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.

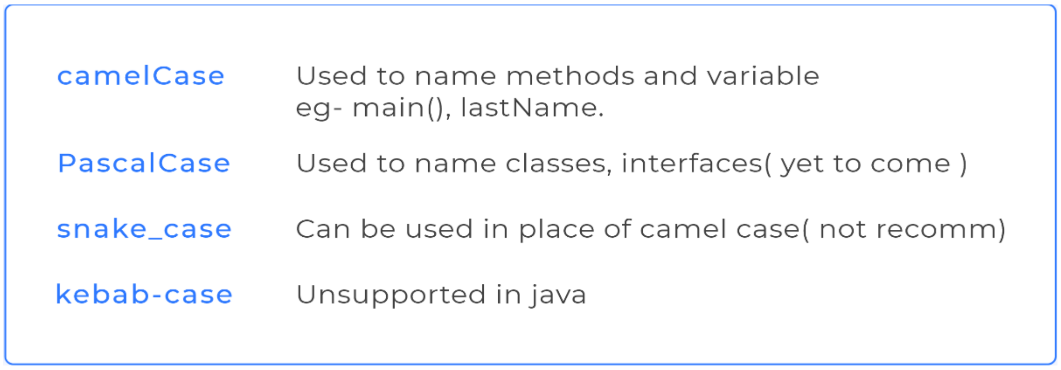
3. No Reserved Words : You cannot use Java's reserved words (also known as keywords) as identifiers.

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.

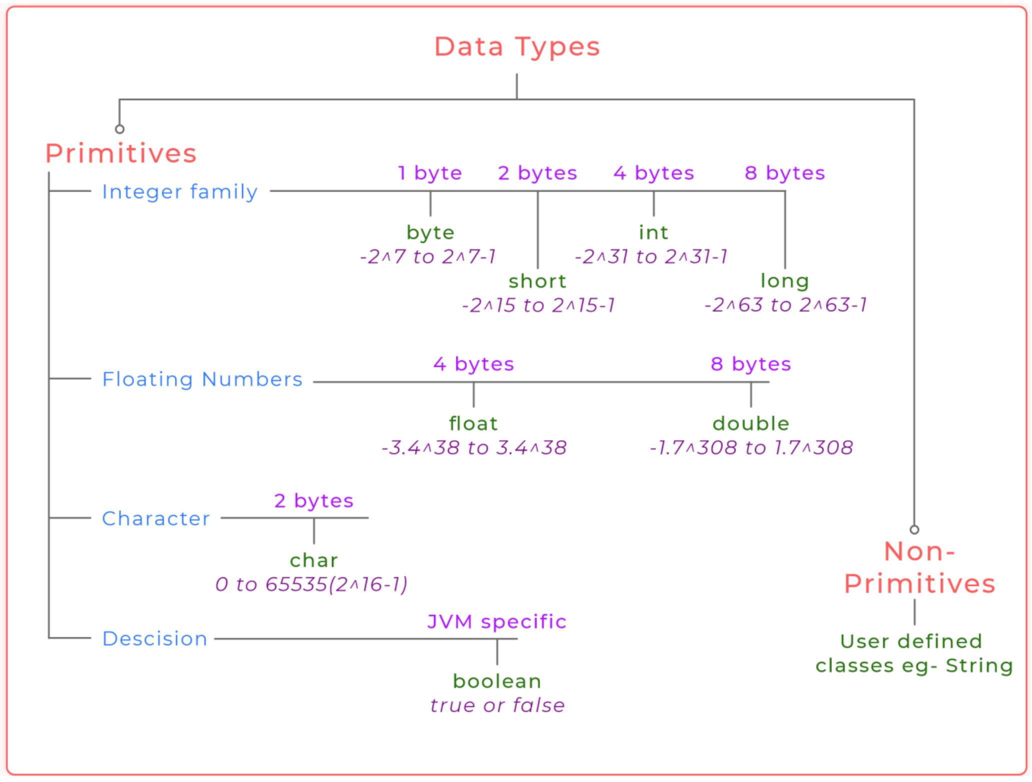
DATA TYPES

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:

If we 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

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 :

```java
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

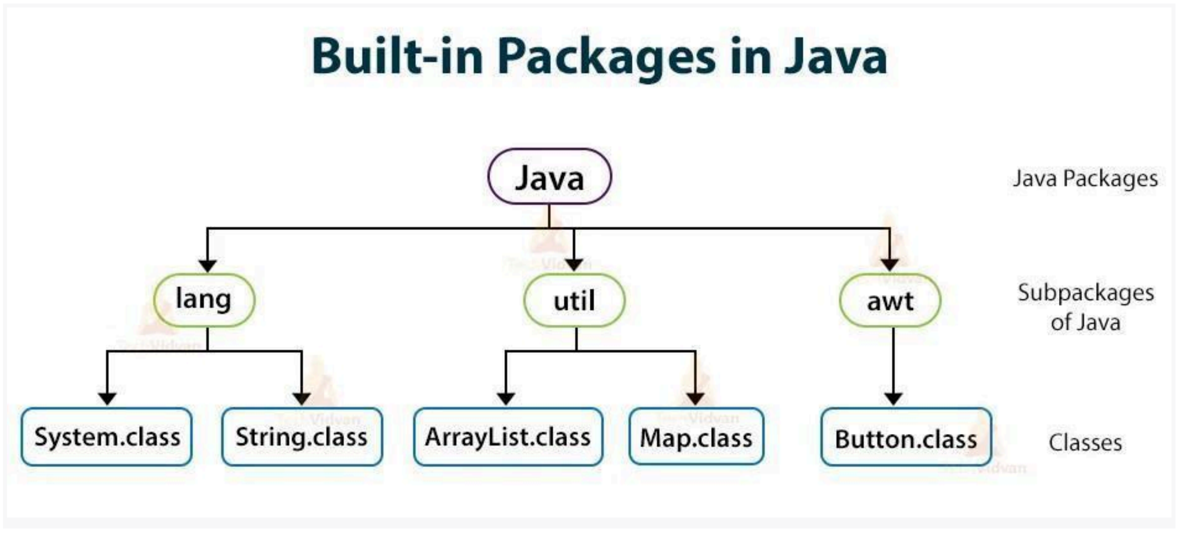
A Java package is a collection of similar types of sub-packages, interfaces, and classes. They help you manage and group related classes, interfaces, and sub-packages to avoid naming conflicts and create a more organized and maintainable codebase.

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 a 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:

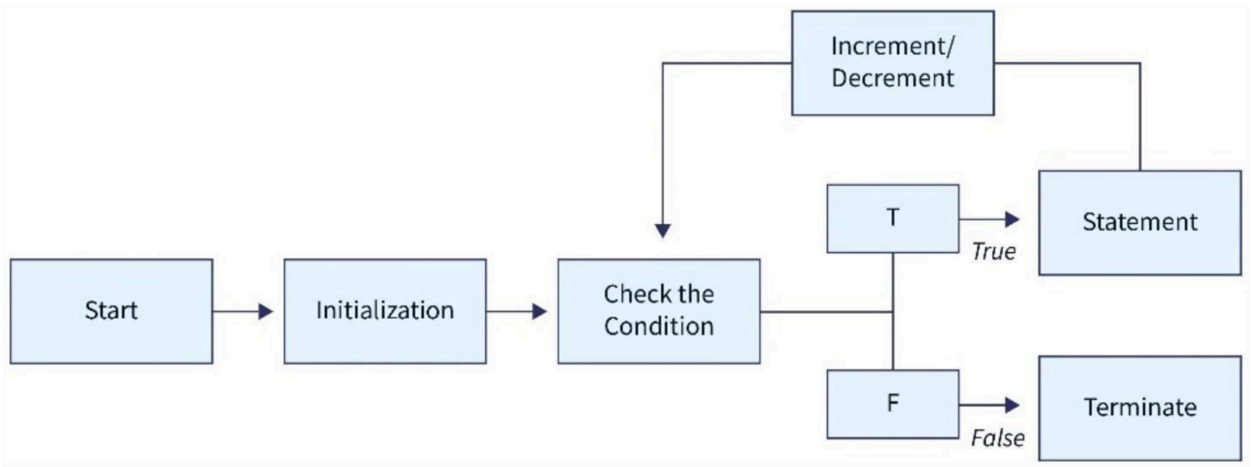
```
for(declaration,Initialization ; Condition ; Change){
   // Body of the Loop (Statement(s))
}
```

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++) {}
```

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:

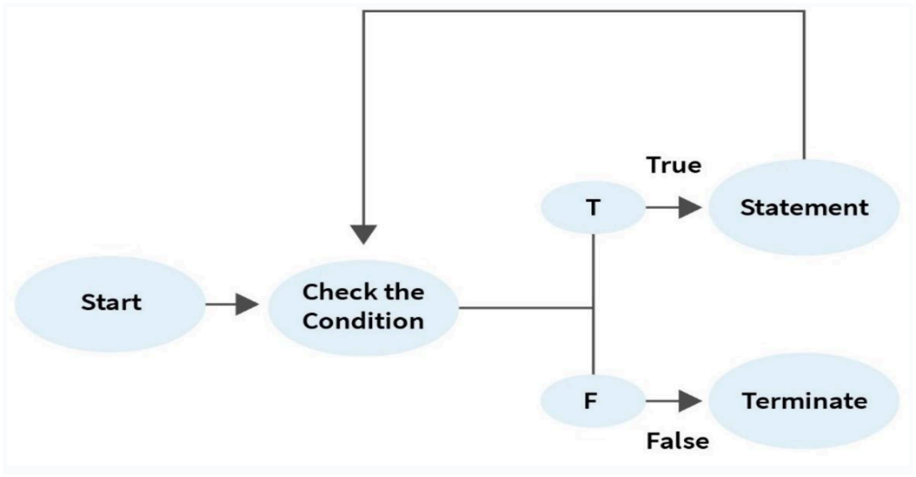
```
initialization
while (condition){
//  Body of the loop
//   Updation
}
```

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 :

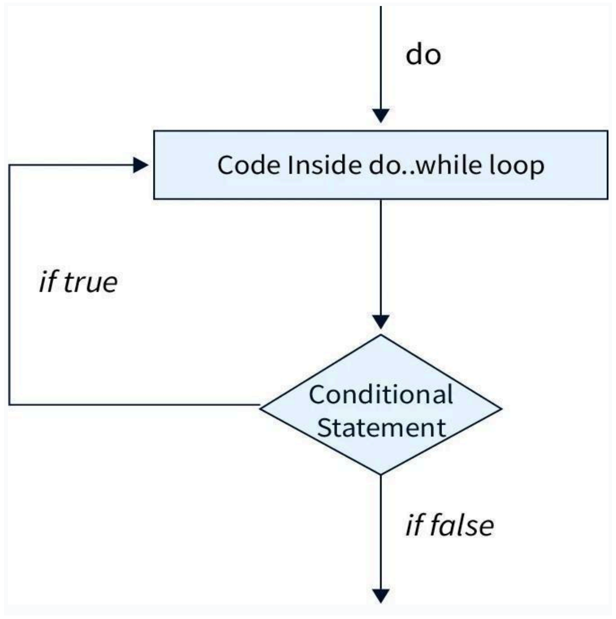
```
Initialization;
do {
// Body of the loop (Statement(s))
    // Updation;
}
while(Condition);
```

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 condition**).
- **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 :

```java
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

```java
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
```

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

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

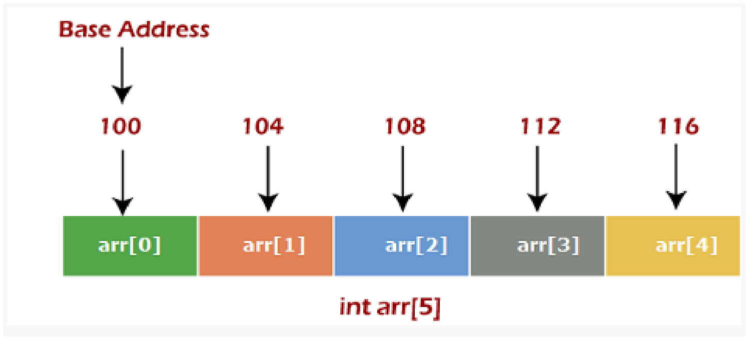
Stack: It is memory in which the size of the stack is limited and predefined during program execution. Exceeding this limit can result in a StackOverflowError(DTL).

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

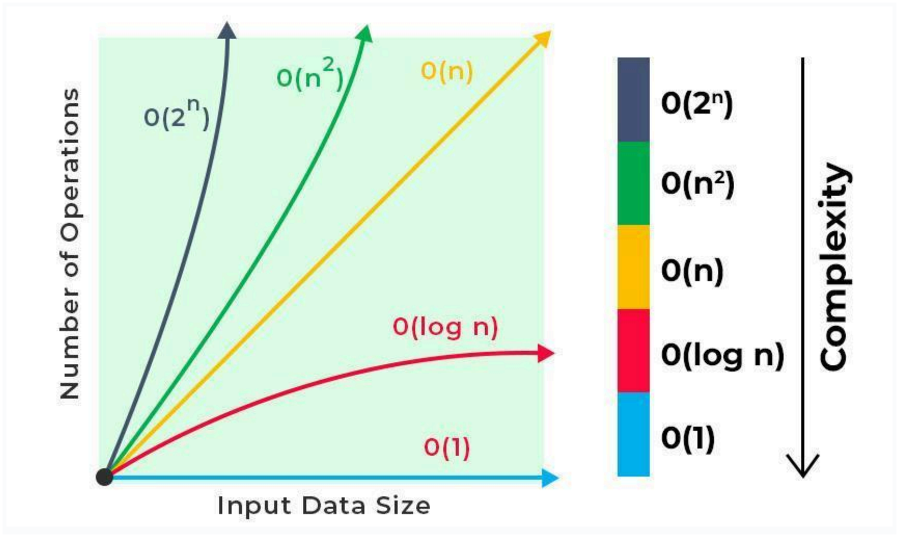
There are major 3 notations –

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;
   }
}
```