPs(DTL).

Method - A method is a block of code which performs certain operations and returns output(DTL).

Entry Point

The **main** method is the entry point for executing a Java application.

When you run a Java program, The JVM or java compiler looks for a public static void main(String args[]) method in that class, and the signature of the main method must be in a specified format for the JVM to recognize it as its entry point.

If we update the method's signature, the program will throw the error NoSuchMethodError:main and terminate.

|| DAY 2 ||

Just like we have some rules that we follow to speak English(the grammar), we have some rules to follow while writing a Java program. The set of these rules is called Syntax.

Variables

A variable is a container that holds data. This value can be changed during the execution of the program.

Before use, you need to declare and define it.

1. Variable Declaration:

```
int age;
String name; //int and string is data types
```

2. Variable Initialization:

```
age = 69;
name = "the boys";
```

3. Combined Declaration and Initialization:

```
int age = 69;
String name = "the boys";
```

4. Final Variables (Constants):

final int a = 7;[DTL]

Role of + operator between String & numbers

- String + String = String - Concatenation
- String + int = String - Concatenation
- int + int = int - Arithmetic Addition

|| DAY 3 ||

Identifiers- Identifiers are used to uniquely identify the variables.

Identifier is a name given to a variable, class, method, package, or other program elements.

Rules for Identifiers in Java:

1. Start: Must start with an alphabet or _ or $ NOT with a digit.

2. End: Can end with an alphabet or _ or $ or numeric digit.

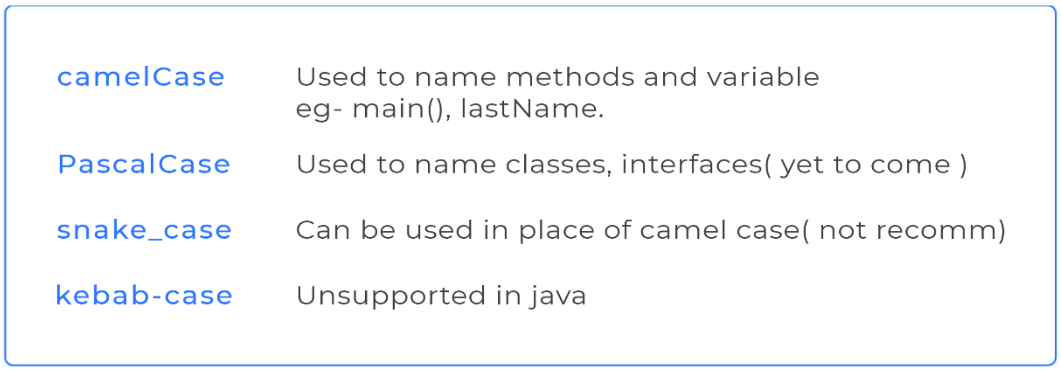
3. No Reserved Words : You cannot use Java's reserved words (also known as keywords) as identifiers.

4. No Special Symbols: Identifiers cannot contain special symbols like @, #, %, etc. except for underscores (_) and dollar signs ($).

5. No Space : Spaces are not allowed.

6. Length – No Limit

Java is CASE SENSITIVE : Shery and shery is different for java



camelCase	Used to name methods and variable eg- main(), lastName.
PascalCase	Used to name classes, interfaces(yet to come)
snake_case	Can be used in place of camel case(not recomm)
kebab-case	Unsupported in java

Keyword and word : Keywords are reserved(built-in) words which has specific meanings and cannot be used as identifiers.e.g.public, class, static, if, else, while etc.

|| DAY 4 ||

Literal or Constant:

Any constant value which can be assigned to the variable.

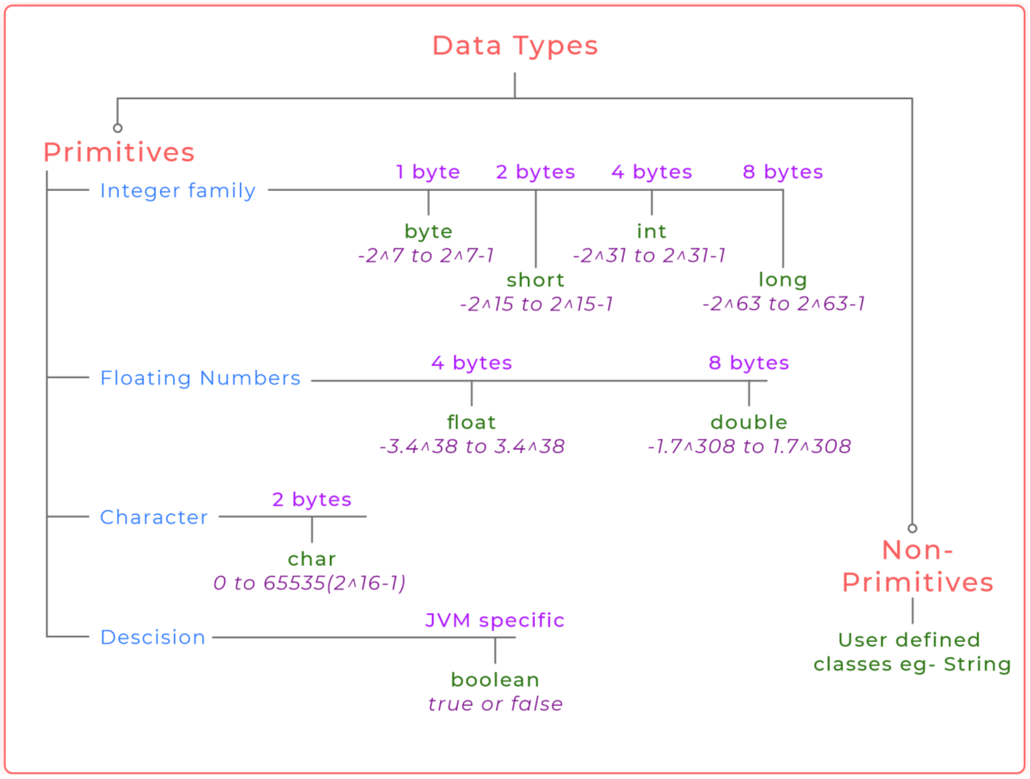
DATA TYPES

Data types are used to classify and define the type of data that a variable can hold.

There are 2 types of Data Types:

1.Primitive Data types: pre-defined, fixed size.

2.Non-Primitive Data types: Customize and no fixed size.



Default Values

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

the compiler never assigns a default value to an uninitialized local variable(DTL).

+ operator between two char values

- It performs addition between their Unicode code points.
- For example :

```
char a = 'a';
char b = 'b';
System.out.println(a+b);//97+98=195
```

Output : 195

|| DAY 5 ||

Scanner

To take input from users we use Scanner class.

Scanner class is a built-in class in the java.util package(DTL). Before using the Scanner class you have to import the Scanner class using the import statement as shown below:

To use the **Scanner** class, you need to create an object of it, and then you can use that object to interact with the input data.

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

```
Scanner sc = new Scanner(System.in);//object
int n = sc.nextInt();
```

The nextInt() method parses the token from the input and returns the integer value.

Use methods to read respective data

nextByte(), nextShort(), nextInt(),nextLong(),nextFloat(), nextDouble(), nextBoolean()

Reading String Data

nextLine() - Reads the whole line

next() - Reads the first word

Reading char data–next().charAt(0)

Problem with nextLine() method:

Ifwe try to read String after reading in an Integer, Double or Float etc.

Java does not give us a chance to input anything for the name variable.

When the method input.nextLine() is called Scanner object will wait for us, to hit enter and the enter key is a character(“\n”).

Example

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");
int age = scanner.nextInt();

System.out.print("Enter a string: ");
```

```
String name = scanner.nextLine();

System.out.println(name + " age is = " + age);
```

Console :

Enter an integer: 69 // 69\n(enter)

Enter a string: age is = 69

1. We first prompt the user to enter an integer age using **nextInt()**.
2. After reading the integer, we immediately hit enter and enter is also a character represented by “\n” – 69\n
3. The int value 69 is assigned in age but not the \n still left in the memory or buffer.
4. In next line when we Call **nextLine()** to consume the name it first check in buffer is there any thing as we have \n in buffer it take \n (for nextLine() method \n is the stopping point it will consider we stop giving input and return) and skip the line.

Solution :

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");
int age = scanner.nextInt();

// Consume the newline character left in the input buffer
scanner.nextLine();

System.out.print("Enter a string: ");
String name = scanner.nextLine();

System.out.println(name + " age is = " + age);
```

1. After taking an integer input we Call nextLine() to consume the name character left in the input buffer.In next line when we Call nextLine() to consume the name character left in the input buffer.

2. Then, we prompt the user to enter a string using nextLine().

\ is a special symbol

\n (next Line), \b (backspace), \t (tab), \" (double quote), \' (single quote), and \\ (backslash).

|| DAY 6 ||

Operators

Operators can be easily defined as characters that represent an operation. These symbols perform different operations on several variables and values.

Example : 5 + 6 = 11. Here, 5 and 6 are the **operands**, and **+** is called the **operator**.

Categories of Operators

Unary operators : perform an action with a **single operand**.

Binary operators: perform actions with two **operands.**

Types of Operator

1. **Arithmetic Operator :**

   Arithmetic operator can be divided into two categories -

- **Binary Operators :** +, -, *, /(int/int will always yield int) , % (Return remainder after dividing two numbers & with int (works perfectly) but with float (produces ambiguity)).

  Special powers of / & % by powers of 10

  / : to reduce the number by 1 digit

  % : to get last digit(s) of number

- **Unary Operators :**

  Increment Operator **(++)** : Increase the value by 1.
  Decrement Operator**(- -)** : Decrease the value by 1.

  +, - and **!**(DTL) is also a unary operator(-5, 5 (is same as +5)).

RULES for Increment and Decrement :

- Cannot applied to constant
  Example : int c = ++10; // compile-time error

- Nesting of both operators is not allowed
  Example :int a = 10;
  int b = ++(++a); // compile-time error [++11]
- They are not operated over final variables
  Example : final int a = 10;
  int b = ++a; // compile-time error
- Increment and Decrement Operators can not be applied to booleans.
  Example : boolean a= false;
  a++;// compile-time error

Quiz On Increment And Decrement Operators :
https://sheryians.com/whizz/test/651ce22ebf489c724c896167

2. Relational Operators : Used to check the relations between two operands. They return a boolean value (true or false) by comparing the two operands.

Greater Than (>) ,Less Than (<), <=, >=, ==, !=

- **Equal To (==)** - Checks if two operands are equal
- **Not Equal To (!=)** - Checks if two operands are not equal
- **Greater Than or Equal To (>=)** - Checks if one operand is either greater than or equal to the other.
- **Less Than or Equal To (<=)** - Checks if one operand is either less than or equal to the other.

3. Logical operators : Combine multiple conditional statements. There are three types of logical operators in Java: **AND(&&), OR (||) and NOT(!)** operators.

- **Logical AND Operator(&&)**

  Returns **true** when both conditions under evaluation are true, otherwise it returns **false**.
  e.g : if(a>b && a<c) System.out.println("Maximum: " + a);

- **Logical OR Operator(||)**

  Returns **true** if any one of the given conditions is true, otherwise it returns **false**. It returns **false** if and only if **both conditions** under evaluation are false.

  e.g : if(a>b || a<c) System.out.println("Max: " + a);

- **Logical Not Operator(!)**

It accepts a single value as an input and returns the inverse of the same. This is a unary operator unlike the AND and OR operators.
e.g : if(!(a<b)) System.out.println("Max: " + a);

4. ShortHand operators

The assignment operator can be combined with other operators to build a shorter version of the statement. **+=, -=, *=, /=, %=**

Example : a = a+5, we can write a **+=** 5.

do not use **=+ & -=** [(=) followed by a unary plus (+)]

|| DAY 7 ||

Package

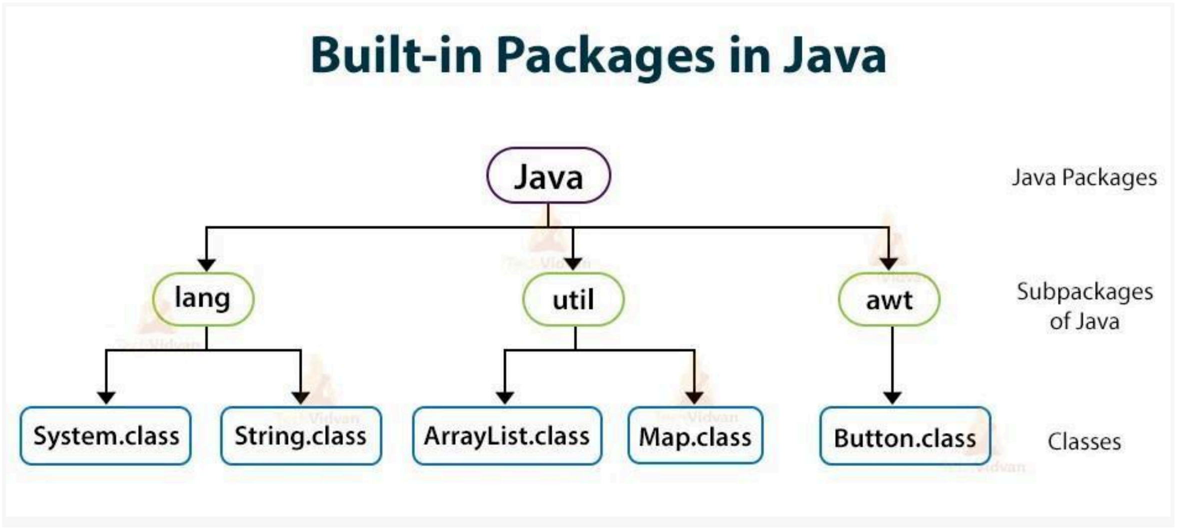
A Java package is a collection of similar types of sub-packages, interfaces, and classes. They help you manage and group related classes, interfaces, and sub-packages to avoid naming conflicts and create a more organized and maintainable codebase.

Example:

Directories or folders on your computer's file system(manage files).

In Java, there are two types of packages: built-in packages and user-defined packages.

in-built packages : They are available in Java, including util, lang, awt etc. We can import all members of a package using package name.* statement



java.lang package is a special package that is automatically **imported by default** in every Java class.

Commonly used classes and types from the java.lang package include: String, System, Math etc.

User-defined packages : User-defined packages are those that the users define. Inside a package, you can have Java files like classes, interfaces, and a package as well (called a sub-package).

Math Class

java.lang.Math class is a built-in class. It provides mathematical functions and constants for mathematical operations.

Commonly used methods and constants:

Math.abs(a)	Returns the absolute value of a value.
Math.sqrt(a)	Returns the sqrt root of a double value.

Math.ceil(a)	Returns the closest value that is greater than or equal to the argument
Math.max(a, b)	Returns the greater of two values.
Math.min(a, b)	Returns the smaller of two values.
Math.pow(a, b)	Returns a raised to the power of b.
Math.random()	Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Math.PI, Math.floor(a)(closest value that is greater than or equal to the argument) etc.

|| DAY 8 ||

CONTROL-FLOW STATEMENTS

Control Flow statements in programming control the order of execution of statements within a program. They allow you to make decisions, repeat actions, and control the flow of your code based on conditions.

Types of control flow statements

1. Conditional or Decision Making statements (if-else and switch)

2. Looping statements (for, while, and do-while)

3. Branching statements (break and continue)

1.Conditional statements **If-else** :

The **if-else** statement allows you to execute a block of code conditionally.

If the condition inside the **if** statement is true, the code inside the **if** block is executed;

otherwise, the code inside the **else** block is executed.

Syntax of if-else :

```
int age = 30;
if(age >18) {
    System.out.println("Adult");//executes if condition is true
}else {
    System.out.println("Abhi chote ho"); //condition false
}
```

If-Else-If Ladder :

"If-Else-If" ladder consists of an if statement followed by multiple else-if statements. It is used to evaluate a condition using multiple statements. The chain of if statements are executed from the top-down.

It checks each if condition, and as soon as one of the if condition yields true, it executes the statement inside that if block and **skip the rest of the ladder**. If none of the conditions evaluates to be true, then the program executes the statement of the final **else** block.

For example :

```
int number = 10;
if (number % 2 ==  0) {
    System.out.println("Number is even.");
}
else if (number % 2 != 0) {
    System.out.println("Number is odd.");
}
else {
    System.out.println("Invalid input.");
}
```

Output : Number is even

If Ladder :

"If" ladder consists of an multiple if statements. It is used to evaluate a condition using multiple statements. The chain of if statements are executed from the top-down.

The program checks each if condition, and as soon as one of the if condition yields true, it executes the statement inside that if block and **still check further conditions**. If none of the conditions evaluates to be true, then the program executes the statement of the final **else** block.

```
int number = 10;
if (number >0) {
    System.out.println("Number is positive.");
}
if (number <20) {
    System.out.println("Number is less than 20.");
}
if (number % 2 == 0) {
    System.out.println("Number is even.");
}
```

Output : Number is positive

Number is less than 20.";

Number is even.

|| DAY 11 ||

Ternary Operator

The ternary operator, also known as the conditional operator, is a shorthand way of writing an **if-else** statement with a single expression.

Syntax : **condition ? expression1 : expression2**

- If the **condition** is true, the expression before the **:** (i.e., **expression1**) is evaluated and returned.
- If the **condition** is false, the expression after the **:** (i.e., **expression2**) is evaluated and returned.

Example:

```
int num = ;
String result = (num % 2 == 0) ? "Even" :"Odd";
System.out.println("The number is " + result);
```

Output : Even

Type Conversion

Type casting in Java is the process of converting one data type to another. It can be done automatically or manually.

Type Casting in Java is mainly of two types.

1. Widening or Implicit Type Casting
2. Narrow or Explicit Type Casting

1.Widening or Implicit Conversion:

- Java allows automatic type conversion when a smaller data type is promoted to a larger data type..
- It is secure since there is no possibility of data loss.
- **Both the data types must be compatible with each other :** converting a string to an integer is not possible as the string may contain alphabets that cannot be converted to digits.

Order :byte->short->int->long->float->double

char->int

Example :

```
int intValue = 42;
double doubleValue = intValue; // Implicit conversion
```

2.Explicit or Narrowing Conversion:

- Sometimes, we need to convert a larger data type to a smaller one explicitly and it requires a cast operator.
- **Narrowing Type Casting in Java is not secure** as loss of data can occur due to shorter range of supported values in lower data type.

Example :

```
double doubleValue = 42.0;
int intValue = (int) doubleValue; // Explicit
conversion (casting)
```

Note : Shorthand operators do implicit conversion.

Byte b = 1;

b=b+2; // error , 2 is int(all non-float by default int) so can't store in byte

b += 2; // works perfectly as += did implicit conversion

|| DAY 12 ||

Looping statements

When we want to perform certain tasks again and again till a given condition.

For e.g. : Our daily routine, certain song listen again & again

Looping is a feature that facilitates the execution of a set of instructions repeatedly until a certain condition holds false.

e.g. : print 1 to 10,000 number

Types of Loop

Categorized into two main types

1.Entry Controlled

Check the loop condition before entering the loop body. If the condition is false initially, the loop body will not execute at all.

for and **while** loops are examples of entry-controlled loops as we check the condition first and then evaluate the body of the loop..

a. for loop

When we know the exact number of times the loop is going to run, we use for loop.

Syntax:

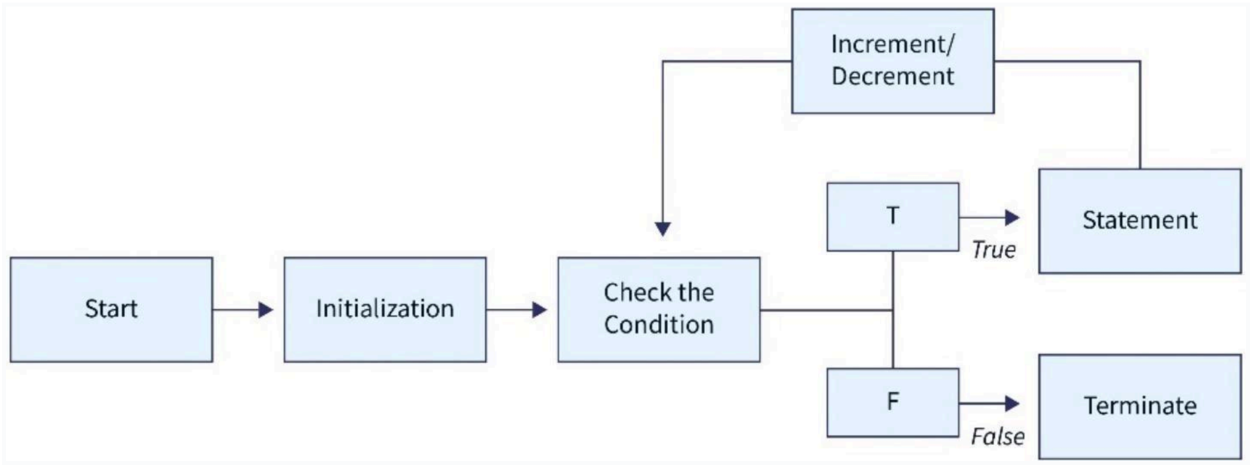
```
for(declaration,Initialization ; Condition ; Change){
   // Body of the Loop (Statement(s))
}
```

Example :

```
for (int i = 1; i<= 5; i++) {
    System.out.println(i);//run 1 to 5
}
```

Output : 1 2 3 4 5

FLOW DIAGRAM



Optional Expressions :

In loops, **initialization**, **condition**, & **update** are optional. Any or all of these are skippable.The loop essentially works based on the semicolon **;**

```
// Empty loop
for (;;) {}

// Infinite loop
for (int i = 0;; i++) {}

// initialization needs to be done outside the loop
// i and n needs to be defined before
for (; i< n; i++) {}
```

Syntax tweaks

- Initialize the variable outside the loop.
- Multiple conditions.
- Increment or Decrement of variable inside loop body

An infinite loop is a loop that continues executing indefinitely, and it doesn't have a condition that will terminate the loop naturally.

```
for (;;){
    System.out.println("This is an infinite loop");
}
```

In the above code there is no initialization, no condition, and no iteration expression, meaning it will run indefinitely unless explicitly terminated.

```
for(;;);
```

This is another example of an infinite loop, but this time, there is no code or statements within the loop. It's just an empty loop that will run indefinitely.

As the loop has started but it never ends, you terminate it by ; it never comes out of the loop and if you write any code after this loop, it will be unreachable because the loop never terminates.

|| DAY 13 ||

while loop

The while loop is used when the number of iterations is not known but the terminating condition is known.

Loop is executed until the given condition evaluates to false.

Syntax:

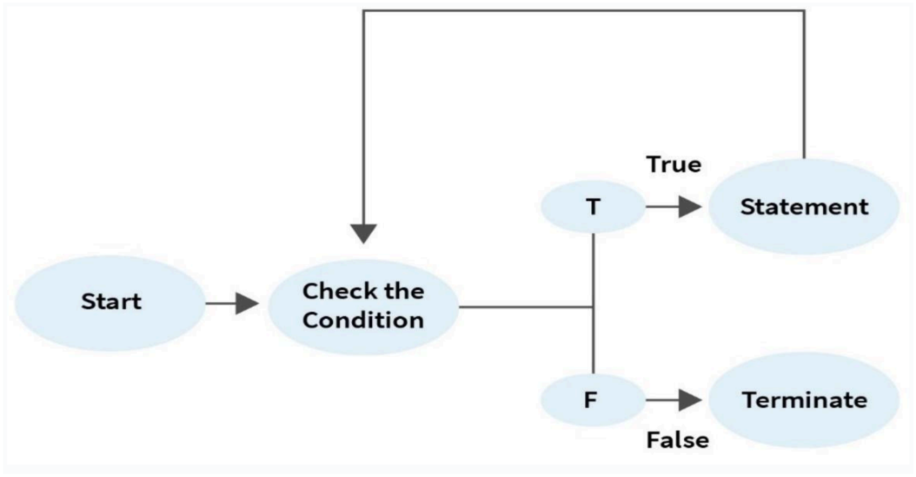
```
// initialization
// while (condition){
//       Body of the loop
//       Updation
// }
```

Example :

```
int i=0;
while (i<5){
    System.out.println(i);
i++;
}
```

Output :0 1 2 3 4

FLOW DIAGRAM



While always accepts true, if you initially give a false condition (not Boolean false) it will neither give a syntax error nor enter in the loop.

```
int i=0;
while (i>9){// false condition but no syntax error
System.out.println(i);
}
```

While loop always accepts true ,if you initially give false(Boolean value) it will give syntax error.

```
while (false){ //Syntax error
    System.out.println(“Hello LOLU”);
```

|| DAY 14 ||

do-while Loop

The do-while loop is like the while loop except that the condition is checked after evaluation of the body of the loop. Thus, the do-while loop is an example of an **exit-controlled loop**.

This loop runs at least once irrespective of the test condition, and at most as many times the test condition evaluates to true.

Syntax :

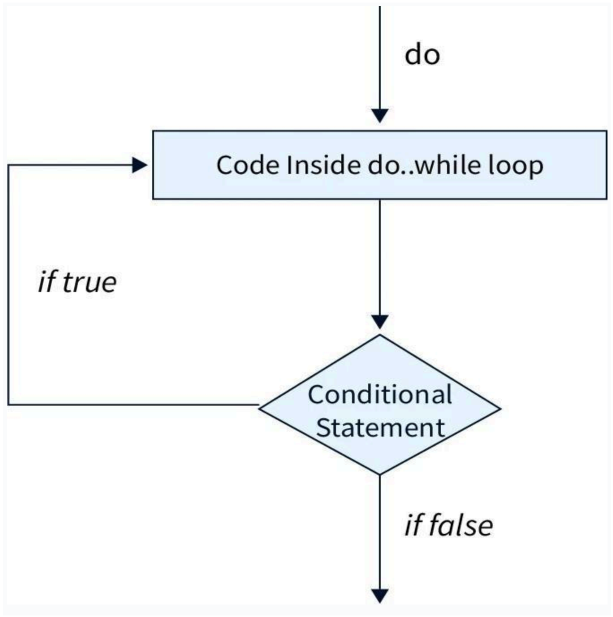
```
Initialization;
do {
// Body of the loop (Statement(s))
    // Updation;
}
while(Condition);
```

Example :

```
int i=1;
do {
    System.out.println("Hii");
i++;
}while (i<3);
```

Output :Hii Hii

FLOW DIAGRAM



The code inside the do while loop will be executed in the first step. Then after updating the loop variable, we will check the necessary condition; if the condition satisfies, the code inside the do while loop will be executed again. This will continue until the provided condition is not true.

Infinitive do-while Loop

```
int i = 0;
do{
i++;
}while(i> -1);
```

There will be no output for the above code also, the code will never end. Value of initialize to 0 then increment by 1 so it can never be -1 hence the loop will never end.

|| DAY 15 ||

Switch Statements

The **switch statement** is a control flow statement that allows you to select one of many code blocks to be executed based on the value of an expression.In simple words, the Java switch statement executes one statement from multiple conditions.

Syntax

```
switch(expression) {
case x:
// code block
break;
case y:
// code block
break;
default: // optional
        // code block to be executed if no cases match
}
```

Example

```
char ch = 'a';
switch (ch) {
      case 'a':
        System.out.println("Vowel");
        break;
    case 'e':
        System.out.println("Vowel");
        break;
    case 'i':
        System.out.println("Vowel");
        break;
    case 'o':
        System.out.println("Vowel");
        break;
    case 'u':
        System.out.println("Vowel");
        break;
    default:
        System.out.println("Consonant");
}
```

Output : Vowel

Important Points about Java's switch statement:

- **No variables:** The case value must be a literal or constant.
- **No duplicates:** No two cases should be of same value. Otherwise, a compilation error is thrown.
- **Allowed Types:** int, long, byte, short and String type. Primitives are allowed with their wrapper types.
- **Optional Break Statement:** Break statement is optional. If a case is matched and there is no break statement mentioned, subsequent cases are executed until a break statement or end of the switch statement is encountered (**fall through statement**).
- **Optional default case:** default case value is optional.
  The default statement is meant to execute when there is no match between the values of the variable and the cases. **It can be placed anywhere in the switch block** .

Multiple cases can be combined together with commas

```
char ch = 'a';
switch (ch) {
case 'a','e','i','o','u' :
        System.out.println("Vowel");
        break;
    default:
        System.out.println("Consonant");
}
```

Fall through statement

A fall-through statement occurs when there is no **break** statement at the end of a **case** block. When a **case** block does not have a **break** statement, the code execution continues to the next **case** block, even if the condition for that **case** is not met. This behavior is known as fall-through.

Example :

```
int number = 2;

switch (number) {
case 1:
        System.out.println("One");
case 2:
        System.out.println("Two");
case 3:
        System.out.println("Three");
default:
        System.out.println("Default");
}
```

Output : Two Three Default

It executed the code for **case 2**, then continued to **case 3**, and finally to the **default** block.

Arrow Switch

```
int number = 2;
switch (number) {
case 1 -> System.out.println("One");
    case 2 -> System.out.println("Two");
    case 3 -> System.out.println("Three");
    default -> System.out.println("Default");
}
```

It simplifies code and eliminates the need for explicit **break** statements.

yield keyword

yield keyword is used in combination with the new switch expression introduced in Java 12 to return a value from a **switch** expression.

It allows you to specify the value to be returned from a particular case block in the switch expression.

```
int dayOfWeek = 3;
String dayName = switch (dayOfWeek) {
case 1 :yield "Monday";
    case 2 : yield "Tuesday";
    case 3 : yield "Wednesday";
    case 4 : yield "Thursday";
    case 5 : yield "Friday";
    default : yield "Unknown";
};
System.out.println("Day of the week is: " + dayName);
```

Output : Day of the week is: Wednesday

|| DAY 16 ||

Nested Loops

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

for loops, **while** loops, and **do-while** loops, and you can nest any of these loop types inside one another.

```
for ( initialization; condition; increment ) {
    for ( initialization; condition; increment ) {
         // statement of inside loop
}
// statement of outer loop
}
```

Note: *There is no rule that a loop must be nested inside its own type. In fact, there can be any type of loop nested inside any type and to any level.*

|| DAY 17 ||

Array

An array is a linear data structure used to store a collection of elements of the same data type in contiguous memory locations.

Arrays in Java are **non-primitive** data types and it can store both primitive and non-primitive types of data in it.They are fixed in **size**, meaning that when you create an array you need to give specific size and you cannot change the size later.

Declaring an Array

```
DataType[] arrayName;
```

Creating an Array

After declaring an array, you need to create an actual instance of the array with a specific size using the **new** keyword. For example, to create an array of integers with a size of 5:

```
int[] numbers = new int[5];
```

size and initialization can't be done together

int[] arr = new int[3]{1, 2, 3}; // compilation err

```
// You can to this
```

```java
int[] arr = new int[]{1, 2, 3};
int[] arr = {1, 2, 3};
```

Stack memory holds the references while Heap memory holds the actual object:

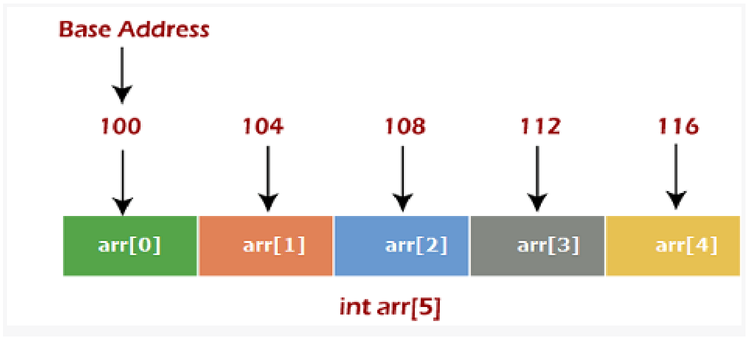
Stack: It is memory in which the size of the stack is limited and predefined during program execution. Exceeding this limit can result in a StackOverflowError(DTL).

Heap: The heap memory in Java can grow and shrink dynamically(DTL).

- Reference of the array is stored on the stack(int[] arr).
- Reference is essentially a memory address that points to the location in the heap where the actual array object is stored.

Address

Contiguous Elements: The elements of the array are stored in contiguous memory locations. This means that the memory addresses for each element are sequential. The memory address of the first element in the array is the base address of the array.



Internally the address is in hexadecimal number (combination of alphabets & numeric characters) this is just for your understanding (we take 100,104,108 etc).

In Java you cannot access the memory directly, you generally work with higher-level abstractions, and the specifics of memory addresses are hidden from you & handled by the Java Virtual Machine (JVM).

Elements in the array are accessed by their index. When you use an index to access an element, Java calculates the memory address of that element using the base address of the array and the size of the elements.

Address = Base address + (index * 4)

Address of 1st index = 100 + (1*4) = 104

Enhanced for loop || for-each loop

The for-each also called as enhanced for loop, was introduced in Java 5. It is one of the alternative approaches that is used for traversing arrays.

Traverse the array without using the index & makes the code simple as it reduces the code length.

Syntax:

```
for(datatype element : arrayName) {
    // Code
}
```

Example:

```
int arr[] = {1, 2, 3};
for (int elem : arr) {
    System.out.print(elem + ", ");
}
```

Output: 1, 2, 3

|| DAY 18 ||

Complexity

Complexity in algorithms refers to the amount of resources (such as time or memory) required to solve a problem or perform a task.

Algorithm: An algorithm is a well-defined sequential computational technique that accepts a value or a collection of values as input and produces the output(s) needed to solve a problem.

TIME COMPLEXITY

The time complexity of an algorithm/code is **not equal to the actual time** required to execute a particular code. You will get different timings on different machines.

Consider two computers: one a **supercomputer** and the other an **older machine**. When you run the same task on a supercomputer and an older computer, you might notice that it takes different amounts of time.

The time complexity of an algorithm is directly proportional to the size of the input. As the size of the input (denoted as "n") increases, the time taken by the algorithm also increases at a consistent and linear rate.

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Example 1 :

```
public class Demo {
    public static void main(String[] args) {
       System.out.print("Hello Duniya!!");
    }
}
```

Hello World" is printed only once on the screen.
So, the time complexity is **constant: O(1)**

Example 2:

```
int n = 5;
for (int i = 1; i <= n; i++) {
    System.out.println("Hello Duniya!!");
}
```

Hello World" is printed n times on the screen.
So, the time complexity is **constant: O(n)**

Complexity Representation

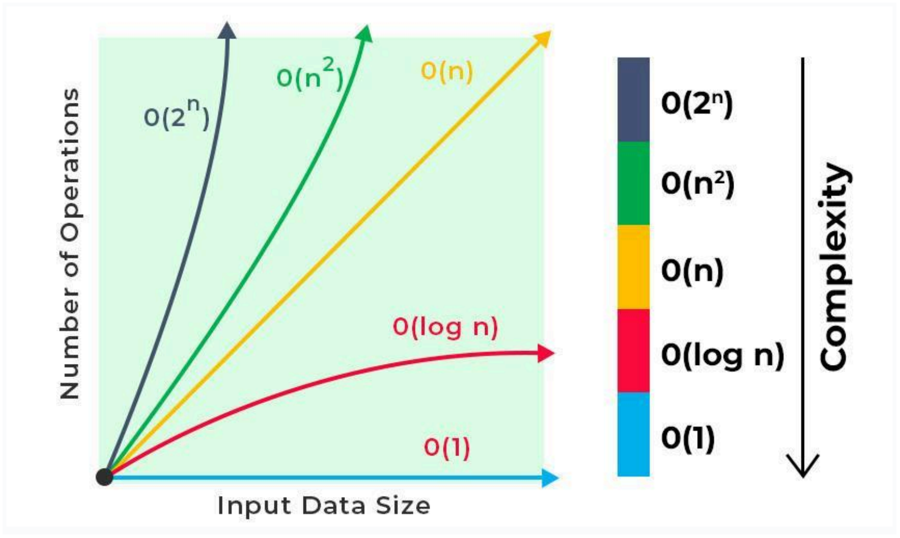
There are major 3 notations –

Big Oh - O(N) - Upper bound : The maximum amount of time required by an algorithm considering all input values. This is how we define the worst case of an algorithm's time complexity.

Big Omega - Ω(N) - Lower bound: The minimum amount of time required by an algorithm considering all input values is also the best case of an algorithm's time complexity.

Theta - θ(N) - Lower & Upper Bound: average bound of an algorithm. In this we know the algorithm will take exactly N steps.

Time complexity graph



We always check the worst time complexity or maximum amount of time required by an algorithm considering all input values.

There are different types of time complexities used

1. Constant time – O (1)
2. Linear time – O (n)
3. Logarithmic time – O (log n)
4. Quadratic time – O (n^2)
5. Cubic time – O (n^3)

Time Limit Exceed(TLE)

When the execution time of a program or algorithm exceeds the maximum time allowed for it to run.

Machine can perform 10^8 op / second

MAX value of N	Time complexity
10^9	O(logN) or Sqrt(N)
10^8	O(N) Border case
10^7	O(N) Might be accepted
10^6	O(N) Perfect

10^5	O(N * logN)
10^4	O(N ^ 2)
10^2	O(N ^ 3)
<= 160	O(N ^ 4)
<= 18	$O(2^N*N^2)$
<= 10	$O(N!), O(2^N)$

make your code within the upper bound limit or **constraints (limit)**.

Space Complexity

The space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space complexity is a parallel concept to time complexity. If we need to create an array of size n, this will require O(n) space. If we create a two-dimensional array of size n*n, this will require $O(n^2)$ space.

Space complexity can be categorized into different classes:

- **Constant Space (O(1)):**
    - Whether you have one local variable or 1000 local variables within a function, the space used remains constant because the number of variables doesn't depend on the input size.
- **Linear Space (O(n)):**

    - The space required grows linearly with the size of the input.
- **Logarithmic Space (O(log n)):**
    - The space required grows logarithmically with the size of the input.
- **Quadratic Space (O(n^2)), Cubic Space (O(n^3**

|| DAY 19 ||

Methods

- It is a block of code that performs a specific task.
- A method runs or executes only when it is called.
- Methods provide for easy modification and code reusability. It will get executed only when invoked/called.

Method Signature / Method Prototype / Method Definition

A method in Java has various attributes like **access modifier, return type, name, parameters** etc.

Methods can be declared using the following syntax:

```
accessModifier returnType methodName(parameters..){
   //logic of the function}
```

Example :

```
public static int sum(int a, int b)  {
   int sum = a+b;
   return sum;
}
```

Here, public - access modifier || static - special specifier int - return type

sum - method name || int a and int b - parameter

Access Modifiers

Access Modifier	Within Class	Within package	Subclass outside package	Outside package
Private	✅	❌	❌	❌
Protected	✅	✅	✅	❌
Default	✅	✅	❌	❌
Public	✅	✅	✅	✅

Methods mainly are of two type -

- Static
- Non-static

Static Method

- *A method declared as static does not need an object of the class to invoke it.*
- All the built-in methods are static - min, max, sqrt etc. called using Math class name

Example :

```
public class Demo {
   public static void main(String args[]){
       Demo.sum(1,2);//call by classname
   }
// static method
   public static int sum(int a, int b){
       int sum = a+b;
       return sum;
   }
}
```

Non-Static Method or Instance Method

- Non-Static Method or Instance methods are attached to the objects of a class, rather than the class itself.
- In simple words, you need to create an object to invoke them.

Example :

```
public class Demo {
   public void main(String args[]){
       Demo obj = new Demo();
       obj.sum(1,2);//call by object reference
   }
// static method
   public static int sum(int a, int b){
       int sum = a+b;
       return sum;
   }
}
```

|| DAY 20 ||

Arguments

An **argument** is a value passed to a function when the function is called.

A **parameter** is a variable used to define a particular value during a method definition.

In common we call both parameter and argument as either parameter/argument.

Classification of Arguments

***Formal* argument :** The identifier used in a method at the time of method definition.

Example:

```
returnType methodName(dataType parameterName1, dataType p2) {
  //  body
}
```

***Actual* argument :** The actual value that is passed into the method at the time of method calling.

Example:

```
methodName(argumentValue1, argumentValue2);
```

Arguments Passing

1. **Pass By Value:**

When we pass only the value part of a variable to a function as an argument, it is referred to as pass by value.

Any change to the value of a parameter in the called method does not affect its value in the calling method.

As can be seen in the figure below, only the value part of the variable is passed i.e. a copy of the existing variable is passed instead of passing the origin variable. Hence, any changes done to the value of the copy will not have any impact on the value of the original variable. Java supports pass-by-value.

Example:

```
public class Main{
  public static  void solve(int a){
     a = a+10; // changes inside called method
  }
  public static void main(String[] args) {
     int a = 10;
     System.out.println("Value of a "+ a);
     solve(a);
     System.out.println("Value of a "+ a);
  }
}
```

Output : Value of a 10
Value of a 10

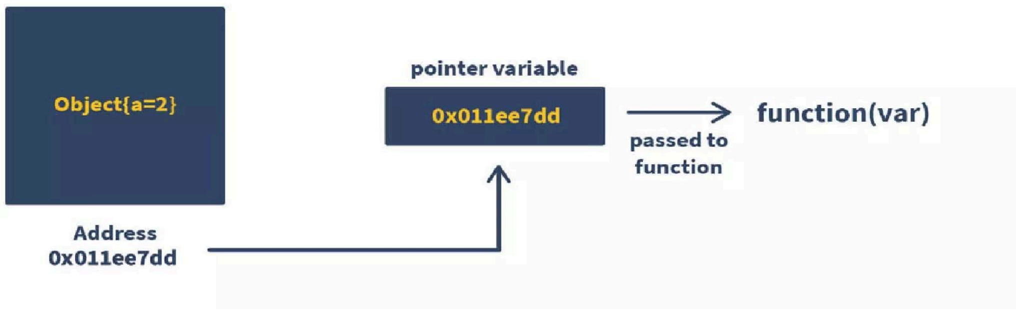
2. Pass by reference: Not supported by Java

In pass-by-reference, changes made to the parameters inside the method are also reflected outside. Though **Java does not support pass-by-reference.**

In Java, when we create a variable of class type or non primitive , the variable holds the reference to the object in the heap memory. This reference is stored in the stack memory. The method parameter that receives the object refers to the same object as that referred to by the argument.

Thus, changes to the properties of the object inside the method are reflected outside as well. This effectively means that objects are passed to methods by use of **call-by-reference**.

Changes to the properties of an object inside a method affect the original argument as well. However, if we change the object altogether, then the original object is not changed. Instead a new object is created in the heap memory and that object is assigned to the copied reference variable passed as argument.



Pass by Value for non-primitives

Example:

```
class PassByValue {
  public static void main(String[] args) {
    int[] array = new int[2];
    array[0] = 2;
    array[1] = 3;
    add(array);
    System.out.println("Result from main: " + (array[0] + array[1]));
  }
  private static void add(int[] array) {
    array[0] = 10;
    System.out.println("Result from method: " + (array[0] + array[1]));
  }
}
```

Output : Result from method: 13

```
            Result from main: 13
```

The called method is able to modify the original object but not replace it with another object.

Example:

```
class PassByValue {
  public static void main(String[] args) {
    Integer[] array = new Integer[2];
    array[0] = 2;
    array[1] = 3;
    add(array);
    System.out.println("Result from main: " + (array[0] + array[1]));
  }
  public static void add(Integer[] array) {
    array = new Integer[2];
    array[0] = 10;
    array[1] = 3;
    System.out.println("Result from method: " + (array[0] + array[1]));
  }
}
```

Output : Result from method: 13

```
Result from main: 5
```

Here, we can see that if we reinitialize the array object, which is passed in arguments, the original reference breaks(i.e., we have replaced the original object reference with some other object reference), and the array no longer is referenced to the original array. Hence the value in the main() method didn't change.

Points to Remember

- Java supports pass-by-value only.
- **Java doesn't support pass-by-reference**.
- Primitive data types and Immutable class objects strictly follow pass-by-value; hence can be safely passed to functions without any risk of modification.
- For non-primitive data types, Java sends a copy of the reference to the objects created in the heap memory.
- Any modification made to the referenced object inside a method will reflect changes in the original object.
- If the referenced object is replaced by any other object, any modification made further will not impact the original object.

Varargs (...)

Varargs also known as **variable argument**s is a method that takes input as a variable number of arguments.

The varargs method is implemented using a single dimension array internally. Hence, arguments can be differentiated using an index. A variable-length argument can be specified by using three-dot (...) or periods.

Syntax

```
public static void solve(int ... a){
    // method body
}
```

Rules

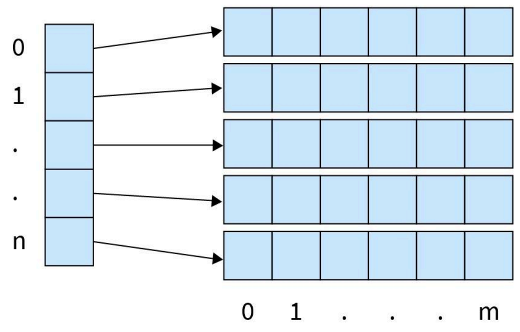
- There can be only one varargs in a method
- If there are other parameters then varargs must be declared in the last.

|| DAY 21 ||

Multi D Arrays

Multidimensional Arrays can be thought of as an array inside the array i.e. elements inside a multidimensional array are arrays themselves.

Multidimensional arrays, like a 2D array, are a bunch of 1D arrays put together in an array. A 3D array is like a bunch of 2D arrays put together in a 1D array, and so on.



Total Elements (n * m)

To access array elements in multidimensional arrays,more than one index is used.

General syntax to initialize the array:

```
arrayName = new DataTye[length 1][length 2]....[length N];

int[ ][ ] twodArray= new int[3][3];
```

|| DAY 22 ||

OOPS Introduction

OOPs is a programming paradigm (procedure or method)to solve real world problems.

It is a way of organizing and designing code to model real-world entities and their interactions. The main purpose of OOPs programming is to implement ideas and solve real-world problems.

OOPs (**Object Oriented Programming System**) refer to languages that use objects in programming.

Main pillars of OOPs:

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

Classes and **objects** are the building blocks of an object-oriented programming language.

Class:

- If we want to store different types of data we use class.
- Class is a user-defined or customized data type.
- Class is not a real world entity. It is just a template or blueprint or prototype of an object.
- Also, it doesn't occupy any memory.
- Class is a collection of objects.

For example – Animal, car, Birds etc. all are categories not a real world entity.

Syntax:

```
accessModifier class ClassName{
//attributes
//methods
}
```

Allowed Access Modifier in class

- Classes can have four different access modifiers:public, default, abstract and final (DTL).
- you can only use one of these modifiers at a time to define the accessibility and behavior of your classes.
- only one public class is allowed per file in Java, multiple non-public classes can coexist within the same file.

Object:

- Object is an instance(part) of a class.

- Object is a real world entity.
- Object occupies memory.

For example – Dog, cat (type of animal).

Verna, MG hector, Jeep (type of car).

Object consists of –

- Identity - Unique Name
- State | Attribute - color, breed, age [DOG] (represent by variable)
- Behavior – run, eat, bark etc [DOG] (represent by methods)

Syntax : –

```
className obj = new className();
```

The new keyword is responsible for allocating memory for objects.

Example :

```
Animal obj = new Animal();
```

Here, Animal () is a **constructor.**

Lets, understand this through the example

```
public class Student {
   String name;
   int age;
   String year;

   public void printInfo() {
       System.out.println("Student{ name='" + name + '\''
+ ", age=" + age + ", year='" + year + '\'' + '}');
   }
   public static void main(String[] args) {
       Student s1 = new Student();
       s1.name = "Golu";
       s1.age = 69;
```

```
        s1.year = "I";
        s1.printInfo();//call method
    }
}
```

Here, we initialize object by reference

If we want to initialize multiple variables then initializing by reference is not an efficient way.

Then we use a constructor.

Constructor:

- It is a special type of method.
- Called at the time of object creation.
- Responsible for initializing the object.
- It can be used to initialize the data member/ attribute of objects.

Rules –

- Same name as the class
- Never have any return type not even void.
- Cannot make static.
- Can have access modifiers, which control the visibility of the constructor.

Allowed access modifier in access modifier

- public, default, , private(when you don't want instances to be created outside the class), protected.
- Cannot be made final(can't override constructor makes no sense), abstract(must have an implementation and cannot be abstract).

Types –

1.Default|No-Arg Constructors|No Parameterized Constructor

- Do not have any arguments.
- Created by default in Java when no constructors are written by the programmer.

2. Parameterized Constructor

- Constructors with one or more arguments..
- It is possible to write multiple constructors for a single class.

For example

```
public Student() {
  String name;
  int age;

  Student(String stuName) {
    name = stuName;
  }
  Student(String stuName, String stuAge) {
    name = stuName;
    age  = stuAge;
  }
}
```

When you use the same name for data members (instance variables) and constructor parameters in a class, it can lead to **ambiguity** for the compiler.

In such cases, you can use the "**this**" keyword to clarify which variable you are referring to.

```java
public Student() {
  String name;
  int age;

  Student(String name) {
    this.name = name;
  }
  Student(String name, String age) {
    this.name = name;
    this.age  = age;
  }
}
```

this keyword helps the compiler understand whether you are working with the local parameter or the instance variable that shares the same name.

- Represent the current calling object.

|| DAY 23 ||

Overloading - Method, Constructor

Method Overloading

Create constructor having the same name but differ in the -

- Type(data type) of parameters.
- Number of parameters.
- Sequence or order of parameters.

Rules :

- Must change number, type, or order of parameters.
- Can change the return type.
- Can change the access modifier

Example :

```java
public class Sum{
   public int sum(int a, int b) {
      return (a + b);
   }
   public int sum(int a, int b, int c) {
      return (a + b + c);
   }
   public double sum(double a, double b) {
      return (a + b);
   }

 public static void main(String[] args) {
      Sum s = new Sum();
   }
}
```

This is how we achieve method overloading.

Constructor Overloading

Create constructor having the same name but differ in the -

- type(data type) of parameters
- number of parameters.
- Sequence or order of parameters.

Example :

```java
public class Sum{
    Sum(){
      System.out.println("Default");
    }
    Sum(int a, int b){
      System.out.println("2 Parameter");
    }
    Sum(int a){
      System.out.println("1 Parameter");
    }
```

```
   public static void main(String[] args) {
        Sum s = new Sum();
   }
}
```

This is how we achieve compile-time polymorphism i.e. Constructor Overloading.

Polymorphism

- Polymorphism is one of the main aspects of Object-Oriented Programming(OOP).
- “Poly” means many and “Morphs” means forms.
- The ability of a message to be represented in many forms.

Polymorphism is mainly divided into two types.

- Compile-time polymorphism
- Runtime polymorphism(DTL)

Compile-time polymorphism can be achieved by **method overloading**.

The decision of which method to call is made by the compiler at compile time based on the method's name and the number and types of its parameters.

It's also referred to as "early binding" or "static polymorphism"

Static in detail(DTL).

|| DAY 24 ||

String API

- If we need to store any name or a sequence of characters then we use String.
- String is an array of characters or sequence of characters.
- Java platform provides the String class to create strings.

Syntax

```
String  stdName =  "Pappu";
```

```
   ||         ||                  ||
```

DataType   variableName   Array of Characters

String str = "abc"; is equivalent to: char data[] = {'a', 'b', 'c'};

String is an array of characters. Let us see how to create string objects. String object can be created using two ways:

1. Using String Literal.
2. Using new keywords.

Using String Literal and String Constant Pool

A **literal**, in computer science, is a notation used for representing a value.

String literal can be created and represented using the double-quotes. All of the content/characters can be added in between the double quotes.

For example :

```java
String name = "apkasubhnaam";
```

Strings are stored in a special place in the heap called **"String Constant Pool"** or "String Pool".

String Constant Pool

- The string constant pool is a storage area in the heap memory that stores string literals.
- When a string is created, the JVM checks if the same value exists in the string pool.
- If it does, the reference to that existing object is returned. Otherwise, a new string object is created and added to the string pool, and its reference is returned.

```java
String str1 = "Pappu";
// New String is not created.
// str2 is pointing to the old string value only.
String str2 = "Pappu";
```

Using New Keyword

Strings can be created using the new keyword. When a string is created with new, a new object of the String class is created in the heap memory, outside the string constant pool.

Unlike string literals, these objects are allocated separate memory space in the heap, regardless of whether the same value already exists in the heap or not.

Syntax : `String str = new String("string_value");`

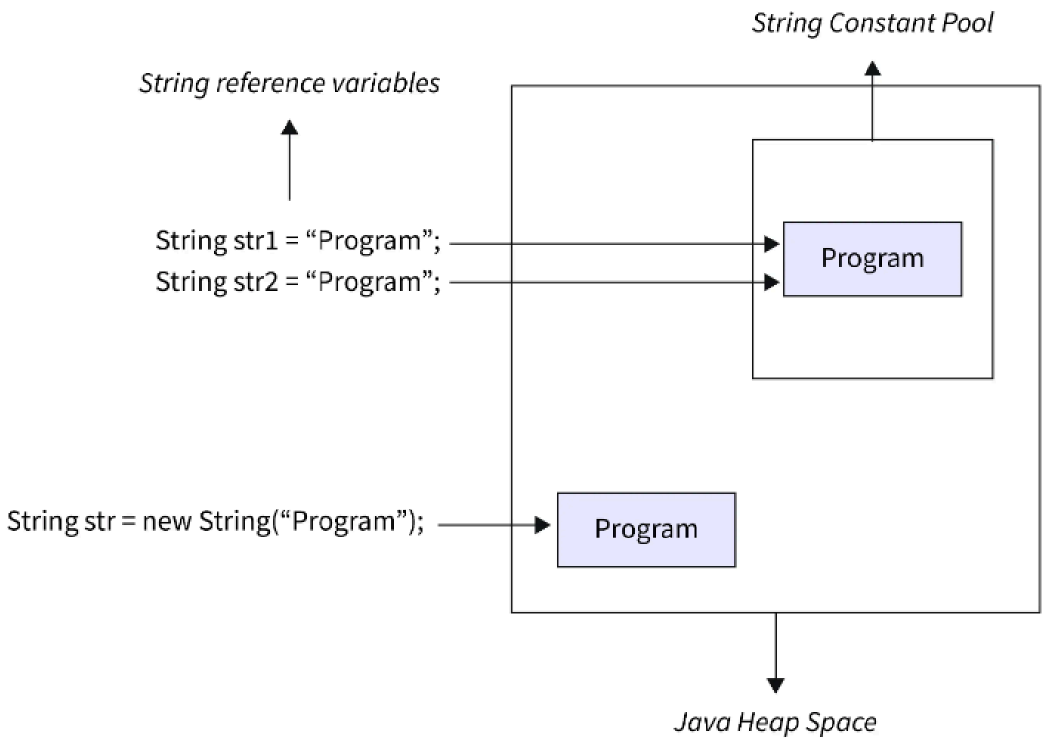
Example:

```
String str1 = new String("Pappu");
// New String is created.
// str2 is pointing to the new string value.
String str2 = new String("Pappu");
```

Let’s understand through example

```
String str1 = "Program";
String str2 = "Program";
String s = new String("Program");
```

In memory it stored like this -



Methods of Java Strings

length(), charAt(int index), , contains(), toUpperCase(), toLowerCase(), equals(), join() etc.

substring(int beginIndex, int endIndex[optional]) -

Example : s = “shery”

s.substring(0, 5) -> shery

s.substring(0) -> shery (behind the seen it works like s.substring(0,s.length())

s.substring(5) -> empty string [substring(5 , 5)]

For more methods : https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Comparing Strings

- Avoid using **==** (compare value and address both)
- Use **equals()**(it compares only value)
- **compareTo()** :[string1.compareTo(string2)] It returns a +ve integer if string1 is greater than string2, -ve if string2 is greater than string1, and zero if both are equal.

Java Strings: Mutable or Immutable

- Strings are **immutable**.
- Means their values cannot be changed once initialized.

Example:

```
String str = "Chacha";
System.out.println(str1);// Output: Chacha
str = str + “ and Chachi”;
System.out.println(str1);// Output: Chacha and Chachi
```

Above, when we concatenate a string " and Chachi" with str, a new string value is created. str then points to this newly created value, while the original value remains unchanged or the actual string value remains unchanged. This behavior demonstrates the immutability of strings.

|| DAY 25 ||

StringBuilder

StringBuilder in Java is an alternative to the String class.

It is used for storing the **mutable** (changeable) sequence which means we can update the elements of the StringBuilder class without creating a new StringBuilder sequence in memory.

Syntax

```
StringBuilder ob = new StringBuilder();
```

Default Capacity

Method - StringBuilder.Capacity()

- **16 bytes** is the default capacity of the StringBuilder when StringBuilder contains no elements.
- When the StringBuilder capacity gets full. Internally StringBuilder updates the capacity by (previous Capacity+1)*2.
- StringBuilder.length() and StringBuilder.capacity() are the two different methods.

Constructors of StringBuilder

StringBuilder()	-	Generates Empty String Builder with 16-character capacity.
StringBuilder(int capacity)	-	Empty String Builder with the provided length capacity.
StringBuilder(String)	-	The provided string is used to generate a String Builder. Length of string + 16 characters capacity.
StringBuilder(char)	-	Generates Empty String Builder with capacity char ASCII

Methods of StringBuilder

append(String s)	-	append the specified string to the provided string.
insert(int offset, String s)	-	insert the provided string at the designated place.
replace(int startIndex,int endIndex,String str)	-	string from the startIndex and endIndex values are replaced using this method.
delete(int startIndex, int endIndex)	-	delete the string from startIndex and endIndex that are provided.
reverse()	-	It is used to reverse the string.
capacity()	-	return the current capacity.
ensureCapacity(int cap)	-	make sure that the capacity is at least equal to the cap(input).
substring(int beginIndex)	-	substring starting at the beginIndex till end is returned using it.
substring(int beginIndex, int endIndex)	-	retrieve the substring starting at the beginIndex and endIndex values.

|| DAY 26 ||

Wrapper Classes

Wrapper classes in Java provide a way to represent the value of primitive data types as an object.

- In the Collection framework, Data Structures such as ArrayList store data only as objects and not the primitive types.
- As a result, Wrapper classes are needed as they wrap or represent the values of primitive data types as an object and its vice versa is also possible.

Below are the Primitive Data Types and their corresponding Wrapper classes:

Primitive Data Type	**Wrapper Class**
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Creating Wrapper Objects

Using the wrapper class and its constructor by passing a value to it

```
Integer number = new Integer(77);//int
Integer number2 = new Integer("77");//String
Float number3 = new Float(77.0);//double argument
Float number4 = new Float(77.0f);//float argument
Float number5 = new Float("77.0f");//String
Character c1 = new Character('S');//Only char
Character c2 = new Character(1234);//COMPILER ERROR
Boolean b = new Boolean(true);//value stored - true
```

This way of creating an instance of wrapper classes using constructor is deprecated as of the latest version of JDK.

Creating the object using the wrapper class.

```
Integer intValue = 10;
Double doubleValue = 8.89;
Character charValue = 'S';
```

Autoboxing

Autoboxing is when the compiler performs the automatic convert the primitive data types to the object of their corresponding wrapper classes.

For example, converting an *int* to *Integer*, a *double* to *Double*, etc.

```
int a = 10;
Integer intValue = a;
```

Unboxing

- It is just the opposite process of autoboxing.
- Unboxing is automatically converting an object of a wrapper type (Integer, for example) to its corresponding primitive (int) value.

```
Integer a=new Integer(5);
//Converting Integer to int explicitly
int first=a.intValue();
```

```
double first=a.doubleValue();
```

Useful Method & Parsing Strings

parseInt() - Returns an Integer type value of a specified String representation

```
int a = Integer.parseInt("69");//String to int
```

you can also parseDouble, parseLong, parseFloat etc.

valueOf() - Returns an Integer object holding the value of the specified primitive data type value.

```
String a = String.valueOf(7);//a = 7(String)
```

```
Integer a = Integer.valueOf("7");a = 7(Integer)
```

```
Double a = Double.valueOf("21");//a = 21.0
```

BufferedReader API

- BufferedReader & Scanner class both are sources that serve as ways of reading inputs.
- Scanner class is a simple text scanner that can parse primitive types and strings.
- BuffereReader reads text from a character-input stream, buffering characters so as to provide for the efficient reading of the sequence of characters.

Syntax :

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

```
BufferedReader reader = new BufferedReader(inpSReader);
```

We create a BufferedReader object through the newly created object of **InputStreamReader**(It reads bytes and decodes them into characters).

Difference Between Scanner And BufferedReader

BufferedReader :

- BufferedReader uses buffering to read a sequence of characters from a character-input stream.
- BufferedReader allows changing the size of the buffer.
- BufferedReader has a larger default buffer size (8 KB).

Scanner :

- Scanner can parse primitive types and strings using regular expressions.
- Scanner has a fixed buffer size
- Scanner has a smaller buffer size (1 KB)

Methods of Java BufferedReader

read() : Reads a single character.

readLine() : Reads a line of text. // recommended

To read input using BufferReader

```
public class ReadInput{

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

     BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

     String s = reader.readLine(); // read String

     int i = Integer.parseInt(reader.readLine());//read Int

     char i = reader.readLine().charAt(0);//read char

      boolean b = Boolean.parseBoolean(booleanVal);//read boolean
```

```
    }
}
```

|| DAY 27 ||

Collection Framework

Collection

- A collection refers to a group or assembly of things.
- In the context of programming, a group of objects or a single unit used to store multiple data is known as a collection.

Framework

- A framework is a set of classes and interfaces which provide a ready-made architecture.

Collection Framework

- It is an API which contains predefined classes and interfaces(DTL) which is used to store multiple data.
- It contains two main parts(packages)
    1. java.util.**Collection** - store data directly
    2. java.util.**Map** - store data in key-value pair e.g: rollNo - name

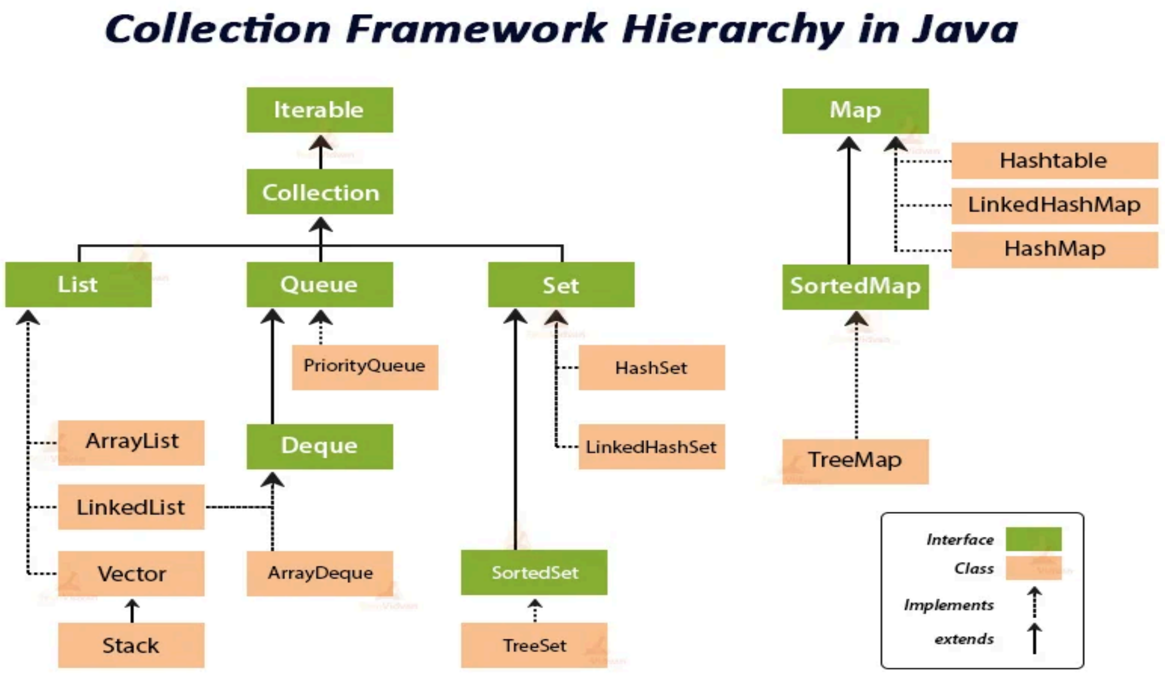
Collection VS Collections

Collection(Interface)

- It is a root interface(present in java.util package) of all the objects.

Collections(Utility class)

- It is a utility class which contains only static methods(invoked using the class name).



ArrayList

- ArrayList is an implemented class of List interface which is present in the java.util package.
- It is a growable and resizable array, used for creating dynamic arrays.

Important Features of ArrayList

1. **Dynamic Resizing :** resizable in nature
2. **Object based :** ArrayList can store only Objects data types. They cannot be used for primitive data types (int, float, etc). We require a wrapper class in that case.
3. **Ordered :** follows the insertion order.

4. **Index based :** supports random access from the list(using index positions starting with 0).

What is Generics

Generics are mainly used to impose type safety in programs. Type safety is when the compiler validates the datatype of constants, variables, and methods whether it is rightly assigned or not.

For example, we cannot initialize a variable as an integer and assign a string value to it. To avoid such mismatch between them, we can restrict the usage of a specific type of object only, like Integer, Float, etc.

We can initialize a List in two ways in Java.

1. By not specifying the datatype of the List

```
ArrayList list = new ArrayList();
```

- When an ArrayList is created, its default capacity or size is 10 if not provided by the user.
- When the array becomes full and we want to add new elements, a new ArrayList with a new capacity is created.
- The elements present in the old ArrayList are copied in the new ArrayList, while deleting the old ArrayList.

2. By using <>(angular brackets or generic) to specify the data type of the objects

```
ArrayList<Integer> list = new ArrayList();
```

Constructors in the ArrayList

1. ArrayList<>(): This constructor is the default one, it creates an empty ArrayList instance with default initial capacity i.e. 10.

Syntax : `ArrayList list = new ArrayList();`

2. ArrayList(int capacity): This constructor creates an empty ArrayList with initial capacity as mentioned by the user.

Syntax : `ArrayList list = new ArrayList(5);`

3. ArrayList(Collection list): This constructor creates an ArrayList and stores the elements that are present in the collection list.

Syntax :

```
ArrayList<String> list = Arrays.asList(new
String[]{"lab","labi"});
ArrayList<String> list1 = new ArrayList<>(list);
```

Methods in ArrayList

Method	Description
add(Object o)	Adds a specific element at the end of ArrayList.
add(int index, Object o)	Inserts a specific element into the list at the specified index
addAll(Collection c)	add all the elements present in collection at the end of list.
addAll(int index, Collection c)	add all the elements present in collection at the index.
remove(Object o)	Removes the first occurrence of the specified element from list
remove(int index)	Removes the element present at specified index from list.

removeAll(Collection c)	**Removes all the elements present in collection from the list.**
clear()	**Remove all the elements from the list.**
contains(Object o)	**Returns true if the list contains the specified element.**
get(int index)	**Returns the element at the specified index**
indexOf(Object o)	**Returns the index of the first occurrence of the specified element**
isEmpty()	**Returns true if the ArrayList contains no elements.**
lastIndexOf(Object o)	**Returns the index of the last occurrence of the specified element**
set(int index, Object o)	**Replace the element at the specified position in the list**
size()	**Returns the number of elements present in the ArrayList.**
toArray()	**Returns an array containing all the elements present in the list**

Example :

```
ArrayList<Integer> list = new ArrayList<>();
list.add(1); // [1]
list.add(1,2); // [1,2]
list.get(0); // 1
list.set(0,5);// [5,2]
list.remove(1);//if list contains Integer value then
remove always take index as a parameter in remove method
not object
list.indexOf(2); // 1
```

Time Complexities of key ArrayList operations:

- Random access takes O(1) time
- Adding element takes amortized constant time O(1)

- Inserting/Deleting takes O(n) time
- Searching takes O(n) time for an unsorted array and O(log n) for a sorted one.

|| DAY 28 ||

Hashing

Hashing is a technique or process that allows for efficient operations, such as insertion, deletion, and searching, with an average constant time complexity of O(1).

Why Hashing ?

The time complexity for searching in an ArrayList is O(n) in the worst case because it involves iterating through the list and the average time complexity for basic operations (insertion, deletion, and retrieval) is O(1) in Hashing.

Hence hashing allows efficient retrieval, Fast Lookups, optimized for Memory Usage.

HashMap and **HashSet** Key components of hashing.

HashMap

- HashMap is a data structure that implements the Map interface.
- It stores data in the form of key-value pairs.
- It uses hashing to efficiently store and retrieve key-value pairs.
- HashMap allows only one null key and multiple null values, but keys are unique (no duplicate keys are allowed).
- It does not maintain the insertion order.

- It provides constant-time average complexity for basic operations such as get, put, and remove.

Syntax :

```
HashMap<KeyDataType,ValueDataType> map = new HashMap<>();
```

Example :

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

Operations on HashMap

- Adding an element : **put(key, value)**
- Retrieve an element : **get(key)**
- Replace or Update : replace(key, newValue)
- Remove : **remove(key)**
- DeleteAll : clear()
- Check if map contain particular key: **containsKey(key)**
- Check if map contain particular value : containsValue(key)
- Check if map is empty : isEmpty()
- Size : **size()**

Example :

```
import java.util.HashMap;
public class Demo {
   public static void main(String[] args) {
     // Initialization of a HashMap
      HashMap<Integer, String> map = new HashMap<>();
       //Adding an element
        map.put(1,"Polu");
        map.put(2,"Laby");
        map.put(3,"Lab");

        //Retrieve using key
        map.get(1); // Polu

       //check value or key exist or not in map
       map.conatinsKey(1) //true
       map.containsValue(“Polu); // true

       //Update
        map.replace(1,"Lolu");
        //Remove
        map.remove(1);
    }
}
```

HashSet

- HashSet is a data structure that implements the Set interface.
- It does not allow duplicate elements.
- It uses hashing to efficiently store and retrieve key-value pairs.
- It can contain at most one null element.
- It does not maintain the insertion order.
- It provides constant-time average complexity for basic operations.

Syntax :

```
HashSet<dataType> set = new HashSet<>();
```

Example :

```
HashSet<Integer> set = new HashSet<>();
```

Operations on HashMap

- Adding an element : **add(data)**
- Remove : **remove(value)**

- DeleteAll : clear()
- Check if set contain particular key: **contains(value)**
- iterator over all the values : iterator()
- Size : **size()**

Example :

```
import java.util.HashSet;
public class Demo {
   public static void main(String[] args) {
     // Initialization of a HashMap
      HashSet<Integer> set = new HashSet<>();
      //Adding an element
      set.add(1);
      set.add(3);
      set.add(5);
      set.add(1);//duplicate set - [1,3,5]


      //Search
      set.contains(1); //true
      set.contains(2); //false

      //Remove
      set.remove(1);// [3,5]

      //Iterate on set
      Iterator itr = set.iterator();
      while (itr.hasNext()){
            System.out.println(itr.next());//print- 3,5
      }
      //remove all
      set.clear();//delete all elements
```

```
    }
}
```

|| DAY 28 ||

Getter and Setter

Getters are also called **accessors** and Setters are also called **mutators**. By using getter & setter we're indirectly accessing the private variables.

Example :

```
class Student {
   //  private variables
   private String name;
   private int rollno;

   // setter method to set the name
   public void setName(String n) {
       this.name = n;
   }
   // getter method to retrieve the name
   public String getName() {
       return name;
   }
   // setter method to set the roll number
   public void setRollno(int r) {
       this.rollno = r;
   }
   // getter method to retrieve the roll number
   public int getRollno() {
       return rollno;
   }
}
```

```
class Main {
   public static void main(String[] args) {
       // object of the class is created
       Student s1 = new Student();
       s1.setName("Labra");
       s1.setRollno(69);
       // printing the value returned by getName()
       System.out.println(s1.getName());
       // printing the value returned by getRollno()
       System.out.println(s1.getRollno());
   }
}
```

- In the above program, inside the "Student" class there are two private variables that are 'name' and 'rollno'.
- Directly we cannot access those variables outside the class which is an implementation of encapsulation or data hiding(DTL).
- So we are using **getters** as getName() and getRollno() methods to retrieve the values and **setters**.
- Inside setters, we used "**this** pointer" to access the data members of the current object it is pointing to.
- Here without directly accessing the variables, we are easily changing their values by just calling the setter functions and printing it by calling the getter functions.

toString() - method of the Object class returns the string representation of an object.

equals() - compares two objects based on elements present in both objects.

hashcode() - The hash value of an object is an integer value that is computed using the properties of that object.

Data members of a class must be private

- Making data members private allows you to control how external code interacts with the internal state of an object.
- You can provide controlled access to the data through **getter and setter methods**(mentioned above), enhancing the **security**.
- Through this we can **make the class read only**.

Array of Objects

- An array is a collection of **similar types** of elements.
- It consists of both **primitive (int, char, etc.**) data-type elements and **non - primitive (Object)** references of a class.

To create an array of objects, we use the class name which is followed by a square bracket([]), and then we also provide the variable name. Below given is the syntax for creating an array of objects.

Syntax : `ClassName obj[ ]= new ClassName[Array_Length];`

Example :

```
import java.util.Scanner;
class Students {
   //can make multiple variable
   String name;
   public Students(String name) {
       this.name = name;
   }
   @Override
   public String toString() {
       return "Student{" + "name='" + name + '\'' + ", age=" + age
+ '}';
   }
   public static void main(String[] args) {
       Students student[] = new Students[5];
       Scanner sc = new Scanner(System.in);
       for (int i = 0; i < student.length; i++) {
           System.out.println("Enter name = ");
           String name = sc.nextLine();
```

```
            Students obj = new Students(name);
            student[i] = obj;
        }
        for (Students obj : student){
            System.out.println(obj);
        }
    }
}
```

Lifetime of object and Garbage Collector

Step 1: Creation of .class file on disk.

Step 2: Loading .class file into memory.

Step 3: Looking for initialized static members of class.

Step 4: Allocation of memory for object and reference variable.

Step 5: Calling of the constructor of class.

Step 6: Removing object and reference variable from memory (Destructor).

|| DAY 29 ||

A static variable is associated with a class rather than an instance.

Static Variable

- Static variables are associated with the class rather than with the objects.

- Only a single copy of the static variable is created and shared among all the instances of the class.
- If an object modifies the value of a static variable, the change is reflected across all objects.
- Memory-efficient as they are not duplicated for each instance.
- We can only create static member variables at the class level, and cannot make static local variables.

Syntax : `static datatype variableName = Value;`

Example

Consider a machine that produces different varieties of pens like blue, black, green, etc. for a company '**X**'.

All pens are different in their properties, but one attribute is common among all the pens is its company name i.e **'X'**.

```
class Pen {
   String penType; // Type of pen whether gel or another
   String color;  // Color of the pen
   int length;  // Length of the pen
   static String companyName;  // static variable
}
```

The **companyName** variable is of **static** type which implies it is shared among all instances of the **Pen** class.

Example 2

```
class Country {
   // Static variable
   static int countryCounter;
   String name;
   int dummyCounter;
   public static void main(String[] args) {
       // Creating first instance
       Country ob1 = new Country();
       // Assigning values to object's data variables.
       ob1.name = "India";
       ob1.dummyCounter = 1;

       ++ob1.countryCounter;
       // Creating second instance of the class
       Country ob2 = new Country();
       ob2.name = "france";
       ob2.dummyCounter = 1;
       ++ob2.countryCounter;

       System.out.println("ob1.countryCounter = " +
ob1.countryCounter);
       System.out.println("ob2.countryCounter = " +
ob2.countryCounter);
       System.out.println("Country.countryCounter = " +
Country.countryCounter);
   }
}
```

Output :

```
ob1.countryCounter = 2

ob2.countryCounter = 2

Country.countryCounter = 2
```

This shows the variable is shared across all instances and it can be accessed using the class identifier as well.

Static Method

The static methods are the class methods and they don't belong to instances of the class. So , they are accessed by the class name of the particular class.

- It cannot access data that is dynamic (instance variables).
- Static methods can be accessed directly in non-static methods as well as static methods.
- The static methods can only access only other static members and methods.
- We cannot use **this** and **super(DTL)** keywords in the body of the static method (because this keyword is the reference of the current object but without creating an object we can access the static method this will cause an error).

Example for declaration of static method in Java:

```
class StaticMethod{
   static void print(){
       System.out.println("This is the static method");
   }
}
```

Static blocks

- A static block is a block that is associated with the static keyword.
- It is stored in the memory during the time of **class loading** and before the main method is executed, so the static block is executed before the main method. It runs once when the class is loaded into the memory.
- Static blocks executed before constructors.
- (JDK) version 1.5 or previous the static block can be executed successfully without the main() method inside the class. JDK version 1.5 or later will throw an error.

- Static block in java is used for changing the default value of static variables, initializing static variables of the class, writing a set of codes that you want to execute during the class loading in memory.

Example

```
public class Main {
   static {
       //static block
       System.out.println("Hi, I'm a Static Block!");
   }
   Main(){
         System.out.println("Hi, I'm a Constructor!");
   }
   public static void main(String[] args){
       //main method
      Maint1 = new Main();
      System.out.println("Hi, I'm a Main Method!");
   }
}
```

OUTPUT - Hi, I'm a Static Block!

Hi, I'm a Main Constructor!

Hi, I'm a Main Method!

Initializer block || Instance block

- Initializer block is used to initialize the instance variable or data member. Every time an object of the class is created, it runs.
- If we create 5 objects, the instance blocks will be executed 5 times

Example

```
class MyClass {

   // Instance block
   {
       System.out.println("Instance Block!");
   }

   // Static block
   static {
       System.out.println("Static Block!");
   }
   public static void main(String args[]) {
       MyClass obj1 = new MyClass();
       MyClass obj2 = new MyClass();

       System.out.println("Main Block!");
   }
}
```

Output : Static Block!

Instance Block!

Instance Block!

Main Block!

|| DAY 30 ||

Inheritance

- Inheritance is an object-oriented programming concept in which one class acquires the properties and behavior of another class.
- It represents a **parent-child relationship** between two classes.
- This parent-child relationship is also known as an **IS-A** relationship.

Let us discuss a real-world example

you are asked to build a scientific calculator, which does additional operations like power, logarithms, trigonometric operations, etc., along with standard operations(you have already written that code before).

Would you write the code for all standard operations, like addition, subtraction, etc., again? No, you have already written that code. Using inheritance in this case, you can achieve this objective.

Inheritance would give us **reusability.**

Inheritance concept

Subclass : A subclass, also known as child class or derived class, is the class that inherits the properties and behaviors of another class.

Superclass : A superclass, also known as parent class or base class, is the class whose properties and behaviors are inherited by the subclass.

Extends : The subclass uses the keyword to inherit the superclass

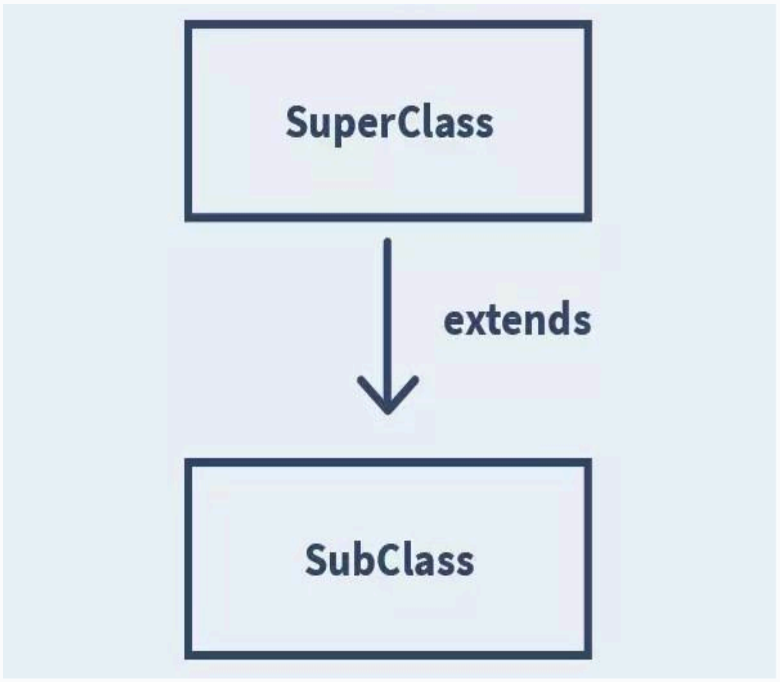
Types of Inheritance in Java

There are five different types of inheritances that are possible in Object-Oriented Programming.

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

1. Single Inheritance

This is the simplest form of inheritance, where one class inherits another class.



Ex: Below is a simple program that illustrates the Single Inheritance

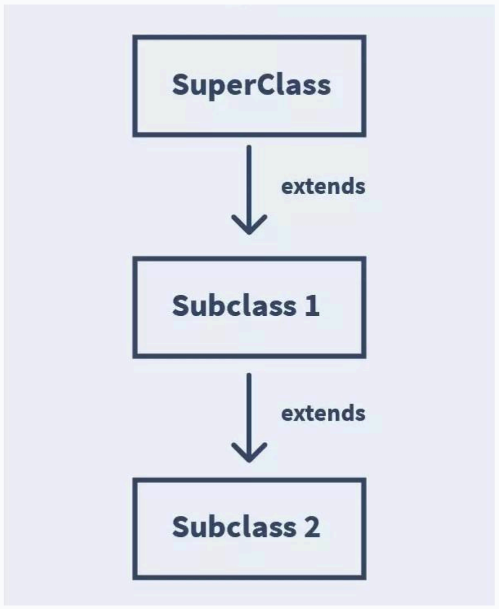
```
class Bird {
   void fly() {
       System.out.println("I am a Bird");
   }
}
// Inheriting SuperClass to SubClass
class Parrot extends Bird {
   void whatColourAmI() {
       System.out.println("I am green!");
   }
}
class Main {
   public static void main(String args[]) {
       Parrot obj = new Parrot();
       obj.whatColourAmI();
       obj.fly();
   }
}
```

Output : I am green!

I am a Bird

2. Multilevel Inheritance

This is an extension to single inheritance, where another class again inherits the subclass, which inherits the superclass.



Ex: Below is a simple program that illustrates the Multilevel Inheritance

```
class Bird {
   void fly() {
       System.out.println("I am a Bird");
   }
```

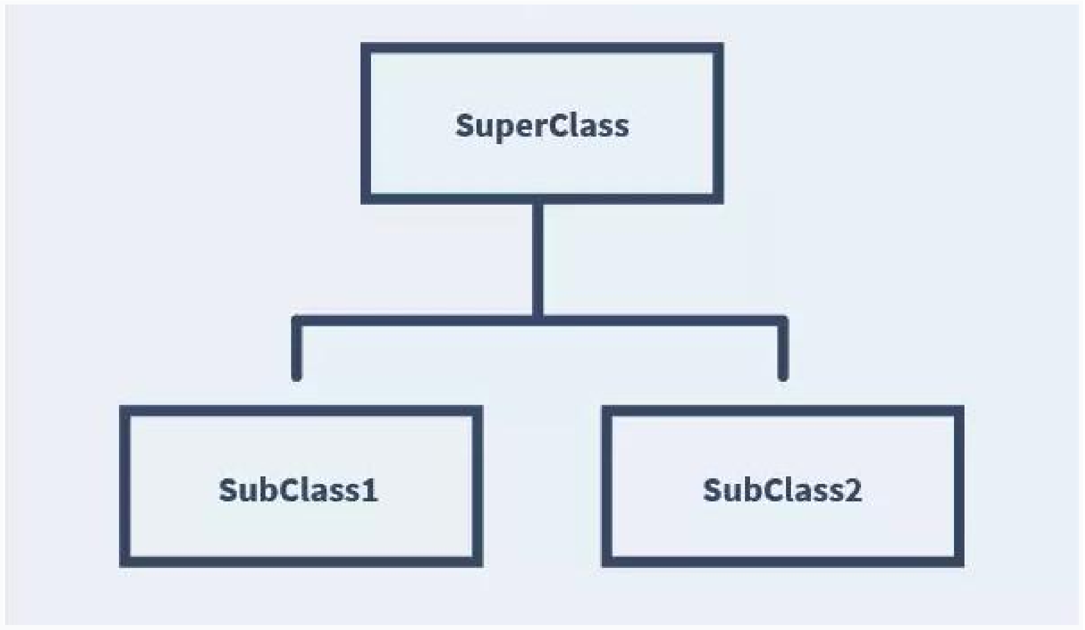
```java
}
// Inheriting class Bird
class Parrot extends Bird {
   void whatColourAmI() {
       System.out.println("I am green!");
   }
}
// Inheriting class Parrot
class SingingParrot extends Parrot {
   void whatCanISing() {
       System.out.println("I can sing Opera!");
   }
}
class Main {
   public static void main(String args[]) {
       SingingParrot obj = new SingingParrot();
       obj.whatCanISing();
       obj.whatColourAmI();
       obj.fly();
   }
}
```

Output : I can sing Opera! || I am green! || I am a Bird

- The object of SingingParrot created in the main method of java will be able to access the methods *whatCanISing(), whatColourAmI(), fly()* as they are all inherited by SingingParrot class using multilevel inheritance.

3. Hierarchical Inheritance

In Hierarchical inheritance, a single superclass is inherited separately by two or more subclasses.



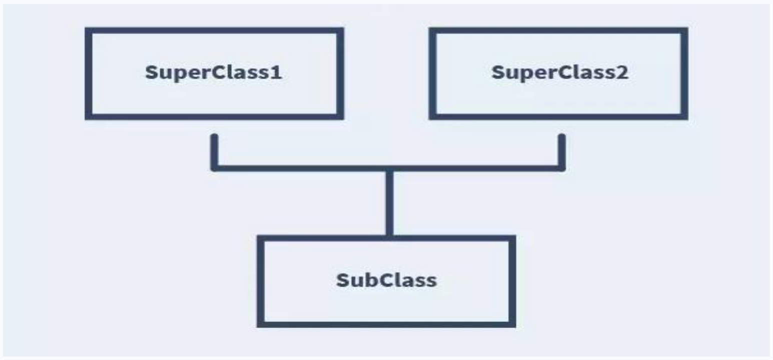
Ex: Below is a simple program that illustrates the Hierarchical Inheritance

```
class Bird {
   void fly() {
       System.out.println("I am a Bird");
   }
}
class Parrot extends Bird {
   void whatColourAmI() {
       System.out.println("I am green!");
   }
}
class Crow extends Bird {
   void whatColourAmI() {
       System.out.println("I am black!");
   }
}
class Main {
   public static void main(String args[]) {
       Parrot par = new Parrot();
       Crow cro = new Crow();
       //Call methods of Parrot Class
       par.whatColourAmI();
       par.fly();

       //Call methods of Crow Class
       cro.whatColourAmI();
       cro.fly();
   }
}
```

4. Multiple Inheritance

In Multiple Inheritance , a single class inherits from two different superclasses. Multiple Inheritance for classes is **Invalid** in Java.



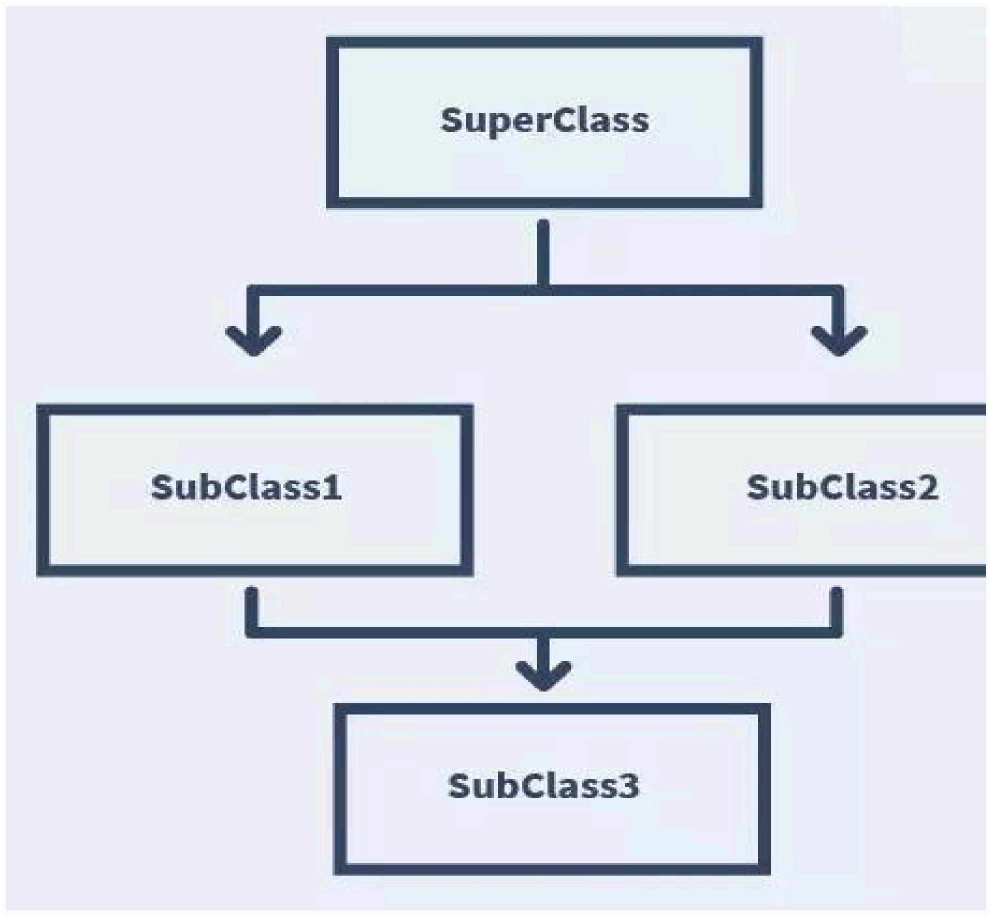
```
class A {
   void testMethod() {
       System.out.println("I am from class A");
   }
}
class B {
   void testMethod() {
       System.out.println("I am from class B");
   }
}
```

```java
// Not possible to inherit classes this way, But for understanding, let us
suppose
class C extends A, B {
   void newMethod() {
       System.out.println("I am from subclass");
   }
}
class Main {
   public static void main(String args[]) {
       C obj = new C();
       obj.testMethod();
       // Ambiguity here as it's present in both A and B class
   }
}
```

Note: The above program is just for understanding that multiple inheritance of classes is invalid in Java.

5. Hybrid Inheritance

Hybrid Inheritance is a combination of hierarchical inheritance and multiple inheritance. This is also not possible and **Invalid** in Java.

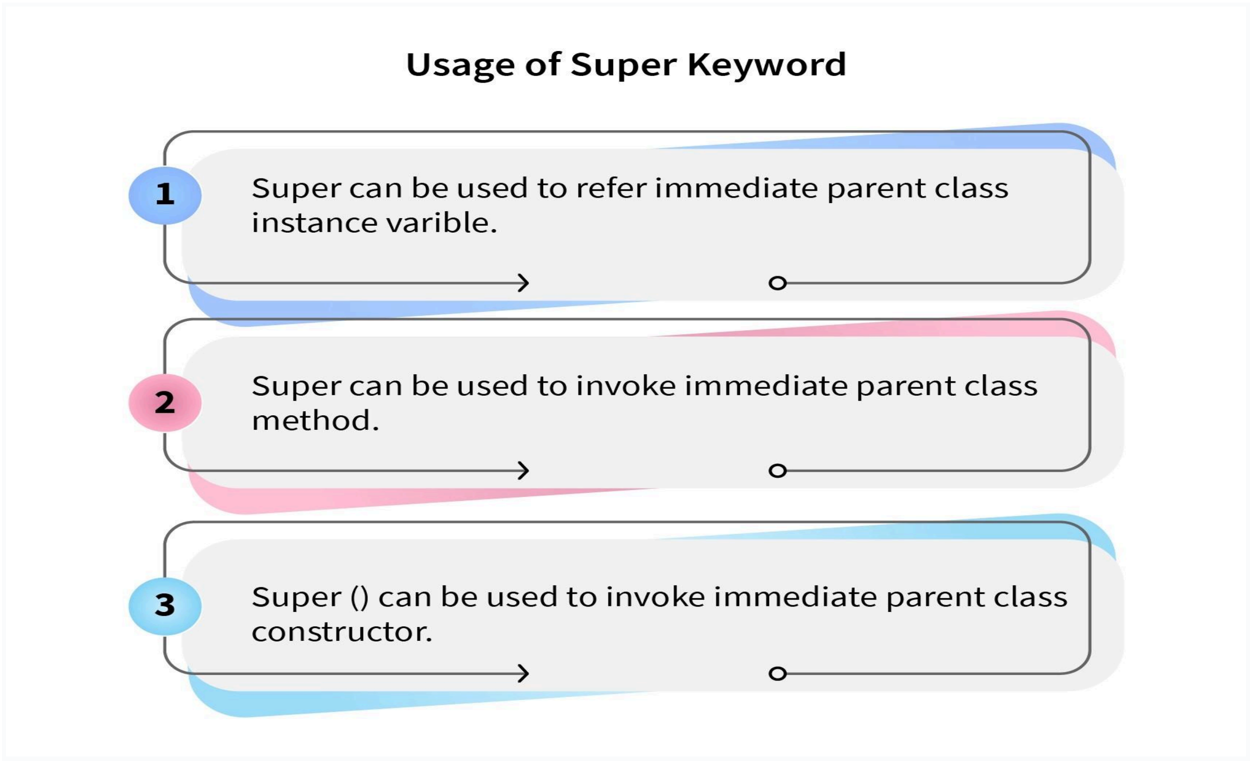


Hybrid Inheritance of classes is invalid in Java.

|| DAY 31 ||

Super and Constructor chaining

The super keyword is used to access the members of the **immediate parent class from child class.**


Usage of Super Keyword
1
Super can be used to refer immediate parent class instance varible.
2
Super can be used to invoke immediate parent class method.
3
Super () can be used to invoke immediate parent class constructor.

1. Refer immediate parent class instance variable

```
class Parent {
   String name = "Pappu";
}

class Child extends Parent {
   String name="Pappi";

   void printSchoolName() {
       //access the name variable of the parent class
       System.out.println("Name:  " + super.name);
   }
   public static void main(String[] args) {
       Child ob = new Child();
       ob.printSchoolName();
   }
}
```

Output : Name: Pappu

In the above example super keyword is used to reference name variables of the parent class.

2. Invoke immediate parent class method

```
class Parent {
   String name = "Pappu";
   void printParent() {
       System.out.println("name = " + name);
   }
}
class Child extends Parent {
   String name="Pappi";
   void printChild() {
       super.printParent();
       System.out.println("name = " + this.name);
   }
   public static void main(String[] args) {
       Child ob = new Child();
       ob.printChild();
   }
}
```

Output : name = Pappu | name = Pappi

3. Invoke immediate parent class constructor

This is useful when we want to initialize the instance members of the parent class from the child class.

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

```java
class Child extends Parent {
   Child() {
       super();// Invoke the parent class constructor
       System.out.println("Child class Constructor");
   }
}
class Main {
   public static void main(String[] args) {
       Child s = new Child();
   }
}
```

Call Parameterized Constructor Using super()

```java
class Parent {
   Parent(int value) {
       System.out.println("Parent constructor: "+value);
   }
}
class Child extends Parent {
   Child() {
       super(10);
       System.out.println("Child constructor");
   }
}


public class Main {
   public static void main(String[] args) {
       Child child = new Child();
   }
}
```

Output : Parent constructor with value: 10 | Child constructor

Note : If a subclass constructor doesn't explicitly invoke a superclass constructor using super(), the Java compiler **automatically** inserts a call to the **no-argument(default) constructor** of the superclass.

```
class Parent {
   Parent() {
       System.out.println("Parent constructor");
   }
}
class Child extends Parent {
   Child() {
       System.out.println("Child constructor");
   }
}
public class Main {
   public static void main(String[] args) {
       Child child = new Child();
   }
}
```

OUTPUT : Parent constructor | Child constructor

Constructor chaining : Constructor chaining is the process of invoking a series of constructors within the same class or by the child class's constructors.

We use **this()** method to call a constructor from another overloaded constructor in the same class, and the **super()** method to invoke an immediate parent class constructor.

Ways to Implement Java Constructor Chaining

There are two ways in which constructor chaining is performed, let's see the two ways below:

1. Constructor Chaining Within the Same Class:

```
class Employee{
   private String name;
   private int empID;
   Employee(){
       this("Lolu");
       System.out.println("Default");
   }
   Employee(String name){
       this(name, 69);
   }
   Employee(String name, int empID){
       this.name = name;
       this.empID = empID;
   }
   public String getName(){
       return name;
   }
   public int getEmpID(){
       return empID;
   }
}
class Main {
   public static void main(String args[]) {
       Employee employee = new Employee();
       System.out.println("Employee Name: " +
employee.getName());
       System.out.println("Employee ID: " +
employee.getEmpID());
   }
}
```

Output : Default | Employee Name: Lolu | Employee ID: 69

This series of invoking constructors from other constructors is known as **constructor chaining within the same class.**

2. Constructor Chaining from Parent/Base Class: The objective of a sub-class constructor is to invoke the parent class's constructor first. In Java, inheritance chains can contain any number of classes; every subclass constructor calls up the constructor of the immediate parent class (using the super() method) in a chain until it reaches the top-level class. Let's see

```
class Base {
   String name;
   Base(){
       this("");
       System.out.println("Default constructor base
class");
   }
   Base(String name){
       this.name = name;
       System.out.println("parameterized constructor base
class");
   }
}
class Derived extends Base {
   Derived() {
       System.out.println("Default constructor derived");
   }
   Derived(String name) {
       super(name);
       System.out.println("parameterized constructor of
derived");
   }
   public static void main(String args[]) {
       Derived obj = new Derived("test");
   }
}
```

OUTPUT : parameterized constructor base class parameterized constructor of derived

NOTE :

- **this() or super()** methods should always be the very **first statements** in a constructor definition to perform constructor chaining.
- A constructor can have a call to **super()** or **this()** but **never both.**
- Constructor chaining can be performed in **any sequence** and it helps in reducing duplicate code.

Relation between superclass and subclass objects and references.

- A superclass reference can be used for any of its subclass objects.
- but you cannot assign an object of the parent class to the reference of its subclass.

```
class Parent {
   Parent() {
       System.out.println("Parent constructor");
   }
}
class Child extends Parent {
   Child() {
       System.out.println("Child constructor");
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        //Object of Child class assign to Parent class
        Parent parent = new Child();

        //Object of Parent class assign to Child class
        Child child = new Parent(); // cannot possible
    }
}
```

Runtime Polymorphism or Method overriding

When you can find the same method with the same signature in inherited classes, it leads to method overriding.

The body of the class might differ in the overridden methods.

Example :

```
class PitaJi{
    //virtual method
    void display() {
        System.out.println("Pita Ji");
    }
}
class Putra1 extends PitaJi{
    //overriding display method
    void display() {
        System.out.println("Putra1");
    }
}
class Putra2 extends PitaJi{
    //overriding display method
    void display() {
        System.out.println("Putra2");
    }
}
class Main {
    public static void main (String[] args) {
        //Create an object of parent class
        PitaJi ob = new PitaJi();
        ob.display();

        Putra1 ob2 = new Putra1();
        ob2.display();
    }
}
```

In this case, the call to an overridden method will be resolved at the time of code execution (**runtime**) rather than the compile time.

Advantages

- The subclasses have the privilege to use the same method as their parent or define their specific implementation for the same method wherever necessary.
- Therefore, in one way, it supports code reusability when using the same method and implementation.

Runtime Polymorphism with Data Members

- Methods can be overridden but not variables.
- Therefore, runtime polymorphism isn't valid for data members (variables).

Let's see an example

```
class Cars{
    int speed = 60;
}
class CarA extends Cars{
   int speed = 218;
}
class CarB extends Cars{
   int speed = 221;
}
class Main {
   public static void main (String[] args) {
       Cars car = new Cars();
       System.out.println("Regular car: " + car.speed + "
mph");

       car=new CarA();
       System.out.println("Car A: " + car.speed + "
mph");

       car=new CarB();
       System.out.println("Car B: " + car.speed + "
mph");
   }
}
```

OUTPUT : Regular car: 60 mph
Car A: 60 mph
Car B: 60 mph

On accessing the value of the variable, it's clear that it doesn't change because **overriding doesn't work on variables** and it still refers to the variable of the parent class. So, the output remains 60.

|| DAY 32 ||

Access Modifiers

- Access modifiers provide a **restriction** on the scope of the class, instance variables, methods, and constructors.
- Used to control the visibility/access of instance variables, constructors, and class methods.

There are four types of Access modifiers:

1. Default, 2. Private, 3. Protected, and 4. Public

Private Access Modifier

You must be having some secrets that you don't want to disclose to anyone, even with your family members. This is the same behavior as the private modifier.

- Accessed only in the class itself.
- Even the other classes of the same package cannot access the private members or methods.

For example: In your Facebook account when you set the privacy of status to the **public** everyone on Facebook can access your status whether it is your friend, friend of your connection, or not a friend.

Let's understand the private modifier

```
class Fb{
   private int roll;
   private String name;
   Fb(int a) {
       System.out.print(a);
   }
   private Fb() {}
   private void print() {
       System.out.println("This is the Scaler class");
   }
}
```

```java
class Main {
   public static void main(String args[]) {
       //Creating the instance of the Fb class
       //This will successful run
       Fbob1 = new Fb(1);

       //Creating another instance of Fb class
       //This will cause an error
       Fb ob = new Fb();
       ob.name = "Aayush";
       ob.print();
   }
}
```

OUTPUT :java: Scaler() has private access in Scaler

java: name has private access in Scaler

java: print() has private access in Scaler

We got a lot of errors, but why?.

Because we are trying to access the private members, private methods in another class.

Public Access Modifier

- It provides no restriction on the methods, classes, and instance members of the particular class.
- When they are prefixed by the public modifier, they can be accessed in any package, in any class.
- It is very flexible as compared to the other modifiers.

For example: In your Facebook account when you change the privacy of your status to only for me, only you will be able to access the status, *and no one, even your friends, will not be able to see your status*..

Let's understand the public modifier:

First package:

```
package First;

public class Hello {
   //Instance member
   public int id = 1;
   //Class method
   public void print() {
       System.out.println("Hello first");
   }
}
```

Second package:

```
package Second;
import First.Hello;
public class Main {
   public static void main(String[] args) {
       Hello ob = new Hello();
       System.out.print(ob.id + " "); //print 1
       ob.print();
   }
}
```

We got the correct output, because we can access the **public** modifier's methods or instance variables in any class of the same or different package.

Default Access Modifier

- When the instance members, methods of the class are not specified with any modifier. It is said to be having a default modifier by default.
- The instance members and methods declared with the default modifier can be accessed only within the same package.
- Hence it is sometimes called package-private.

For example: In the Facebook account, when you set your privacy status to "visible to your friends only".You and your known friends can access your status. *Any other person will not be able to access your status info.*

Let's understand the default modifier:

First Package

```
package First;
public class Hello {
   int id = 1;
   void print() {
       System.out.println("Hello class");
   }
}
class Main {
   public static void main(String[] args) {
       Hello ob = new Hello();
       ob.print();
   }}
```

Output : Hello class

The program will run successfully because default members and methods can be accessed in the same package.

Second Package

```
package Second;
```

```
import First.Hello;
class Second{
   public static void main(String[] args) {
       Hello ob = new Hello();
       ob.print();
   }}
```

Output: `print() is not public in First.Hello; cannot be accessed from outside package`

Protected Access Modifier

- We can use protected modifier methods, instance members within the same package and access them in another package with the help of a child class.
- We have to extend the class that contains protected members in another package.
- We cannot access them directly by creating the object of a class that contains protected members and functions.

Let's understand the default modifier:

First Package

```
package First;

public class First {
   int id = 1;
   protected void print() {
       System.out.println("First class");
   }
}
```

Second Package

```
package Second;
import First.First;
```

```java
class Second extends First {
   public static void main(String[] args) {
       First ob = new First();
       //This line will cause an error
       ob.print();
       Second ob1 = new Second();
       //This line will not cause an error
       ob1.print();
   }
}
```

The line ob.print() **will cause an error** because we cannot access the protected method outside its own package, but we can access it by inheriting it in some other class of different packages, **that's why ob1.print() will not cause any error.**

Pictorial Explanation of Access Modifiers

Access Modifier	Same class	Same package subclass	Same package non-subclass	Different Package subclass	Different package non-subclass
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Protected	**Yes**	**Yes**	**Yes**	**Yes**	**No**
Public	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Final keyword

The final keyword is a non-access modifier that is used to define entities that cannot be changed or modified.

Consider the below table to understand where we can use the final keyword:

Type	**Description**
Final Variable	Variable with final keyword cannot be assigned again
Final Method	Method with final keyword cannot be overridden by its subclasses
Final Class	Class with final keywords cannot be extended or inherited from other classes

1. Final Variable

```
public class Main{
   public static void main(String[] args) {
       final int count = 10;
```

```
        // Again initializing final variable
        count = 15;//error
        final int myNumber = 10;
        int result = myNumber * 5;//you can do this
        System.out.println(result);
    }
}
```

count=15 compilation error final variable cannot be re-assigned.

Final Reference Variable

- Final variables referencing objects cannot be reassigned to point to another object.
- But you can modify the properties of the object they reference.
- This applies to class objects, arrays, and similar data structures.

Example

```
public class Main{
   public static void main(String[] args) {
       final StringBuffer strBuffer = new
StringBuffer("Hellow");
       System.out.println(strBuffer);

       // changing internal state of object reference by
       // final reference variable strBuffer
       strBuffer.append("World");
       System.out.println(strBuffer);
   }
}
```

Output : Hellow | HellowWorld

Final Fields or instance variable

- When a field is declared as static and final, it is referred to as a "constant”.
- Its value cannot be modified, even from another class.

Example - `static final int MAX_WIDTH = 999;`

We can initialize instance final fields -

- while declaration
- Inside the instance initializer block
- Inside the constructor

Final Method

Final keyword can be used with methods to prevent them from being overridden by subclasses.

Example

```
class Parent {
   // declaring method as final
   public final void display() {
       System.out.println("Parent");
   }
}
class Child extends Parent {
   // try to override final method
   public void display(){
       System.out.println("Child");
   }
}
```

Output : cannot override display() method is final

Final Class

- Likewise with the final variables and final methods, we can also use the final keyword with classes. These classes are called the final classes.
- Final classes can not be inherited by any other classes.
- If we try to inherit a class which is declared as a final, the system will throw a compile time error.

Example

```
// Final Class
public final class X {
   public void displayX() {
       System.out.println("Hellow World!");
   }
}
public class Y extends X {
   public void displayY() {
       System.out.println("Welcome!");
   }
}
```

Output : `cannot inherit from final X`

Note:

Declaring a class with a final keyword doesn't mean we can not change/modify the object of that class. We just can not inherit it.

|| DAY 33 ||

Abstraction

Abstraction in daily life, such as using a car, means benefiting from functionalities like brakes and engines without understanding their intricate workings. We enjoy the benefits without needing to understand all the complicated details, keeping things simple for us users.

Abstraction is used to hide the complexities of hardware and machine code from the programmer.

- It refers to hiding the implementation details of a code and exposing only the necessary information to the user.

Abstraction in Java can be achieved using the following tools it provides:

- Abstract classes
- Interfaces

Abstract Class

- An abstract class can be created without having any abstract methods.
- It can have final methods, which means methods that cannot be overridden by subclasses.
- Abstract classes are uninstantiable, implying that objects cannot be directly created from them.
- Abstract classes can have constructors, including parameterized constructors.

When Constructors are Called ?

- The constructor of an abstract class is called when an object of its child class is created.
- The child class constructor invokes the constructor of the abstract class using the super() keyword to ensure proper initialization of the inherited attributes.

Example

```java
abstract class Animal {
   private String name;
   public Animal(String name){
       this.name = name;
       System.out.println("An animal named " + name + "
is created.");
   }
   abstract void move();
   public void eat() {
       System.out.println(name + " is eating.");
   }
    public final void makeSound() {
       System.out.println(name + " makes a sound.");
   }
}
class Lion extends Animal {
    public Lion(String name) {
       super(name); // constructor of the abstract class
   }
   void move() {
       System.out.println("The lion is running.");
   }
}

public class Main {
   public static void main(String[] args) {
       Animal lion = new Lion("Simba");
       lion.eat();
```

```
        lion.makeSound();
        lion.move();
    }
}
```
OUTPUT : An animal named Simba is created.

Simba is eating.

Simba makes a sound.

The lion is running.

When an object of the *Lion* class is created, the constructor chain is triggered, starting with the constructor of the abstract class (*Animal*).

Abstract Method

- An abstract method is a method declared **without** an **implementation(semi-implemented)** in an abstract class**.**
- Abstract methods do not have a method body. They end with a semicolon (;) instead of curly braces.
- The presence of an abstract method in a class forces that **class** to be declared as abstract.
- A **derived class** of an abstract class must either **override** all its **abstract methods** or be declared abstract itself.
- Abstract methods **cannot** be declared as **private** because they need to be accessible to subclasses for overriding(private is class level access modifier).
- **Static methods**, being part of the class and not the object's state, cannot be overridden in the traditional sense.

Example

```java
abstract class Animal {
   private String name;
   public Animal(String name) {
       this.name = name;
   }
   abstract void makeSound();

   public void displayInfo() {
       System.out.println("Name: " + name);
   }
}
class Dog extends Animal {
   public Dog(String name) {
       super(name);
   }
   void makeSound() {
       System.out.println("Woof! Woof!");
   }
}
class Cat extends Animal {
   public Cat(String name) {
       super(name);
   }
   @Override
   void makeSound() {
       System.out.println("Meow!");
   }
}

public class Main {
   public static void main(String[] args) {
       // Creating objects of concrete subclasses
       Dog myDog = new Dog("Buddy");
       Cat myCat = new Cat("Whiskers");
```

```
        // Calling abstract method and regular method for
Dog
        myDog.displayInfo();
        myDog.makeSound();

        // Calling abstract method and regular method for
Cat
        myCat.displayInfo();
        myCat.makeSound();
    }
}
```

OUTPUT : Name: Buddy

Woof! Woof!

Name: Whiskers

Meow!

- **The** Animal **class is an abstract class with an abstract method** makeSound **and a regular method** displayInfo**.**
- **The** Dog **and** Cat **classes are concrete subclasses that extend** Animal **and provide specific implementations for the abstract method.**

NOTE : We **cannot** achieve **100% of abstraction** using **abstract class** because it contains concrete(non-abstract) methods. To achieve 100% abstraction we use **interfaces**.

Abstract classes can be used to achieve anything **between 0–100%** abstraction.

|| DAY 34 ||

Interface

- An interface is a blueprint of a class and contains abstract methods.
- Methods are public and abstract by default(you don't have to explicitly use the "abstract" keyword).
- Any class implementing your interface will need to provide implementations of those methods.

Example

```
interface Car {
   public void start();
}
```

Methods and variables inside interface

Methods declared inside an interface are implicitly marked as public and abstract, and variables declared inside an interface are implicitly marked as public static final by the compiler.

Inherit using implement

- To use an interface, a class must implement it using the **implements** keyword and override all the methods declared in the interface.
- We **cannot** create **objects** of an interface, it can be used to reference a class that implements the interface.

Example

```java
class DieselCar implements Car{
   public void start(){
       System.out.println("Diesel Car
starting...Vrooooom!");
   }
}
public class Demo {
   public static void main(String[] args) {
       Car mercedes = new DieselCar();
   }
}
```

We can extends and implements at the same time

Implements will come after extends because any number of interfaces can be implemented but only one class can be inherited

Let's consider example where a class **extends** another class and **implements multiple** interfaces:

```java
interface Walkable {
   void walk();
}
interface Swimmable {
   void swim();
}

class Animal {
   void eat() {
       System.out.println("Animal is eating");
   }
}
class Dog extends Animal implements Walkable, Swimmable {
   public void walk() {
       System.out.println("Dog is walking");
```

```java
    }
    public void swim() {
        System.out.println("Dog is swimming");
    }
// This class can also have its own methods
    void bark() {
        System.out.println("Woof! Woof!");
    }
}
public class Demo {
    public static void main(String[] args) {
        // Create an instance of the derived class
        Dog myDog = new Dog();
        myDog.eat();
        myDog.walk();
        myDog.swim();
        myDog.bark();
    }
}
```

"In the above example, the class Dog inherits from two parent interfaces, indicating that multiple inheritance can be achieved using interfaces."

Inheritance between class-class, class-interface, interface-interface

1.Class-Class Inheritance (using extends):

```java
class ParentClass {
    // ...
}

class ChildClass extends ParentClass {
    // ...
}
```

2.Class-Interface Inheritance (using implements):

```
interface MyInterface {
   void myMethod();
}
class MyClass implements MyInterface {
   public void myMethod() {
//Implementation of the method specified in the interface
   }
}
```

3.Interface-Interface Inheritance (using extends):

```
interface BaseInterface {
   void baseMethod();
}

interface ExtendedInterface extends BaseInterface {
   void extendedMethod();
}
```

Important Points

- An interface can be used when we want to achieve 100% abstraction. On the other hand, abstract classes can be used to achieve anything between 0–100% abstraction.
- An interface cannot have constructors because we cannot create objects of an interface.
- If you want a class to achieve multiple inheritances, there is only one way: interfaces.
- If an interface is made private, or if the methods in it are made private or protected, then a compilation error will be thrown.

|| DAY 35 ||

Exception Handling

Exception

Exceptions are unwanted conditions that disrupt the flow of the program. For example, if a user is trying to divide an integer by 0 then it is an exception, as it is not possible mathematically.

- Exceptions usually occur due to the code and can be recovered.
- They can occur at both runtime and compile time.
- Exceptions belong to java.lang.Exception class.

Let's understand with the following example

```
class SampleCode {
   public static void main(String args[]) {
       System.out.println("Hello World!");
       int a = 10;
       int b = 0;
       System.out.println(a / b);
       System.out.println("Welcome to java 
programming.");
       System.out.println("Bye.");
   }
}
```

Output : Hello World!

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Demo.main(Demo.java:8)
```

In the above example, an integer is divided by 0, which is not possible. Exceptions are not handled by the programmer which will halt the program in between by throwing the exception, and the rest of the lines of code won't be executed.

Error

- An error is also an unwanted condition but it is caused due to lack of resources and indicates a serious problem.
- Errors are irrecoverable, they cannot be handled by the programmers.
- They can occur only at run time.
- In java, errors belong to the java.lang.error class.

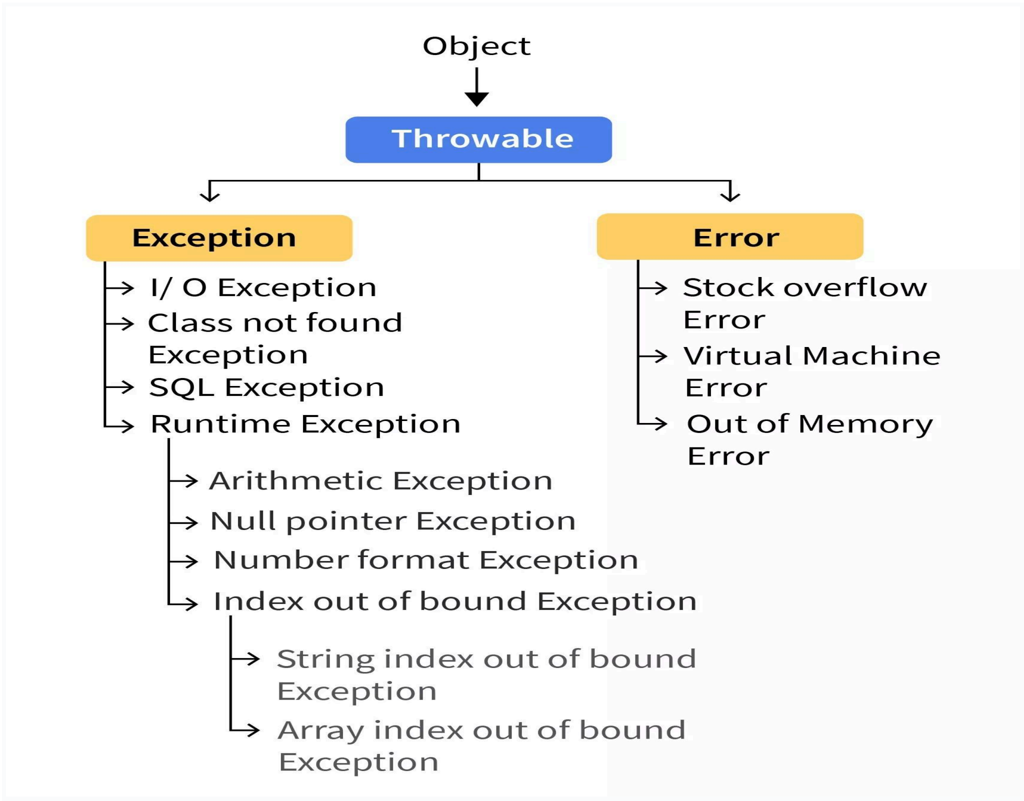
Exception Handling

Exception handling is a mechanism to handle unwanted interruptions like exceptions and continue with the normal flow of the program.

We handle exceptions to make sure the program executes properly without any halt, which occurs when an exception is raised.

We use try-catch blocks and other keywords like finally, throw, and throws to handle exceptions.

Hierarchy of Java Exception Classes : As java is an object-oriented language every class extends the Object class. All exceptions and errors are subclasses of class **Throwable**.



Types of Exceptions Handling

1. Checked Exceptions

- Checked exceptions are those exceptions that are checked at compile time by the compiler.
- The program will not compile if they are not handled.
- These exceptions are child classes of the Exception class.
- IOException, ClassNotFoundException, InvocationTargetException, and SQL Exception are a few of the checked exceptions in Java.

2. Unchecked Exceptions

- Unchecked exceptions are those exceptions that are checked at run time by JVM, as the compiler cannot check unchecked exceptions,

- The programs with unchecked exceptions get compiled successfully but they give runtime errors if not handled.
- These are child classes of Runtime Exception Class.
- ArithmeticException, NullPointerException, NumberFormatException, IndexOutOfBoundException are a few of the unchecked exceptions in Java.

How do we handle exceptions?

Customized exception handling is achieved using five keywords: **try, catch, throw, throws, and finally**.

1. try block -

- try block is used to execute doubtful statements which can throw exceptions.
- try block can have multiple statements.
- Try block cannot be executed on itself, there has to be at least one catch block or finally block with a try block.
- When any exception occurs in a try block, the appropriate exception object will be redirected to the catch block, this catch block will handle the exception according to statements in it and continue the further execution.
- The control of execution goes from the try block to the catch block once an exception occurs.

Syntax :

```
try{
   //doubtful statement
}
```

2. catch block

- catch block is used to give a solution or alternative for an exception.
- catch block is used to handle the exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception or the generated exception type in the exception class hierarchy or a user-defined exception.
- You can use multiple catch blocks with a single try block.

Syntax :

```
try {
   // Code that can raise exception
   int div = 509 / 0;
} catch (ArithmeticException e) {
   System.out.println(e);
}
```

3. try-catch block , Multiple Catch Blocks, Nested Try Catch

- Java can have a single try block and multiple catch blocks and a relevant catch block gets executed.
- we can have deep (two-level) nesting which means we have a try-catch block inside a nested try block

Example

```
try {
   int a[] = new int[5];
   System.out.println(a[10]);
} catch (ArithmeticException e) {
   System.out.println("Arithmetic Exception occurs");
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("ArrayIndexOutOfBounds Exception
occurs");
} catch (Exception e) {
   System.out.println("Parent Exception occurs");
}
System.out.println("rest of the code");
```

4. finally block

- finally block is associated with a try, catch block.
- It is executed every time irrespective of whether an exception is thrown or not.
- finally block is used to execute important statements such as closing statements, release the resources, and release memory also.
- finally block can be used with try block with or without catch block.

Example :

```
try {
   int data = 100/0;
   System.out.println(data);
} catch (Exception e) {
   System.out.println("Can't divide integer by 0!");
} finally {
   System.out.println("finally block");
}
```

5. throw keyword

- throw keyword is used to throw an exception explicitly.
- We can throw checked as well as unchecked exceptions (compile-time and runtime) using it.

Example:

```java
static void checkAge(int age) {
   if (age < 18) {
       throw new ArithmeticException("Access denied");
   } else {
       System.out.println("Access granted");
   }
}
public static void main(String[] args) {
   checkAge(15);
}
```

6. throws keyword

- throws keyword is used in the method signature to indicate that this method might throw one of the exceptions from the Java exception class hierarchy.
- throws keyword is used only for checked exceptions like IOException as using it with unchecked exceptions is meaningless(Unchecked exceptions can be avoided by correcting the programming mistakes.)
- throws keyword can be used to declare an exception.

Example :

```java
public class Main {
 static void checkAge(int age) throws ArithmeticException
{
       if (age < 18) {
         throw new ArithmeticException("Access denied");
       } else {
          System.out.println("Access granted");
       }
 }
```

```
   public static void main(String[] args) {
       checkAge(15);
   }
}
```

Common Scenarios of Exceptions

1. ArithmeticException : we try to perform any arithmetic operation which is not possible in mathematics

2. NullPointerException : when a user tries to access a variable that stores null values.

3. NumberFormatException : variable with an incompatible data type

4. ArrayIndexOutOfBoundsException : accessing an index that is not present.

5. StringIndexOutOfBoundsException : if the length of a string is less than what we are trying to access.

Different ways to print information about exceptions

1. obj.toString(): returns a string representation of the exception.

```
try {
   // some code that may throw an exception
} catch (Exception e) {
   System.out.println(e.toString());
}
```

2. obj.printStackTrace(): print the stack trace of an exception,where the exception occurred in the code.

```
try {
   // some code that may throw an exception
} catch (Exception e) {
   e.printStackTrace();
}
```

3. obj.getMessage(): retrieves the error message associated with the exception.

```
try {
   // some code that may throw an exception
} catch (Exception e) {
   System.out.println(e.getMessage());
}
```

|| DAY 36 ||

File API

- File handling refers to the process of manipulating files and directories, meaning reading and writing data to a file.
- With the help of File Class, we can work with files. This File Class is inside the java.io package.

File Class: How to create a file

- File class can be used by creating an object of the class.
- It provides methods to create, delete, and inspect properties of files and directories.

Syntax : `File folder = new File("<path_of file>");`

Example

```
String path = "C:\\Users\\pc\\Desktop\\";
File folder = new File(path + "temp");//temp is directory name
folder.mkdir();//make directory or folder
```

Create a new file : We use the createNewFile() method to attempt to create a new file in combination with exception handling.

Make sure to handle IOException appropriately, as file operations may throw this exception in case of errors (e.g., insufficient permissions, disk full, etc.).

Example:

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

public class Main {
   public static void main(String[] args) throws IOException {
       String path = "C:\\Users\\pc\\Desktop\\";
       File file = new File(path + "folder" + "\\file.txt");
       file.createNewFile();
   }
}
```

Writing text in file

FileWriter class is a subclass of the Writer class and is used for writing characters to a file.

- Useful when you want to write character data to a file as text.
- The class provides various constructors and methods for efficient file writing operations.

1. Constructor: **FileWriter(File reference)** : Creates a new FileWriter given a File object. This constructor establishes a connection to the file specified by the File reference for writing.

```
FileWriter writer = new FileWriter(fileName);
```

2. Methods:

a. `void write(int c)`

- Description: Writes a single character to the file.
- `writer.write('A'); // Writes the character 'A' to the file`

b. `void write(char[] chArr)`

- Description: Writes an array of characters to the file.

```
char[] data = {'H', 'e', 'l', 'l', 'o'};
writer.write(data); // Writes the string "Hello" to the
file
```

c. `void write(String str)`

- Description: Writes a string to the file.
- `writer.write("data");`

d. `void flush()`

- Description: Flushes the stream, ensuring that any buffered data is written to the file.
- `writer.flush(); // Flushes the stream`

e. `void close()`

- Description: Flushes the stream and closes the `FileWriter`. After closing, further writing operations on the writer may result in an `IOException`.
- `writer.close(); // Flushes the stream and closes the FileWriter`

Always remember to close the FileWriter after writing to ensure proper resource management.

Example:

```
class FileWriterExample {
   public static void main(String[] args)throws IOException{
       File file = new File("example.txt");
       FileWriter writer = new FileWriter(file);

        writer.write("Aur bhai kaise ho ??");
        writer.close(); // Close the writer when done
   }
}
```

File class Methods

mkdir() : Creates a new directory using the pathname. If a directory is created, the method returns true; otherwise, false.

createnewfile() : Generates a new file with no content. Method returns true, and a new file is created. If the filename already exists, it returns false.

delete() : Deletes the file or directory. The method returns true if the file or directory deleted, otherwise returns false. Syntax : boolean var = file.delete();

getname() : Name of the file is returned.
Syntax : String var= file.getName();

getabsolutepath() : The absolute pathname of the file is returned.
Syntax : String var= file.getabsolutepath();

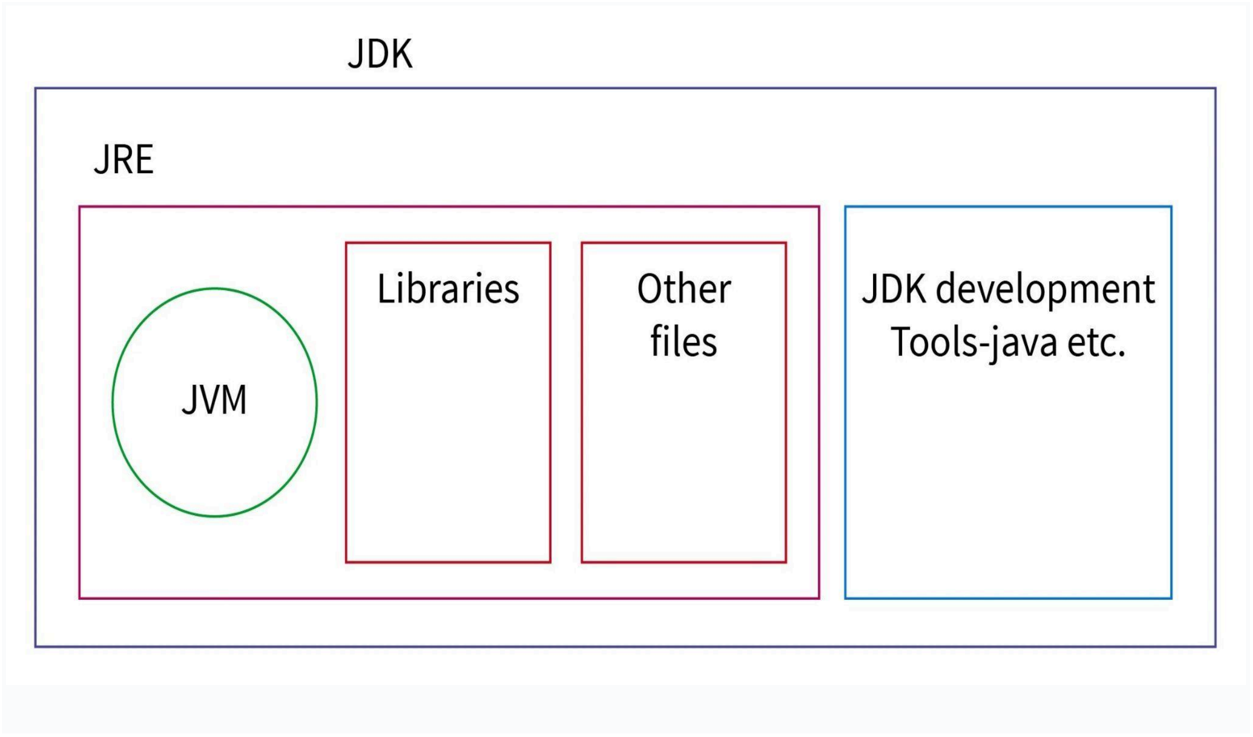
list() : Returns lists the names of files and folders in a given directory.
Syntax : String[] paths= file.list();

exists() : If the abstract file path exists, the method returns true; otherwise, it returns false. Syntax : boolean var = file.exists();

|| DAY 37 ||

Java Development Kit(JDK)

- As the name suggests, the Java Development kit is used for developing Java applications and applets.
- It consists of Java Runtime Environment (JRE), interpreter, compiler, and other tools like archiver, document generator, etc.
- It is responsible for compiling, debugging, and executing.
- It is platform-dependent as a different JDK is required for every platform.
- It provides different tools and libraries for building frameworks, Android applications, desktop applications, etc.
- Some of the popular JDK include- OpenJDK and Oracle JDK.



Java RunTime(JRE)

- Java Runtime Environment is provided by Java to run the applications.
- It is an implementation of JVM. It consists of run-time libraries.
- The components of JRE include - JVM(the most important component of JRE), Deployment technologies, Class Loader Subsystem, Bytecode verifier, Interpreter, Libraries, User Interface Toolkits.
- It consists of various integration libraries like- Java Database Connectivity(JDBC), JNDI(Java Naming and Directory Interface), RMI (Remote Method Invocation) etc.

Java Virtual Machine (JVM)

- Java Virtual Machine is a part of the JRE.
- JRE provides the resources and libraries to run Java applications and is responsible for converting the byte code to machine-specific code.
- As it is a part of JRE, it gets installed when JRE is installed.
- It is known as a virtual machine as it is not present physically.

Importance

- The feature of **Write Once Run Anywhere** or WORA is possible because of JVM. Java code can be developed on one platform but it can be run on different platforms also.
- JVM is actually responsible for calling the main() method of Java.
- It also provides abstraction by hiding the inner implementation from the developers utilizing libraries from JVM.

Platform Independent

Platform Definition: A platform is a combination of hardware, operating system, and software that provides an environment for running programs.
Platform Independent: Java is known as **"Platform Independent"** due to the concept of **"Write Once, Run Anywhere" (WORA).**

Because programs written in Java can be run on multiple platforms.

- Java achieves this using **JVM** and **Byte Code.**
- When a Java program is compiled, it is converted into **bytecode** instead of **native machine code.**
- Byte code is platform-independent and can be run on any processor or system.
- The JVM, specific to each platform, interprets and executes the bytecode.

Running Java program via command Line

Step 1 : Open a text editor like Notepad and write your Java code. Save the file with a **".java"** extension, ensuring that the filename matches the class name(e.g., save it as "Main.java").

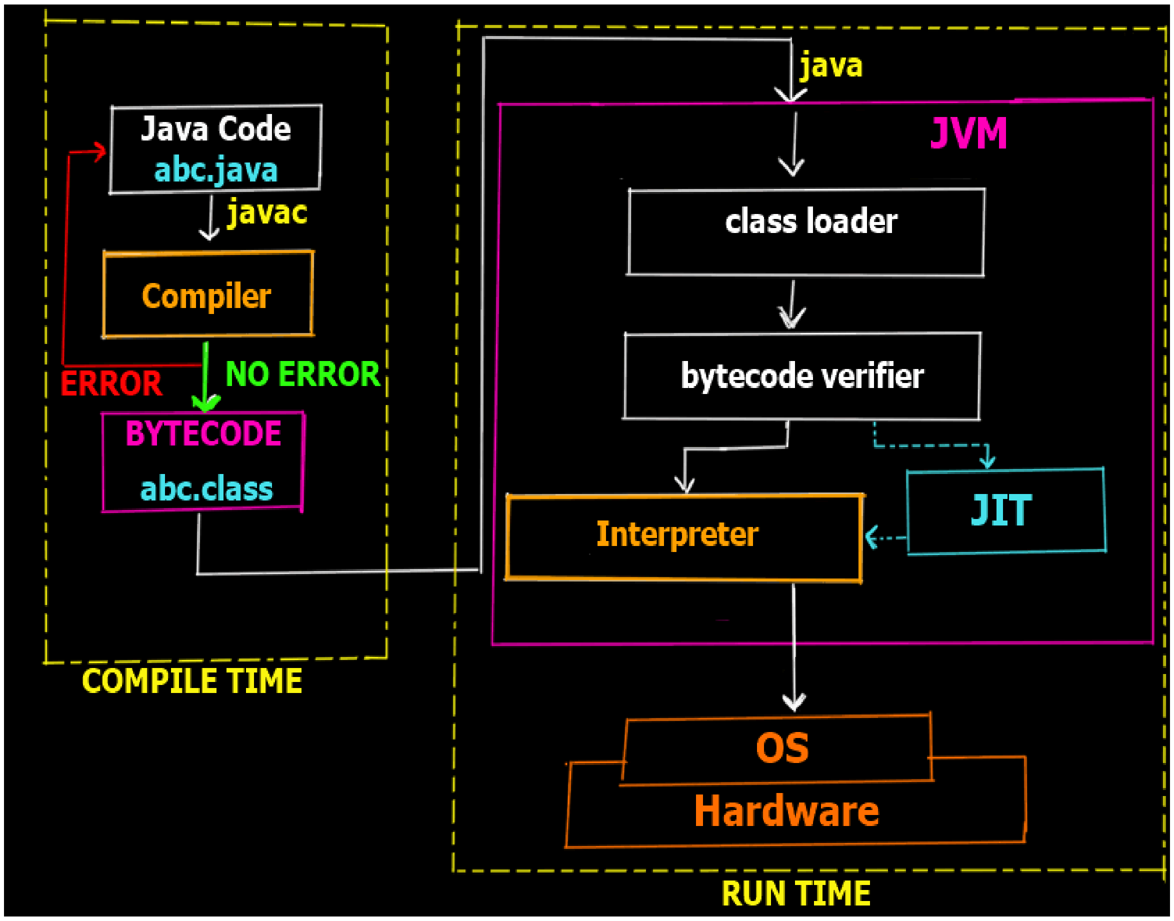
Step 2 : Open the command prompt (cmd) and navigate to the directory where your Java code file is located. Use **javac file_name.java** to compile the code.

Step 3 : After compilation, a new file with the same name as your class but with a **".class"** extension will be created in the same folder.

Step 4 : type **java file_name** execute the compiled Java program.

Step 5 : Press Enter, and you will see the **output** displayed in the command prompt window.

Execution Process of Java Program



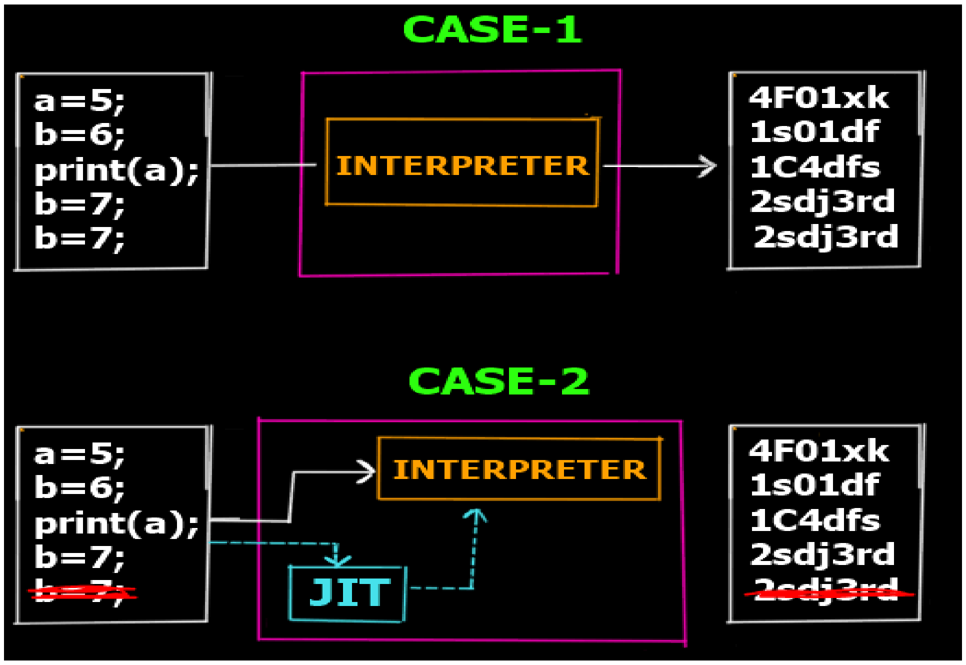
- **Step 1:** Writing the code in IDE, we have our code file as abc.java
- **Step 2:** Once you have written the code. The compiler checks the code for syntax errors and any other compile time errors and if no error is found the compiler converts the java code into an

intermediate code(abc.class file) known as **bytecode**. This intermediate code is **platform independent** (you can take this bytecode from a machine running windows and use it in any other machine running Linux or MacOS etc). Also this bytecode is an intermediate code, hence it is only understandable by the JVM and not the user or even the hardware /OS layer.

- **Step 3:** This is the start of the Run Time phase, where the bytecode is loaded into the JVM by the class loader(another inbuilt program inside the JVM).
- **Step 4:** Now the bytecode verifier(an inbuilt program inside the JVM) checks the bytecode for its integrity and if no issues are found passes it to the interpreter.
- **Step 5**: Since java is both compiled and interpreted language, now the interpreter inside the JVM converts each line of the bytecode into executable machine code and passed it to the OS/Hardware i.e. the CPU to execute.

Working of JIT (Just-In-Time) Compiler

In the above steps we didn’t mention the working of **JIT compiler**.Lets understand this by taking 2 case examples –



- **Case 1:**
    - We are at the interpretation phase (Step 5 of the overall program execution). Clearly the last 2 lines are the same or redundant and does not have any effect on the actual output but yet since the interpreter works line by line it still creates 5 lines of machine code for 5 lines of the bytecode.**[inefficient]**
- **Case 2:**
    - In case 2 we have the **JIT compiler.** Now before the bytecode is passed onto the interpreter for conversion to machine code, the **JIT compiler scans** the full code to see if it can be **optimized**. As it finds the last line is redundant it removes it from the bytecode and passes only 4 lines to the interpreter thus making it more **efficient** and **faster** as the interpreter now has 1 line less to interpret.

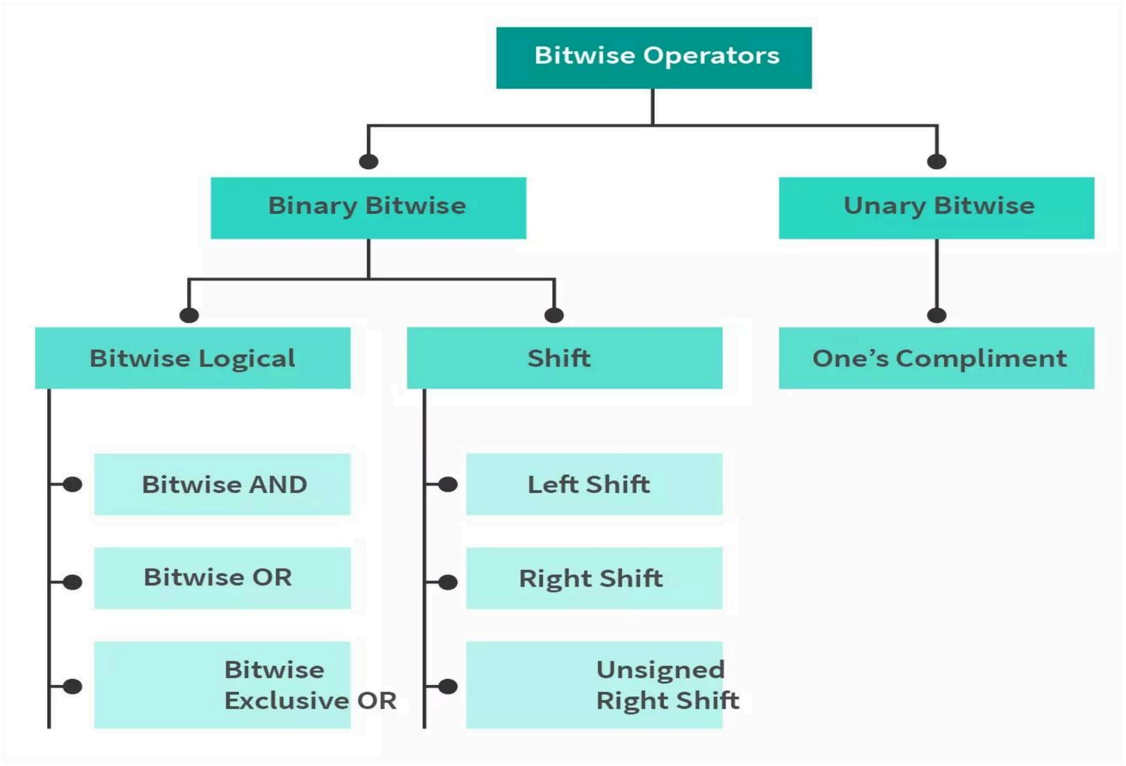
- So this is how the JIT compiler speeds up the overall execution process.
- JIT compilers are an **optional** step and not invoked every time.

|| DAY 38 ||

Bitwise Operators

Bitwise operators work on a binary equivalent of decimal numbers

- First, the operands are converted to their binary representation
- Next, the operator is applied to each binary number and the result is calculated
- Finally, the result is converted back to its decimal representation



Bitwise Logical

1. Bitwise AND(&) : If **both** the bits are **1**, the solution has **1** in that bit position else **0**.

Its Truth Table:

- 1 & 0 => gives 0
- 0 & 1 => gives 0
- 0 & 0 => gives 0
- 1 & 1 => gives 1

Example : Perform Bitwise AND Operation of **6 and 8 (6 & 8)**

6 = 0110 (In Binary), 8 = 1000 (In Binary)

0110 & 1000 -> 0000 = 0 (In decimal).

2. Bitwise OR(|): if **any bits** is **1** then it will give **1**

Its Truth Table:

- 1 | 0 => gives 1
- 0 | 1 => gives 1
- 0 | 0 => gives 0
- 1 | 1 => gives 1

Example : Perform Bitwise AND Operation of 6 and 8 (6 & 8)

6 = 0110 (In Binary), 8 = 1000 (In Binary)

0110 | 1000 -> 1110 = 14 (In decimal)

2. Bitwise XOR(^)

If the bits are **opposite**, the solution has a **1** in that bit position and if they are **matched**, a **0** is returned.

Its Truth Table:

- 1 ^ 0 => gives 1
- 0 ^ 1 => gives 1
- 0 ^ 0 => gives 0
- 1 ^ 1 => gives 0

Example : Perform Bitwise AND Operation of 6 and 8 (6 & 8)

6 = 0110 (In Binary), 8 = 1000 (In Binary)

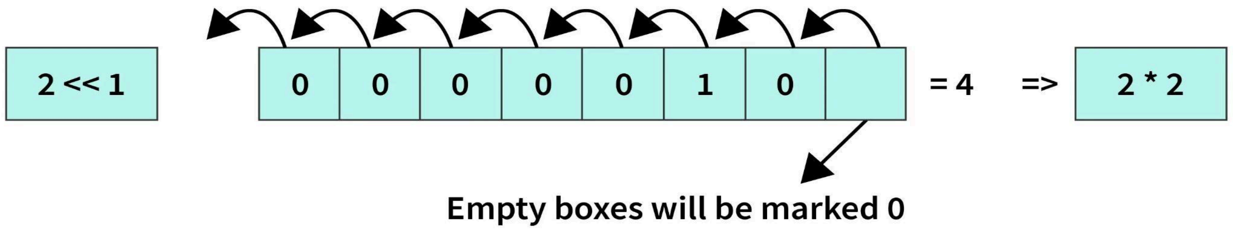
0110 ^ 1000 -> 1110 = 14 (In decimal)

Bit Shift (>>, <<,>>>)

Shifts each digit in a number's binary representation **left(<<) or right(>>)** by as many spaces as specified by the **second operand.**

There are three types of shift:

1. Left shift: << : 2 << 1



- 2 << 1 => 0010 << 1 => 0100 => 4(in decimal).
- Add as no. of 0’s on the right side of no. as no. of digit you need to shift.

2. Signed right shift: >> : 8 >> 2

- 8 >> 2 => 1000 >> 2 => 0010 => 2(in decimal).

4. Bitwise Complement (~)

- Bitwise Not or Complement operator invert each input bit.This inverted cycle is called the 1’s complement of a bit series.
- All samples of 0 become 1, and all samples of 1 become 0

Example : ~6(means 1's complement of 6)

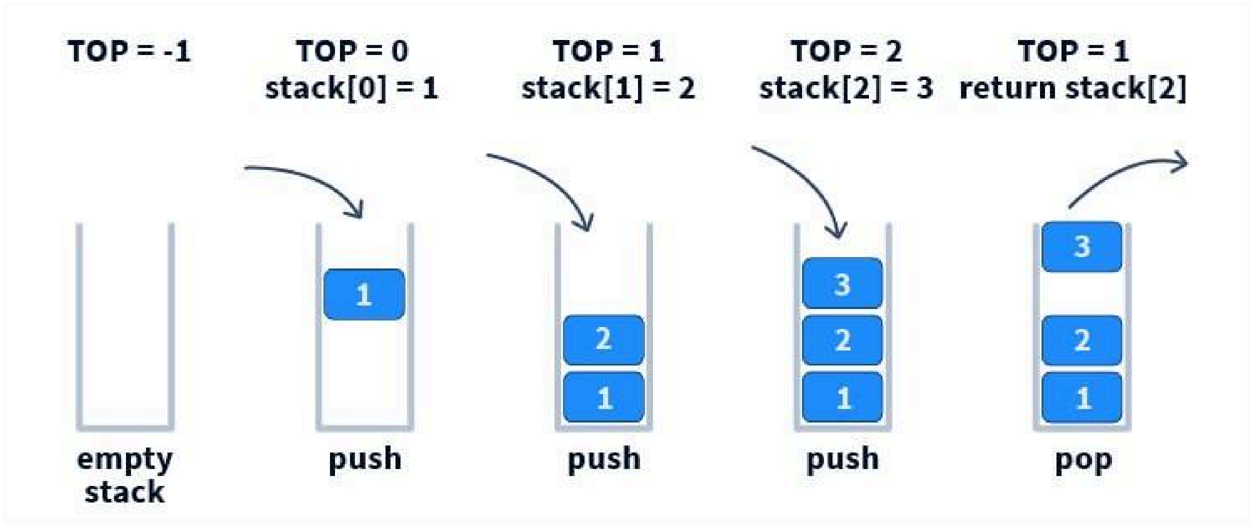
~6 => -7   **[Trick => (-n+1)]**

|| DAY 39 ||

Stack Memory

Stack Data Structure

- A stack is a **linear data structure**. It provides a container to store data items which can be accessed from one side of the container only.
- Here the element that is inserted at last is deleted first.
- This is called the Last-In-First-Out (LIFO) principle.



Stack Memory

- The Stack Memory utilizes Static Memory Allocation, storing function calls, method-specific data, and object references.
- It has a fixed size that doesn't change during execution.
- Access is in the Last-In-First-Out (LIFO) order.

Example:

Consider the following program with methods **main**, **addOne**, and **addTwo**. The stack usage is explained step by step.

```
public class Main {
   public static int addNumbers(int x, int y) {
       int result = x + y;
       return result;
   }
   public static void main(String[] args) {
       int a = 5;
       int b = 10;
       int sum = addNumbers(a, b);
       System.out.println("Sum: " + sum);
   }
}
```

The **main** method starts executing, and **local variables a and b** are stored in the stack memory. The **addNumbers** method is called, and local variables **x, y,** and **result** are stored in the stack memory for the duration of the method execution.

What exactly is the return statement ?

When a **return** statement is encountered in a method, the **control flow** immediately **exits** the method, and the program continues executing from the point where the method was **called**.

Stack OverflowError

In cases where there's insufficient space to create new objects, the system may raise a java.lang.StackOverFlowError.

Recursion

- Process by which a problem depends on solutions to smaller instances of the same problem.
- Breaks down a complex problem into simpler, similar subproblems.
- The **base case** is the simplest form of the problem that doesn't need further breakdown.
- **Base case(s)** are essential to **prevent infinite recursion.**

Uses extra space in the stack for recursive calls.

- Each recursive call adds a new frame to the call stack.
- The stack keeps track of the function calls and their local variables.

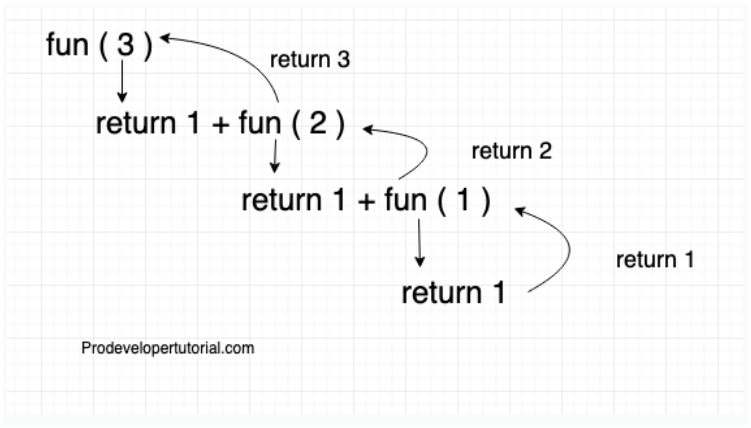
Considerations:

- Choose between recursion and iteration based on problem characteristics and simplicity of implementation.
- Recursion can lead to clearer, more concise code for certain problems, but it may come with a performance cost.
- Understanding the call stack is crucial for managing recursion efficiently.

Iterative vs Recursion

- **Iterative**:
    - Uses loops for repetitive execution.
    - Generally more memory-efficient.
- **Recursion**:
    - Uses function calls to solve problems.
    - May be more intuitive for problems with inherent recursive structure.
    - Can lead to stack overflow for deep recursion or without proper base cases.

Factorial Recursion Tree


fun (3)
return 3
return 1 + fun (2)
return 2
return 1 + fun (1)
return 1
return 1
Prodevelopertutorial.com