Basic Structure of a Java Program: Understanding our First Java Hello World Program

Basic Structure of a Java Program

```
package com.company; // Groups classes
public class Main{// Entrypoint into the application
        public static void main(String[]args){
                System.out.println(“Hello World”);
        }
}
```

Working of the "Hello World" program shown above :

1. package com.company :
    - Packages are used to group the related classes.
    - The "Package" keyword is used to create packages in Java.
    - Here, com.company is the name of our package.
2. public class Main :
    - In Java, every program must contain a class.
    - The filename and name of the class should be the same.
    - Here, we've created a class named "Main".
    - It is the entry point to the application.
3. public static void main(String[]args){..} :
    - This is the main() method of our Java program.
    - Every Java program must contain the main() method.
4. System.out.println("Hello World"):
    - The above code is used to display the output on the screen.
    - Anything passed inside the inverted commas is printed on the screen as plain text.

Naming Conventions

- For classes, we use Pascal Convention. The first and Subsequent characters from a word are capital letters (uppercase).
  Example: Main, MyScanner, MyEmployee, CodeWithHarry

- For functions and variables, we use camelCaseConvention. Here the first character is lowercase, and the subsequent characters are uppercase like myScanner, myMarks, CodeWithHarry

Java Tutorial: Variables and Data Types in Java Programming

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. This set of these rules is called syntax. It’s like Vocabulary and Grammar of Java.

Variables

- A variable is a container that stores a value.
- This value can be changed during the execution of the program.
- Example: int number = 8; (Here, int is a data type, the number is the variable name, and 8 is the value it contains/stores).

Rules for declaring a variable name

We can choose a name while declaring a Java variable if the following rules are followed:

- Must not begin with a digit. (E.g., 1arry is an invalid variable)
- Name is case sensitive. (Harry and harry are different)
- Should not be a keyword (like Void).
- White space is not allowed. (int Code With Harry is invalid)
- Can contain alphabets, $character, _character, and digits if the other conditions are met.

Data Types

Data types in Java fall under the following categories

1. Primitive Data Types (Intrinsic)
2. Non-Primitive Data Types (Derived)

Primitive Data Types

Java is statically typed, i.e., variables must be declared before use. Java supports 8 primitive data types:

Data Type	Size	Value Range
1. Byte	1 byte	-128 to 127
2. short	1 byte	-32,768 to 32,767
3. int	2 byte	-2,147,483,648 to 2,147,483,647
4. float	4 byte	$3.40282347 \times 10^{38}$ to $1.40239846 \times 10^{-45}$
5. long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
6. double	8 byte	$1.7976931348623157 \times 10^{308}$, $4.9406564584124654 \times 10^{-324}$
7. char	2 byte	0 to 65,535
8. boolean	Depends on JVM	True or False

Quick Quiz: Write a Java program to add three numbers,

Java Tutorial: Literals in Java

Literals

A constant value that can be assigned to the variable is called a literal.

- 101 – Integer literal
- 10.1f – float literal
- 10.1 – double literal (default type for decimals)
- 'A' – character literal
- true – Boolean literal
- "Harry" – String literal

Keywords

Words that are reserved and used by the Java compiler. They cannot be used as an Identifier. {You can visit docs.oracle.com for a comprehensive list}

Code as Described in the Video

```
package com.company;

public class CWH_04_literals {
    public static void main(String[] args) {
        byte age = 34;
        int age2 = 56;
        short age3 = 87;
        long ageDino = 5666666666666L;
        char ch = 'A';
        float f1 = 5.6f;
        double d1 = 4.66;

        boolean a = true;
        System.out.print(age);
        String str = "Harry";
        System.out.println(str);

    }
}
```

Java Tutorial: Getting User Input in Java

Reading data from the Keyboard :

Scanner class of java.util package is used to take input from the user's keyboard.The Scanner class has many methods for taking input from the user depending upon the type of input. To use any of the methods of the Scanner class, first, we need to create an object of the Scanner class as shown in the below example :

```
import java.util.Scanner;  // Importing  the Scanner class
Scanner sc = new Scanner(System.in);  //Creating an object named "sc" of the Scanner class.
```

Copy

Taking an integer input from the keyboard :

```
Scanner S = new Scanner(System.in);  //(Read from the keyboard)
int a = S.nextInt();  //(Method to read from the keyboard)
```

Code as Described in the Video

```
package com.company;
import java.util.Scanner;

public class CWH_05_TakingInpu {
    public static void main(String[] args) {
        System.out.println("Taking Input From the User");
        Scanner sc = new Scanner(System.in);
//        System.out.println("Enter number 1");
//        int a = sc.nextInt();
//        float a = sc.nextFloat();
//        System.out.println("Enter number 2");
//        int b = sc.nextInt();
//        float b = sc.nextFloat();

//        int sum = a +b;
//        float sum = a +b;
//        System.out.println("The sum of these numbers is");
//        System.out.println(sum);
//        boolean b1 = sc.hasNextInt();
//        System.out.println(b1);
```

```
//        String str = sc.next();
        String str = sc.nextLine();
        System.out.println(str);


    }
}
```

Java Tutorial: Operators, Types of Operators & Expressions in Java

- An operator is a symbol that the compiler to perform a specific operation on operands.
- Example : a + b = c
- In the above example, 'a' and 'b' are operands on which the '+' operator is applied.

Types of operators :

1. Arithmetic Operators :
    - Arithmetic operators are used to perform mathematical operations such as addition, division, etc on expressions.
    - Arithmetic operators cannot work with Booleans.
    - % operator can work on floats and doubles.
    - Let x=7 and y=2

Operator	Description	Example
+ (Addition)	Used to add two numbers	x + y = 9
- (Subtraction)	Used to subtract the right-hand side value from the left-hand side value	x - y = 5
* (Multiplication)	Used to multiply two values.	x * y = 14
/ (Division)	Used to divide left-hand Value by right-hand value.	x / y = 3
% (Modulus)	Used to print the remainder after dividing the left-hand side value from the right-hand side value.	x % y = 1
++ (Increment)	Increases the value of operand by 1.	x++ = 8
-- (Decrement)	Decreases the value of operand by 1.	y-- = 1

2. Comparison Operators :
    - As the name suggests, these operators are used to compare two operands.
    - Let x=7 and y=2

Operator	Description	Example
== (Equal to)	Checks if two operands are equal. Returns a boolean value.	x == y --> False
!= (Not equal	Checks if two operands are not equal. Returns a boolean value.	x != y --> True
> (Greater than)	Checks if the left-hand side value is greater than the right-hand side value. Returns a boolean value.	x > y --> True
< (Less than)	Checks if the left-hand side value is smaller than the right-hand side value. Returns a boolean value.	x < y --> False
>=(Greater than or equal to)	Checks if the left-hand side value is greater than or equal to the right-hand side value. Returns a boolean value.	x >= y --> True
<= (Less than or equal to)	Checks if the left-hand side value is less than or equal to the right-hand side value. Returns a boolean value.	x <= y -->False

3. Logical Operators :

- These operators determine the logic in an expression containing two or more values or variables.
- Let x = 8 and y =2

&& (logical and)	Returns true if both operands are true.	x<y && x!=y --> True
|| (logical or)	Returns true if any of the operand is true.	x<y && x==y --> True
! (logical not)	Returns true if the result of the expression is false and vice-versa	!(x<y && x==y) --> False

4. Bitwise Operators :

- These operators perform the operations on every bit of a number.
- Let x =2 and y=3. So 2 in binary is 100, and 3 is 011.

Operator	Description	Example
& (bitwise and)	1&1 =1, 0&1=0,1&0=0,1&1=1, 0&0 =0	(A & B) = (100 & 011) = 000
| (bitwise or)	1&0 =1, 0&1=1,1&1=1, 0&0=0	(A | B) = (100 | 011) = 111
^ (bitwise XOR)	1&0 =1, 0&1=1,1&1=0, 0&0=0	(A ^ B) = (100 ^ 011) = 111
<< (left shift)	This operator moves the value left by the number of bits specified.	13<<2 = 52(decimal)
>> (right shift)	This operator moves the value left by the number of bits specified.	13>>2 = 3(decimal)

Precedence of operators

The operators are applied and evaluated based on precedence. For example, (+, -) has less precedence compared to (*, /). Hence * and / are evaluated first.
In case we like to change this order, we use parenthesis ().

Code as Described in the Video

```
package com.company;

public class CWH_Ch2_Operators {
    public static void main(String[] args) {
        // 1. Arithmetic Operators
        int a = 4;
        // int b = 6 % a; // Modulo Operator
        // 4.8%1.1 --> Returns Decimal Remainder

        // 2. Assignment Operators
        int b = 9;
        b *= 3;
        System.out.println(b);

        // 3. Comparison Operators
        // System.out.println(64<6);

        // 4. Logical Operators
```

```
        // System.out.println(64>5 && 64>98);
        System.out.println(64>5 || 64>98);

        // 5. Bitwise Operators
        System.out.println(2&3);
        //        10
        //        11
        //        ----
        //        10
    }
}
```

Java Tutorial: Associativity of Operators in Java

Associativity

Associativity tells the direction of the execution of operators. It can either be left to right or vice versa.
/ * -> L to R
+ - -> L to R
++, = -> R to L
Here is the precedence and associativity table which makes it easy for you to understand these topics better:

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
->	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
-	Unary minus (Negation)		
++	Increment		
--	Decrement		
&	Address-of		
*	Indirection		
(type)	Cast		
sizeof	Sizeof		
*	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
-	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
!=	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, += *=, etc.	Assignment operators	Right-to-Left	1
,	Comma	Left-to-Right	Lowest 0

Code as Described in the Video

```
package com.company;

public class cwh_09_ch2_op_pre {
    public static void main(String[] args) {
        // Precedence & Associativity
```

```
        //int a = 6*5-34/2;
        /*
        Highest precedence goes to * and /. They are then evaluated on the basis
        of left to right associativity
            =30-34/2
            =30-17
            =13
         */
        //int b = 60/5-34*2;
        /*
            = 12-34*2
            =12-68
            =-56
         */

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

        // Quick Quiz
        int x =6;
        int y = 1;
        //  int k = x * y/2;

        int b = 0;
        int c = 0;
        int a = 10;
        int k = b*b - (4*a*c)/(2*a);
        System.out.println(k);

    }
}
```

Java Tutorial: Data Type of Expressions & Increment/Decrement Operators

Resulting data type after arithmetic operation

- Result = byte + short -> integer
- Result = short + integer -> integer
- Result = long + float -> float
- Result = integer + float -> float
- Result = character + integer -> integer
- Result = character + short -> integer
- Result = long + double -> double
- Result = float + double -> double

Increment and Decrement operators

- a++, ++a (Increment Operators)
- a--, --a (Decrement Operators)

These will operate on all data types except Booleans.
Quick Quiz: Try increment and decrement operators on a Java variable

- a++ -> first use the value and then increment
- ++a -> first increment the value then use it

Quick Quiz: What will be the value of the following expression(x).

1. int y=7;
2. int x = ++y*8;
3. value of x?
4. char a = 'B';
5. a++; (a is not 'C')

Code as Described in the Video

```
package com.company;

public class cwh_10_resulting_data_type {
    public static void main(String[] args) {
        /* byte x = 5;
        int y = 6;
        short z = 8;
        int a = y + z;
        float b = 6.54f + x;
        System.out.println(b); */

        // Increment and Decrement Operators
```

```
        int i = 56;
        // int b = i++; // first b is assigned i (56) then i is incremented
        int j = 67;
        int c = ++j; // first j is incremented then c is assigned j (68)
        System.out.println(i++);
        System.out.println(i);
        System.out.println(++i);
        System.out.println(i);
        int y = 7;
        System.out.println( ++y *8);
        char ch = 'a';
        System.out.println(++ch);
    }
}
```

Java Tutorial: Introduction to Strings

- A string is a sequence of characters.
- Strings are objects that represent a char array. For example :

```
char[] str = {'H','A','R','R','Y'};
```

```
String s = new String(str);
```

Copy

is same as :

```
String s = "Harry";
```

Copy

- Strings are immutable and cannot be changed.
- java.lang.String class is used to create a String object.
- The string is a class but can be used as a data type.

Syntax of strings in Java :

```
String <String_name> = "<sequence_of_string>";
```

Copy

Example :

```
String str = "CodeWithHarry";
```

Copy

In the above example, str is a reference, and “CodeWithHarrt” is an object.

Different ways to create a string in Java :

In Java, strings can be created in two ways :

1. By using string literal
2. By using the new

Creating String using String literal :

```
String s1= "String literal"
```

Copy

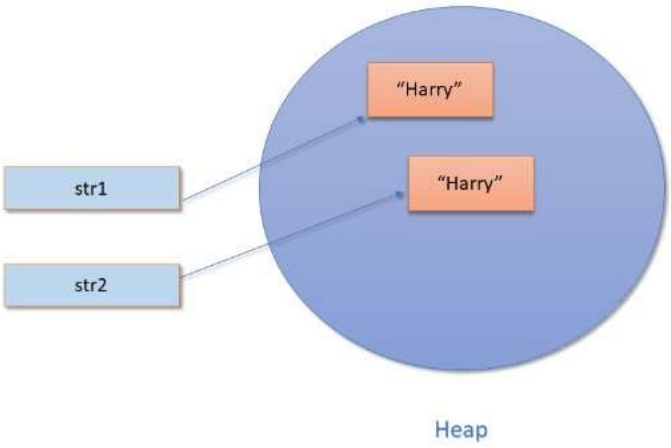
We use double quotes("") to create string using string literal. Before creating a new string instance, JVM verifies if the same string is already present in the string pool or not. If it is already present, then JVM returns a reference to the pooled instance otherwise, a new string instance is created.
In the above diagram, notice that string "Harry" is already present in the string pool, which is pointed by the str1. When we try to create the same string object using str2, JVM finds that string object with the value "Harry" is already present in the string pool; therefore, instead of creating a new object, a reference to the same object is returned.

Creating String using new :

```
String s=new String("Harry");
```

Copy

When we create a string using "new", a new object is always created in the heap memory.



In the above diagram, you can see that although the value of both string objects is the same, i.e., "Harry" still two different objects are created, and they are referred by two different reference variables, i.e., str1 and str2.

See the examples given below to get a better understanding of String literal and String object :

```
String str1 = "CodeWithHarry";
String str2 = "CodeWithHarry"
System.out.println(str1 == str2);
```

Copy

Output :

```
True
```

Copy

Returns true because str1 and str2 are referencing the same object present in the string constant pool. Now, let's see the case of the String object :

```
String str1 = new String("Keep coding");
String str2 = new String("Keep coding"");
System.out.println(str1 == str2);
```

Copy

Output :

```
False
```

Copy

Although the value of both the string object is the same, still false is displayed as output because str1 and str2 are two different string objects created in the heap. That's why it is not considered a good practice two compare two strings using the == operator. Always use the equals() method to compare two strings in Java.

Different ways to print in Java :

We can use the following ways to print in Java:

- System.out.print() // No newline at the end
- System.out.println() // Prints a new line at the end
- System.out.printf()
- System.out.format()

```
System.out.printf("%c",ch)
```

Copy

- %d for int
- %f for float
- %c for char
- %s for string

Code as written in the video

```
package com.company;
import java.util.Scanner;

public class cwh_13_strings {
    public static void main(String[] args) {
        // String name = new String("Harry");
        // String name = "Harry";
        // System.out.print("The name is: ");
        // System.out.print(name);
        int a = 6;
        float b = 5.6454f;
        System.out.printf("The value of a is %d and value of b is %8.2f", a, b);
```

```
        //System.out.format("The value of a is %d and value of b is %f", a, b);
        Scanner sc = new Scanner(System.in);
//        String st = sc.next();
//        String st = sc.nextLine();
//        System.out.println(st);


    }
}
```

Java Tutorial: String Methods in Java

String Methods operate on Java Strings. They can be used to find the length of the string, convert to lowercase, etc.
Some of the commonly used String methods are:

```
String name = "Harry";
```

Copy

(Indexes of the above string are as follows: 0-H, 1-a, 2-r, 3-r, 4-y)

Method	Description
1. length()	Returns the length of String name. (5 in this case)
2. toLowerCase()	Converts all the characters of the string to the lower case letters.
3. toUpperCase()	Converts all the characters of the string to the upper case letters.
4. trim()	Returns a new String after removing all the leading and trailing spaces from the original string.
5. substring(int start)	Returns a substring from start to the end. Substring(3) returns "ry". [Note that indexing starts from 0]
6. substring(int start, int end)	Returns a substring from the start index to the end index. The start index is included, and the end is excluded.
7. replace('r', 'p')	Returns a new string after replacing r with p. Happy is returned in this case. (This method takes char as argument)
8. startsWith("Ha")	Returns true if the name starts with the string "Ha". (True in this case)
9. endsWith("ry")	Returns true if the name ends with the string "ry". (True in this case)
10. charAt(2)	Returns the character at a given index position. (r in this case)

11. indexOf(“s”)	Returns the index of the first occurrence of the specified character in the given string.
12. lastIndexOf(“r”)	Returns the last index of the specified character from the given string. (3 in this case)
13. equals(“Harry”)	Returns true if the given string is equal to “Harry” false otherwise [Case sensitive]
14.equalsIgnoreCase(“harry”)	Returns true if two strings are equal, ignoring the case of characters.

Escape Sequence Characters :

- The sequence of characters after backslash ‘\’ = Escape Sequence Characters
- Escape Sequence Characters consist of more than one character but represent one character when used within the strings.
- Examples: \n (newline), \t (tab), \’ (single quote), \\ (backslash), etc.

Code as described in the video

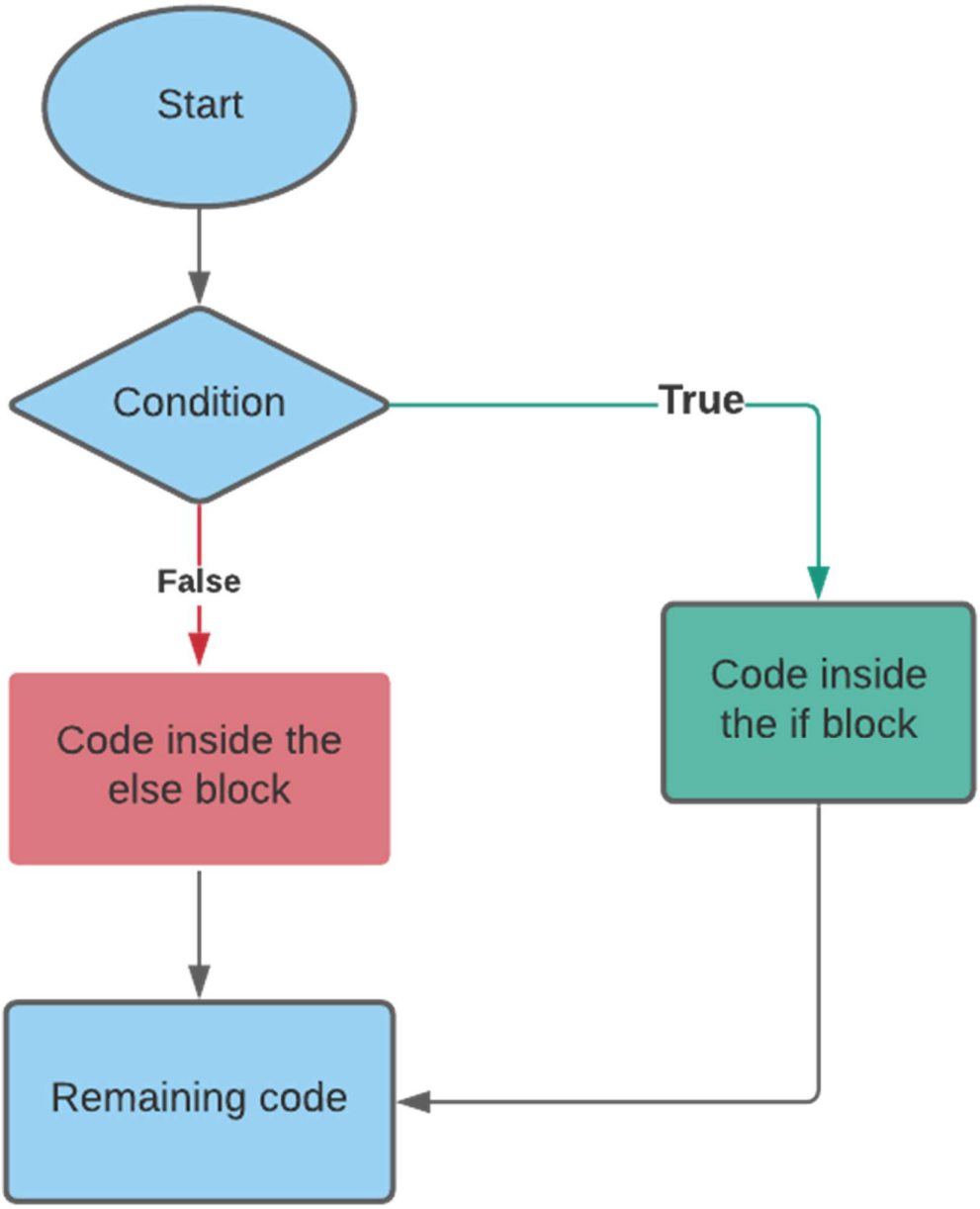
```
package com.company;

public class cwh_14_string_methods {
    public static void main(String[] args) {
        String name = "Harry";
        // System.out.println(name);
        int value = name.length();
        //System.out.println(value);

        //String lstring = name.toLowerCase();
        //System.out.println(lstring);

        //String ustring = name.toUpperCase();
        //System.out.println(ustring);

        //String nonTrimmedString = "     Harry     ";
        //System.out.println(nonTrimmedString);

        //String trimmedString = nonTrimmedString.trim();
        //System.out.println(trimmedString);
```

```
        //System.out.println(name.substring(1));
        //System.out.println(name.substring(1,5));

        //System.out.println(name.replace('r', 'p'));
        //System.out.println(name.replace("r", "ier"));

        //System.out.println(name.startsWith("Har"));
        //System.out.println(name.endsWith("dd"));

        //System.out.println(name.charAt(4));

        //String modifiedName = "Harryrryrry";
        //System.out.println(modifiedName.indexOf("rry"));
        //System.out.println(modifiedName.indexOf("rry", 4));
        //System.out.println(modifiedName.lastIndexOf("rry", 7));

        //System.out.println(name.equals("Harry"));
        System.out.println(name.equalsIgnoreCase("HarRY"));

        System.out.println("I am escape sequence\tdouble quote");




    }
}
```

Java Conditionals: If-else Statement in Java

Sometimes we want to drink coffee when we feel sleepy. Sometimes, we order junk food if it is our friend's birthday. You might want to buy an umbrella if it's raining. All these decisions depend on a certain condition being met. Similar to real life, we can execute some instructions only when a condition is met in programming also. If-else block is used to check conditions and execute a particular section of code for a specific condition.

Flow control of if-else in Java :



Decision-making instructions in Java

- If-Else Statement
- Switch Statement

If-Else Statement

Syntax of If-else statement in Java :

```
/* if (condition-to-be-checked) {
        statements-if-condition-true;
}
else {
        statements-if-condition-false;
} */
```

Copy

Example:

```
int a = 29;
if (a>18) {
        System.out.println(“You can drive”);
}
else{
    System.out.println(“You are underage!");
}
```

Copy

Output:

```
You can drive
```

Copy

If-else ladder :

- Instead of using multiple if statements, we can also use else if along with if thus forming an if-else-if-else ladder.
- Using such kind of logic reduces indents.
- Last else is executed only if all the conditions fail.

```
/* if (condition1) {

        //Statements;

else if {
```

```
        // Statements;

}

else {

        //Statements

} */
```

Java Tutorial: Relational and Logical Operators in Java

Relational Operators in Java :

Relational operators are used to evaluate conditions (true or false) inside the if statements. Some examples of relational operators are:

- == (equals)
- >= (greater than or equals to)
- > (greater than)
- < (less than)
- <= (less than or equals to)
- != (not equals)

Note: '=' is used for an assignment whereas '==' is used for equality check. The condition can be either true or false.

Logical Operators :

- Logical operators are used to provide logic to our Java programs.
- There are three types of logical operators in Java :
- && - AND
- || - OR
- ! – NOT

AND Operator :

Evaluates to true if both the conditions are true.

- Y && Y = Y
- Y && N = N
- N && Y = N
- N && N = N

Convention: # Y – True and N - False

OR Operator :

Evaluates to true when at least one of the conditions is true.

- Y || Y = Y
- Y || N = Y
- N || Y = Y
- N || N = N

Convention: # Y – True and N - False

NOT Operator :

Negates the given logic (true becomes false and vice-versa)

- !Y = N
- !N = Y

Code as Described in the Video

```
package com.company;


public class cwh_17_logical {
    public static void main(String[] args) {
        System.out.println("For Logical AND...");
        boolean a = true;
        boolean b = false;
//        if (a && b){
//            System.out.println("Y");
//        }
//        else{
//            System.out.println("N");
//        }


        System.out.println("For Logical OR...");


//        if (a || b){
//            System.out.println("Y");
//        }
//        else{
//            System.out.println("N");
//        }
```

```
        System.out.println("For Logical NOT");
        System.out.print("Not(a) is ");
        System.out.println(!a);
        System.out.print("Not(b) is ");
        System.out.println(!b);
    }
}
```

Java Tutorial: Switch Case Statements in Java

Switch Case-Control Instruction

- Switch-Case is used when we have to make a choice between the number of alternatives for a given variable.
- Var can be an integer, character, or string in Java.
- Every switch case must contain a default case. The default case is executed when all the other cases are false.
- Never forget to include the break statement after every switch case otherwise the switch case will not terminate.

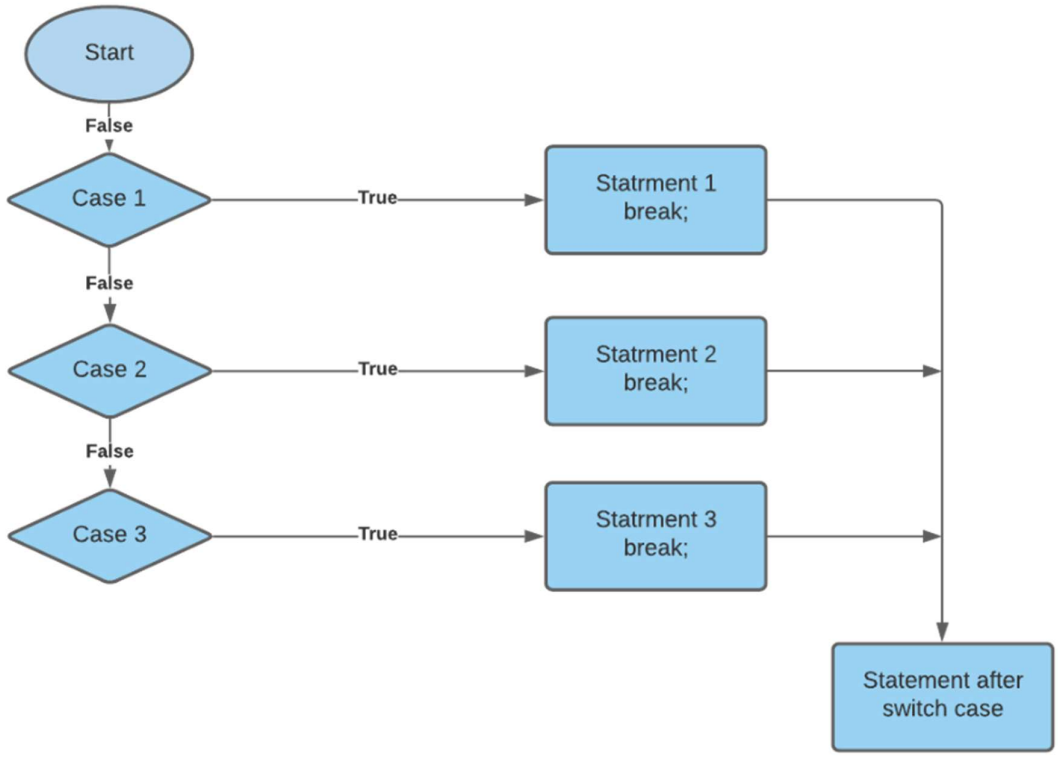
Syntax :

```
Switch(var) {
  Case C1:
            //Code;
            break;
  Case C2:
            //Code;
            break;
  Case C3:
            //Code
            break;
  default:
            //Code
```

Copy

- A switch can occur within another but in practice, this is rarely done.

Flow control of switch case in Java :



Code as Described in the Video

```
package com.company;
import java.util.Scanner;

public class cwh_18_elseif {
    public static void main(String[] args) {
        String var = "Saurabh";


        switch (var) {
            case "Shubham" -> {
                System.out.println("You are going to become an Adult!");
                System.out.println("You are going to become an Adult!");
```

```
            System.out.println("You are going to become an Adult!");
        }
        case "Saurabh" -> System.out.println("You are going to join a Job!");
        case "Vishaka" -> System.out.println("You are going to get retired!");
        default -> System.out.println("Enjoy Your life!");
    }
    System.out.println("Thanks for using my Java Code!");



    /*
    int age;
    System.out.println("Enter Your Age");
    Scanner sc = new Scanner(System.in);
    age = sc.nextInt();
    if (age>56){
        System.out.println("You are experienced!");
    }
    else if(age>46){
        System.out.println("You are semi-experienced!");
    }
    else if(age>36){
        System.out.println("You are semi-semi-experienced!");
    }
    else{
        System.out.println("You are not experienced");
    }
    if(age>2){
        System.out.println("You are not a baby!");
    }
    */


  }
}
```

Java Tutorial: While Loops in Java

- In programming languages, loops are used to execute a particular statement/set of instructions again and again.
- The execution of the loop starts when some conditions become true.
- For example, print 1 to 1000, print multiplication table of 7, etc.
- Loops make it easy for us to tell the computer that a given set of instructions need to be executed repeatedly.

Types of Loops :

Primarily, there are three types of loops in Java:

1. While loop
2. do-while loop
3. for loop

Let's look into these, one by one.

While loops :

- The while loop in Java is used when we need to execute a block of code again and again based on a given boolean condition.
- Use a while loop if the exact number of iterations is not known.
- If the condition never becomes false, the while loop keeps getting executed. Such a loop is known as an infinite loop.

```
/*
while (Boolean condition)

{

        // Statements    -> This keeps executing as long as the condition is true.

}
*/
```

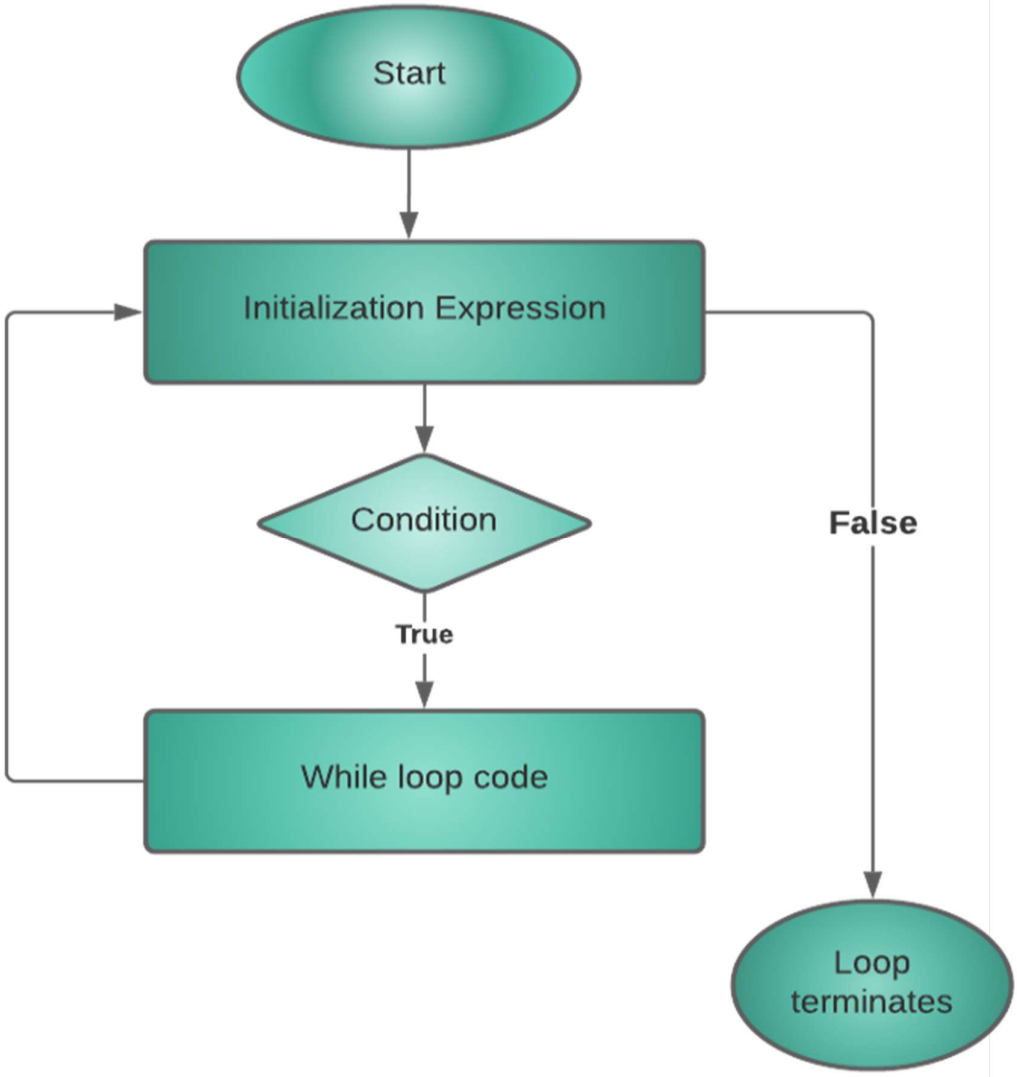
- Copy
- Example :

```
int i=10;
while(i>0){
System.out.println(i);
i--;
```

```
}
```

Copy

Flow control of while loop :



Quick Quiz: Write a program to print natural numbers from 100 to 200.

Code as described in the video :

```
package com.company;

public class cwh_21_ch5_loops {
    public static void main(String[] args) {
        System.out.println(1);
```

```
        System.out.println(2);
        System.out.println(3);

        System.out.println("Using Loops:");
        int i = 100;
        while(i<=200){
            System.out.println(i);
            i++;
        }
        System.out.println("Finish Running While Loop!");

//      while(true){
//          System.out.println("I am an infinite while loop!");
//      }
    }
}
```

Java Tutorial: The do-while loop in Java

Do-while loop:

- Do- while loop is similar to a while loop except for the fact that it is guaranteed to execute at least once.
- Use a do-while loop when the exact number of iterations is unknown, but you need to execute a code block at least once.
- After executing a part of a program for once, the rest of the code gets executed on the basis of a given boolean condition.

Syntax :

```
/* do {

        //code

} while (condition);          //Note this semicolon */
```

Copy

Example :

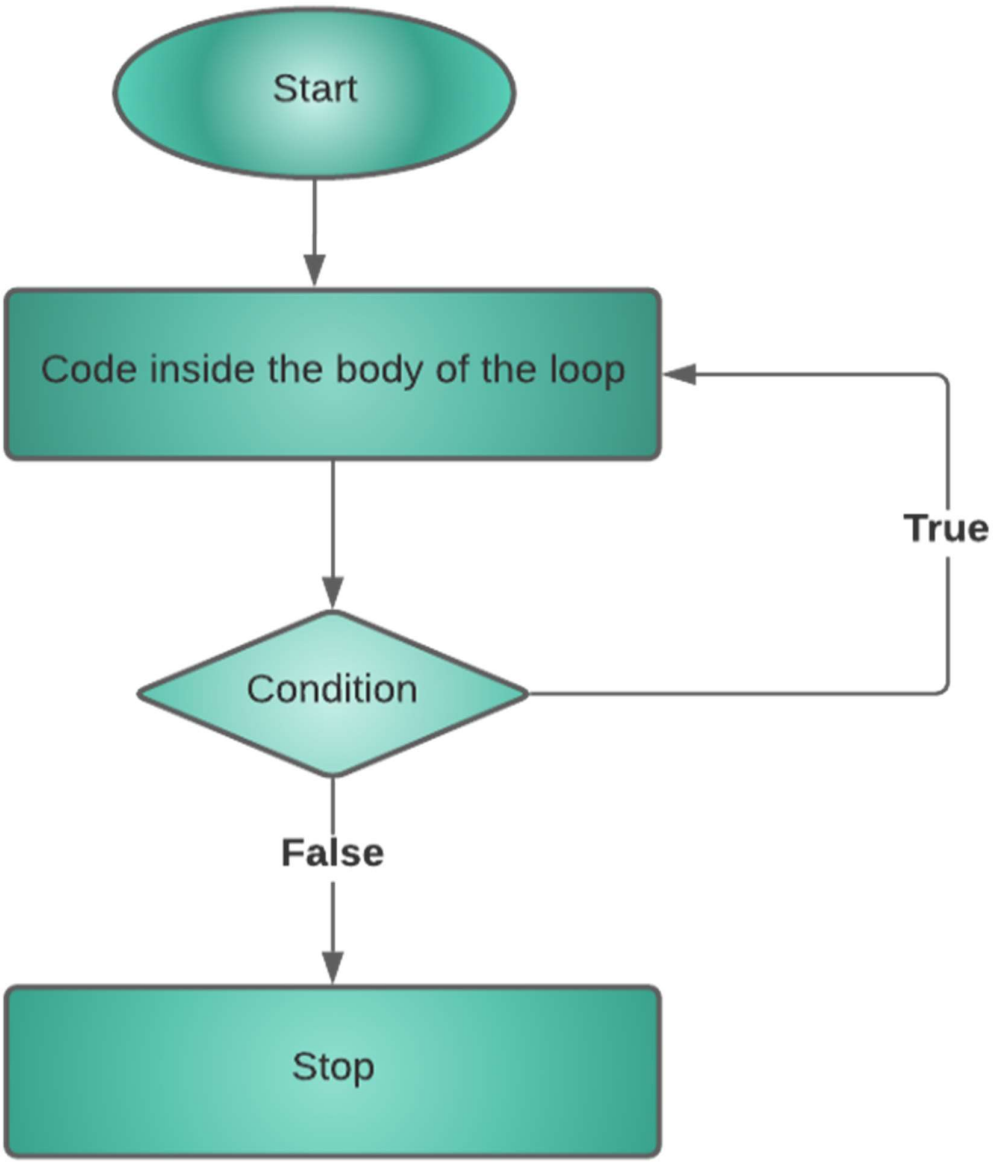
```
int i=1;
do{
System.out.println(i);
i++;
}while(i<=10);
```

Copy

Difference Between while loop and do-while loop :

- while – checks the condition & executes the code.
- do-while – executes the code at least once and then checks the condition. Because of this reason, the code in the do-while loop executes at least once, even if the condition fails.

Flow control of do-while loop :



Quick Quiz: Write a program to print first n natural numbers using a do-while loop.

Code as described in the video :

```
package com.company;

public class cwh_22_ch4_do_while {
    public static void main(String[] args) {
//        int a = 0;
```

```
//        while(a<5){
//            System.out.println(a);
//            a++;
//        }
        int b = 10;
        do {
            System.out.println(b);
            b++;
        }while(b<5);

        int c = 1;
        do{
            System.out.println(c);
            c++;
        }while(c<=45);

    }
}
```

Java Tutorial: The for Loop in Java

For loop:

- For loop in java is used to iterate a block of code multiple times.
- Use for loop only when the exact number of iterations needed is already known to you.

Syntax :

```
/* for (initialize; check_bool_expression; update){

            //code;

} */
```

Copy

- Initializer: Initializes the value of a variable. This part is executed only once.

- check_bool_expression: The code inside the for loop is executed only when this condition returns true.
- update: Updates the value of the initial variable.

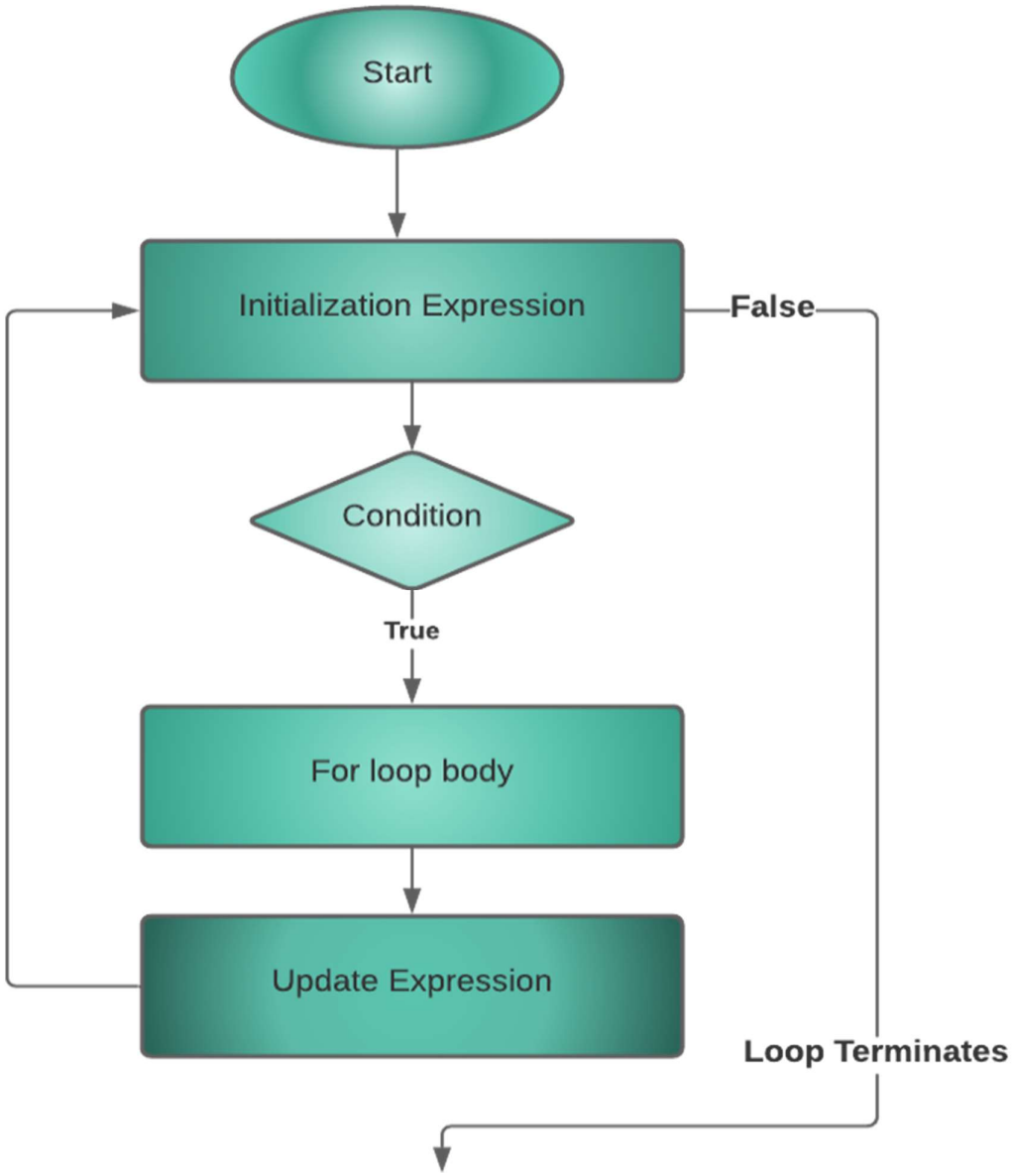
Example :

```
for (i=7; i!=0; i--){

        System.out.println(i);

}
```

Copy

The above for loop initializes the value of i=7 and keeps printing as well as decrementing the value of i till i do not get equals to 0.

Flow control of for loop :



Quick Quiz 1: Write a program to print first n odd numbers using a for loop.
Quick Quiz 2: Write a program to print first n natural numbers in reverse order.

Code as described in the video :

```
package com.company;

public class cwh_23_for_loop {
    public static void main(String[] args) {
//        for (int i=1; i<=10; i++){
```

```
//            System.out.println(i);
//        }
        // 2i = Even Numbers = 0, 2, 4, 6, 8
        // 2i+1 = Odd Numbers = 1, 3, 5, 7, 9
        //int n = 3;
        //for (int i =0; i<n; i++){
        //    System.out.println(2*i+1);
        //}

        for(int i=5; i!=0; i--){
            System.out.println(i);
        }
    }
}
```

Java Tutorial: break and continue in Java

Break statement :

- The break statement is used to exit the loop irrespective of whether the condition is true or false.
- Whenever a 'break' is encountered inside the loop, the control is sent outside the loop.

Syntax :

```
break;
```

Copy

Example to demonstrate the use of break inside a for loop :

```
public class CWH_break {
public static void main(String[] args) {
   //using for loop
   for(int i=10;i>0;i--){
       if(i==7){
           break;   //break the loop
       }
```

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

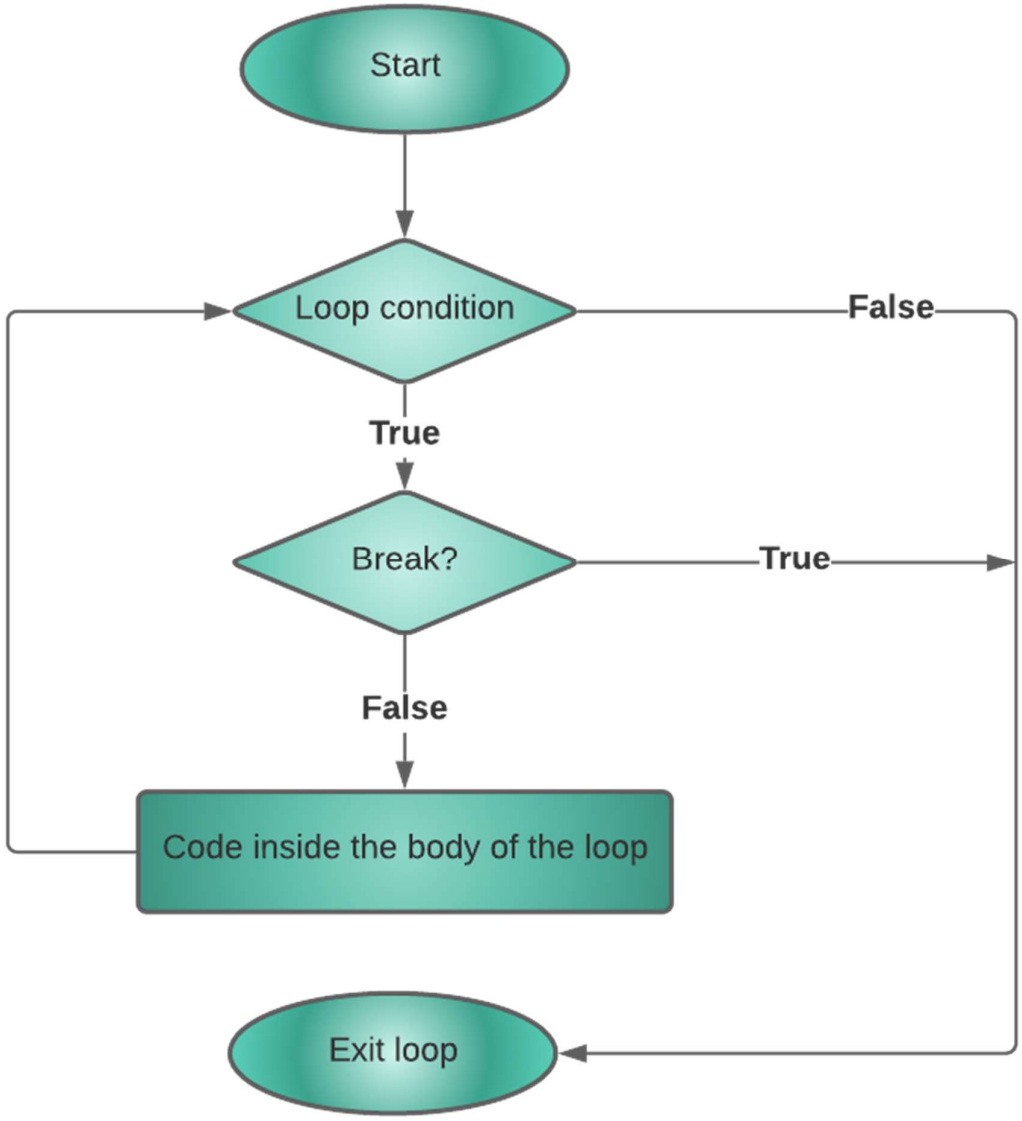
Copy

Output :

```
10
9
8
```

Copy

Flow control of break statement :



Continue statement :

- The continue statement is used to immediately move to the next iteration of the loop.
- The control is taken to the next iteration thus skipping everything below ‘continue’ inside the loop for that iteration.

Syntax :

```
continue;
```

Copy

Example to demonstrate the use of continue statement inside a for loop :

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

    for(int i=7;i>0;i--){
        if(i==3){
            continue;//continue skips the rest statement
        }
        System.out.println(i);
    }
}
}
```

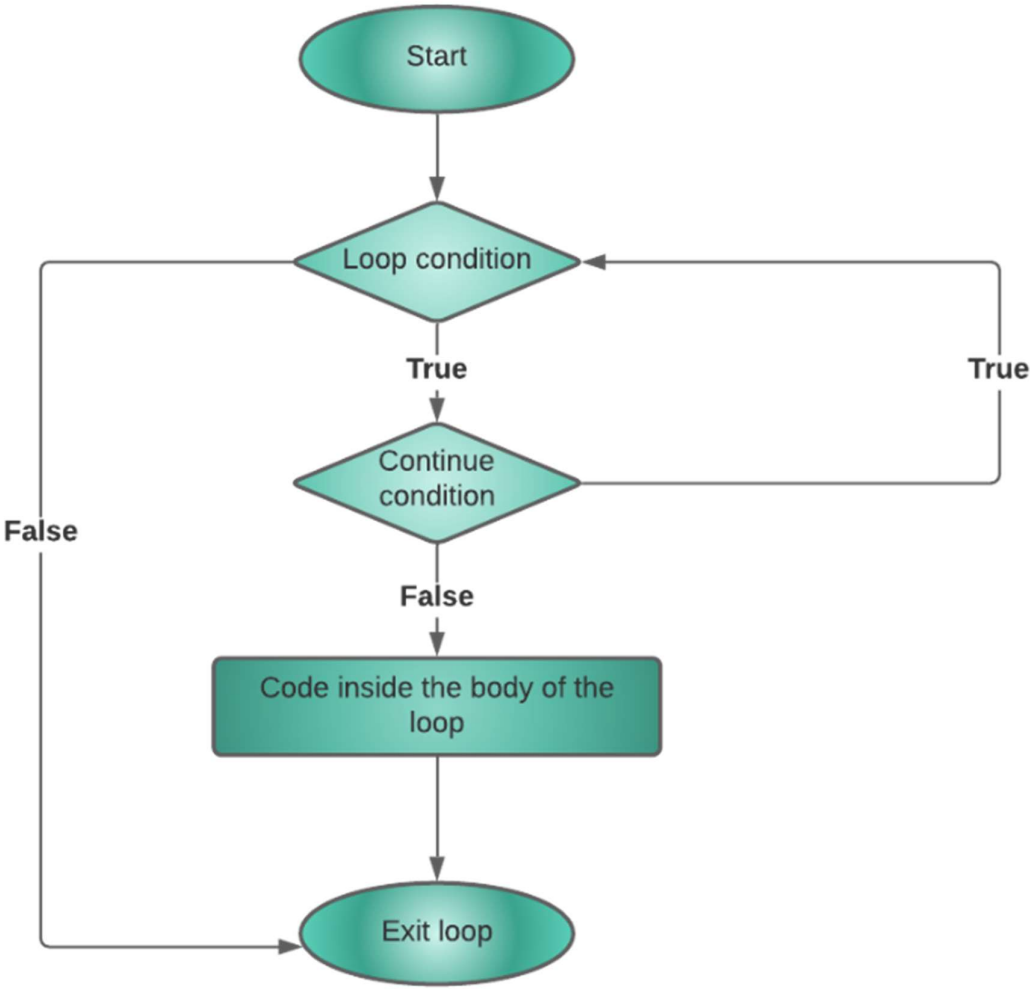
Copy

Output :

```
7
6
5
4
2
1
```

Copy

Flow control of continue statement :



In a Nut Shell ...

1. break statement completely exits the loop
2. continue statement skips the particular iteration of the loop.

Handwritten Notes: Click to Download
Ultimate Java Cheatsheet: Click To Download

Code as Described in the Video

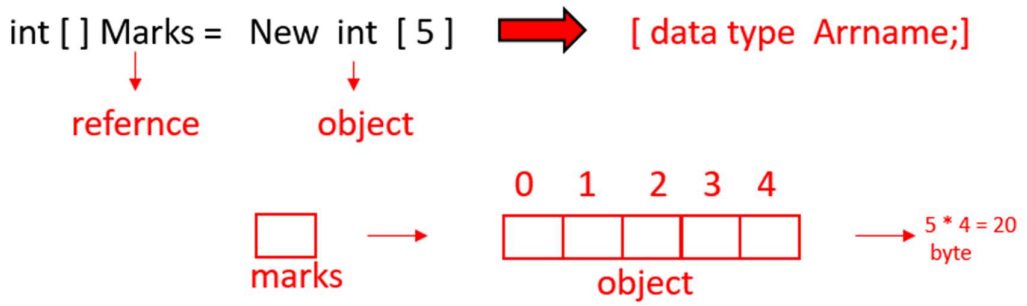
```
package com.company;

public class cwh_24_break_and_continue {
    public static void main(String[] args) {
        // Break and continue using loops!
```

```
//        for (int i=0;i<50;i++){
//            System.out.println(i);
//            System.out.println("Java is great");
//            if(i==2){
//                System.out.println("Ending the loop");
//                break;
//            }
//        }
//        int i=0;
//        do{
//            System.out.println(i);
//            System.out.println("Java is great");
//            if(i==2){
//                System.out.println("Ending the loop");
//                break;
//            }
//            i++;
//        }while(i<5);
//        System.out.println("Loop ends here");


//            for(int i=0;i<50;i++){
//            if(i==2){
//                System.out.println("Ending the loop");
//                continue;
//            }
//                System.out.println(i);
//                System.out.println("Java is great");
//        }
        int i=0;
        do{
            i++;
            if(i==2){
                System.out.println("Ending the loop");
```

```
            continue;
        }
        System.out.println(i);
        System.out.println("Java is great");

    }while(i<5);
    System.out.println("Loop ends here");
  }
}
```

Java Tutorial: Introduction to Arrays

- An array is a collection of similar types of data having contiguous memory allocation.
- The indexing of the array starts from 0., i.e 1st element will be stored at the 0th index, 2nd element at 1st index, 3rd at 2nd index, and so on.
- The size of the array can not be increased at run time therefore we can store only a fixed size of elements in array.
- Use Case: Storing marks of 5 students



Accessing Array Elements :

Array elements can be accessed as follows,

```
/* marks[0] = 100       //Note that index starts from 0
marks[1] = 70
.
.
marks[4] = 98 */
```

Copy

So in a nut shell, this is how array works:

1. int[] marks; //Declaration!

```
2. marks = new int[5];          //Memory allocation!

2. int[] marks = new int[5]; //Declaration + Memory allocation!
3. int[] marks = {100,70,80,71,98} // Declare + Initialize!
```

Note : Array indices start from 0 and go till (n-1) where n is the size of the array.

Array length :

Unlike C/C++, we don't need to use the sizeof() operator to get the length of arrays in Java because arrays are objects in Java therefore we can use the length property.

```
marks.length //Gives 5 if marks is a reference to an array with 5 elements
```

Copy

Displaying an Array :

An array can be displayed using a for loop:

```
for (int i=0; i<marks.length; i++)
{
        Sout(marks[i]);    //Array Traversal
}
```

Copy

Quick Quiz: Write a Java program to print the elements of an array in reverse order.

Code as Described in the Video

```
package com.company;


public class cwh_26_arrays {
    public static void main(String[] args) {
        /* Classroom of 500 students - You have to store marks of these 500 students
        You have 2 options:
        1. Create 500 variables
        2. Use Arrays (recommended)
        */
        // There are three main ways to create an array in Java
        // 1. Declaration and memory allocation
        // int [] marks = new int[5];
```

```
        // 2. Declaration and then memory allocation
        // int [] marks;
        // marks = new int[5];
        // Initialization
        // marks[0] = 100;
        // marks[1] = 60;
        // marks[2] = 70;
        // marks[3] = 90;
        // marks[4] = 86;

        // 3. Declaration, memory allocation and initialization together
        int [] marks = {98, 45, 79, 99, 80};

        // marks[5] = 96; - throws an error
        System.out.println(marks[4]);
    }
}
```

Java Tutorial: For Each Loop in Java

- For each loop is an enhanced version of for loop.
- It travels each element of the data structure one by one.
- Note that you can not skip any element in for loop and it is also not possible to traverse elements in reverse order with the help of for each loop.
- It increases the readability of the code.
- If you just want to simply traverse an array from start to end then it is recommended to use for each loop.

Syntax :

```
/* for (int element:Arr) {
            Sout(element);    //Prints all the elements
} */
```

Copy

Example :

```
class CWH_forEachLoop{
```

```
  public static void main(String args[]){
   //declaring an array
   int arr[]={1,2,3,3,4,5};
   //traversing the array with for-each loop
   for(int i:arr){
     System.out.println(i);
   }
  }
}
```

Copy

Output :

```
1
2
3
4
5
```

Copy

Code as Described in the Video

```
package com.company;

public class cwh_27_arrays {
    public static void main(String[] args) {

        /*
        float [] marks = {98.5f, 45.5f, 79.5f, 99.5f, 80.5f};
        String [] students ={"Harry", "Rohan", "Shubham", "Lovish"};
        System.out.println(students.length);
        System.out.println(students[2]);
        */
```

```
        int [] marks = {98, 45, 79, 99, 80};
        // System.out.println(marks.length);

        // Displaying the Array (Naive way)
        System.out.println("Printing using Naive way");
        System.out.println(marks[0]);
        System.out.println(marks[1]);
        System.out.println(marks[2]);
        System.out.println(marks[3]);
        System.out.println(marks[4]);

        // Displaying the Array (for loop)
        System.out.println("Printing using for loop");
        for(int i=0;i<marks.length;i++){
            System.out.println(marks[i]);
        }

        // Quick Quiz: Displaying the Array in Reverse order (for loop)
        System.out.println("Printing using for loop in reverse order");
        for(int i=marks.length -1;i>=0;i--){
            System.out.println(marks[i]);
        }

        // Quick Quiz: Displaying the Array (for-each loop)
        System.out.println("Printing using for-each loop");
        for(int element: marks){
            System.out.println(element);
        }

    }
}
```

Multidimensional Arrays are an Array of Arrays. Each elements of an M-D array is an array itself. Marks in the previous example was a 1-D array.

Multidimensional 2-D Array

A 2-D array can be created as follows:

```
int [][] flats = new int[2][3]          //A 2-D array of 2 rows + 3 columns
```

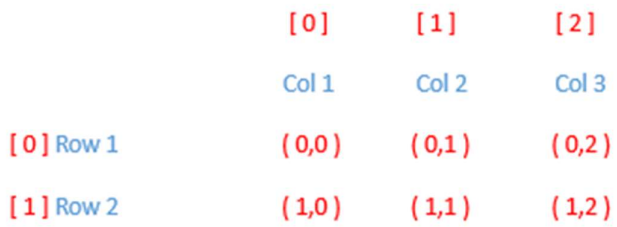
Copy

We can add elements to this array as follows

```
flats[0][0] = 100
flats[0][1] = 101
flats[0][2] = 102
// … & so on!
```

Copy

This 2-D array can be visualized as follows:



Similarly, a 3-D array can be created as follows:

```
String[][][] arr = new String [2][3][4]
package com.company;


public class cwh_28_multi_dim_arrays {
    public static void main(String[] args) {
        int [] marks; // A 1-D Array
        int [][] flats; // A 2-D Array
        flats = new int [2][3];
        flats[0][0] = 101;
        flats[0][1] = 102;
        flats[0][2] = 103;
```

```
        flats[1][0] = 201;
        flats[1][1] = 202;
        flats[1][2] = 203;

        // Displaying the 2-D Array (for loop)
        System.out.println("Printing a 2-D array using for loop");
        for(int i=0;i<flats.length;i++){
            for(int j=0;j<flats[i].length;j++) {
                System.out.print(flats[i][j]);
                System.out.print(" ");
            }
            System.out.println("");
        }

    }
}
```

Java Tutorial: Methods in Java

- Sometimes our program grows in size, and we want to separate the logic of the main method from the other methods.
- For instance, if we calculate the average of a number pair 5 times, we can use methods to avoid repeating the logic. [DRY – Don't Repeat Yourself]

Syntax of a Method

A method is a function written inside a class. Since Java is an object-oriented language, we need to write the method inside some class.

Syntax of a method :

```
returnType nameOfMethod() {
//Method body
}
```

Copy

The following method returns the sum of two numbers

```
int mySum(int a, int b) {
            int c = a+b;
```

```
        return c;  //Return value
}
```

Copy

- In the above method, int is the return data type of the mySum function.
- mySum takes two parameters: int a and int b.
- The sum of two values integer values(a and b) is stored in another integer value named 'c'.
- mySum returns c.

Calling a Method :

A method can be called by creating an object of the class in which the method exists followed by the method call:

```
Calc obj = new Calc(); //Object Creation


obj.mySum(a , b); //Method call upon an object
```

Copy

The values from the method call (a and b) are copied to the a and b of the function mySum. Thus even if we modify the values a and b inside the method, the values in the main method will not change.

Void return type :

When we don't want our method to return anything, we use void as the return type.

Static keyword :

- The static keyword is used to associate a method of a given class with the class rather than the object.
- You can call a static method without creating an instance of the class.
- In Java, the main() method is static, so that JVM can call the main() method directly without allocating any extra memory for object creation.
- All the objects share the static method in a class.

Process of method invocation in Java :

Consider the method Sum of the calculate class as given in the below code :

```
class calculate{
    int sum(int a,int b){
        return a+b;
    }
```

Copy

The method is called like this:

```
class calculate{
    int sum(int a,int b){
        return a+b;
    }

public static void main(String[] args) {

        calculate obj = new calculate();
        int c = obj.sum(5,4);
        System.out.println(c);
    }
}
```

Copy

Output :

```
9
```

Copy

- Inside the main() method, we've created an object of the calculate class.
- obj is the name of the calculate class.
- Then, we've invoked the sum method and passed 5 and 4 as arguments.

Note: In the case of Arrays, the reference is passed. The same is the case for object passing to methods.

Source code as described in the video:

```
package com.company;

public class cwh_31_methods {

    static int logic(int x, int y){
        int z;
        if(x>y){
            z = x+y;
```

```
    }
    else {
        z = (x +y) * 5;
    }
    x = 566;
    return z;
  }


  public static void main(String[] args) {
    int a = 5;
    int b = 7;
    int c;
    // Method invocation using Object creation
    //cwh_31_methods obj = new cwh_31_methods();
    //c = obj.logic(a, b);
    c = logic(a, b);
    System.out.println(a + " "+ b);
    int a1 = 2;
    int b1 = 1;
    int c1;
    c1 = logic(a1, b1);
    System.out.println(c);
    System.out.println(c1);
  }
}
```

Java Tutorial: Method Overloading in Java

- In Java, it is possible for a class to contain two or more methods with the same name but with different parameters. Such methods are called Overloaded methods.
- Method overloading is used to increase the readability of the program.

```
void foo()
void foo(int a)       //Overloaded function foo
```

```
int foo(int a, int b)
```

Copy

Ways to perform method overloading :

In Java, method overloading can be performed by two ways listed below :

1. By changing the return type of the different methods
2. By changing the number of arguments accepted by the method

Now, let's have an example to understand the above ways of method overloading :

a. By changing the return type :

- In the below example, we've created a class named calculate.
- In the calculate class, we've two methods with the same name i.e. multiply
- These two methods are overloaded because they have the same name but their return is different.
- The return type of 1st method is int while the return type of the other method is double.

```
class calculate{
    int multiply(int a,int b){
        return a*b;
    }
    double multiply(double a,double b){
        return  a*b;
    }

public static void main(String[] args)  {

        calculate obj = new calculate();
        int c = obj.multiply(5,4);
        double d = obj.multiply(5.1,4.2);
        System.out.println("Mutiply method : returns integer : " + c);
        System.out.println("Mutiply method : returns double : " +  d);

    }
}
```

Copy

Output :

```
Mutiply method : returns integer : 20
Mutiply method : returns double : 21.419999999999998
```

Copy

b.

c. By changing the number of arguments passed :

- Again, we've created two methods with the same name i.e., multiply
- The return type of both the methods is int.
- But, the first method 2 arguments and the other method accepts 3 arguments.

Example :

```
class calculate{
    int multiply(int a,int b){
        return a*b;
    }
   int multiply(int a,int b,int c){
        return  a*b*c;
    }

public static void main(String[] args) {

        calculate obj = new calculate();
        int c = obj.multiply(5,4);
        int d = obj.multiply(5,4,3);
        System.out.println(c);
        System.out.println(d);

}
}
```

Copy

Output :

```
20
60
```

Copy

Note: Method overloading cannot be performed by changing the return type of methods.

Source code as described in the video:

```
package com.company;

public class cwh_32_method_overloading {
    static void foo(){
        System.out.println("Good Morning bro!");
    }

    static void foo(int a){
        System.out.println("Good morning " + a + " bro!");
    }

    static void foo(int a, int b){
        System.out.println("Good morning " + a + " bro!");
        System.out.println("Good morning " + b + " bro!");
    }

    static void foo(int a, int b, int c){
        System.out.println("Good morning " + a + " bro!");
        System.out.println("Good morning " + b + " bro!");
    }

    static void change(int a){
        a = 98;
    }

    static void change2(int [] arr){
```

```
        arr[0] = 98;
    }
    static void tellJoke(){
        System.out.println("I invented a new word!\n" +
                "Plagiarism!");
    }

    public static void main(String[] args) {
        // tellJoke();

        // Case 1: Changing the Integer
        //int x = 45;
        //change(x);
        //System.out.println("The value of x after running change is: " + x);

        // Case 1: Changing the Array
        // int [] marks = {52, 73, 77, 89, 98, 94};
        // change2(marks);
        // System.out.println("The value of x after running change is: " + marks[0]);



        // Method Overloading
        foo();
        foo(3000);
        foo(3000, 4000);
        // Arguments are actual!



    }
}
```

Java Tutorial: Variable Arguments (VarArgs) in Java

- In the previous tutorial, we discussed how we can overload the methods in Java.

- Now, let's suppose you want to overload an "add" method. The "add" method will accept one argument for the first time and every time the number of arguments passed will be incremented by 1 till the number of arguments is equaled to 10.
- One approach to solve this problem is to overload the "add" method 10 times. But is it the optimal approach? What if I say that the number of arguments passed will be incremented by 1 till the number of arguments is equaled to 1000. Do you think that it is good practice to overload a method 1000 times?
- To solve this problem of method overloading, Variable Arguments(Varargs) were introduced with the release of JDK 5.
- With the help of Varargs, we do not need to overload the methods.

Syntax :

```
/*
public static void foo(int … arr)
{
// arr is available here as int[] arr
}
*/
```

Copy

- foo can be called with zero or more arguments like this:
  - foo(7)
  - foo(7,8,9)
  - foo(1,2,7,8,9)

Example of Varargs In Java :

```
class calculate {

   static int add(int ...arr){
       int result = 0;
       for (int a : arr){
           result = result + a;
       }
       return result;
}

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

```
    System.out.println(add(1,2));
    System.out.println(add(2,3,4));
    System.out.println(add(4,5,6));
}
}
```

Copy

Output :

```
3
9
15
```

Java Tutorial: Recursion in Java

One does not simply understand RECURSION without understanding RECURSION.

- In programming, recursion is a technique through which a function calls itself.
- With the help of recursion, we can break down complex problems into simple problems.
- Example: Factorial of a number

```
//factorial(n) = n*factorial(n-1)                [n >= 1]
```

Copy

Now, let's see an example to see the beauty of recursion in programming. First, we will print numbers from 1 to n and then n to 1 using recursion.

Program for printing 1 to n :

```
class recursion {
    static void fun2(int n){
        if(n>0){
            fun2(n-1);
            System.out.println(n);
        }
}
```

```
public static void main(String[] args){
        int n = 3;
        fun2(n);
}
}
```

Copy

Output :

```
1
2
3
```

Copy

In the above code, the print statement is getting executed at returning time. Watch the video given below to get the proper understanding of the recursive tree for the above program :

```
class recursion {
    static void fun1(int n){
        if(n>0){
            System.out.println(n);
            fun1(n-1);
        }
}
public static void main(String[] args){
        int n = 3;
        fun1(n);
}
}
```

Copy

Output :

```
3
2
1
```

Copy

In the above recursive code, the print statement is getting executed at the calling time. Before the recursive function is called, printing was done. Watch the video given below to get the proper understanding of the recursive tree for the above program :
Notice that by just changing the order of the print statement, the output of the code is completely reversed. This is the beauty of recursion. The same trick can be used to reverse a linked list.
Quick Quiz: Write a program to calculate (recursion must be used) factorial of a number in Java?

```
package com.company;

public class cwh_34_recursion {
    // factorial(0) = 1
    // factorial(n) = n * n-1 *....1
    // factorial(5) = 5 * 4 * 3 * 2 * 1 = 120
    // factorial(n) = n * factorial(n-1)

    static int factorial(int n){
        if(n==0 || n==1){
            return 1;
        }
        else{
            return n * factorial(n-1);
        }
    }
    static int factorial_iterative(int n){
        if(n==0 || n==1){
            return 1;
        }
        else{
            int product = 1;
            for (int i=1;i<=n;i++){ // 1 to n
                product *= i;
            }
            return product;
        }
```

```
    }
    public static void main(String[] args) {
        int x = 0;
        System.out.println("The value of factorial x is: " + factorial(x));
        System.out.println("The value of factorial x is: " + factorial_iterative(x));
    }
}
```

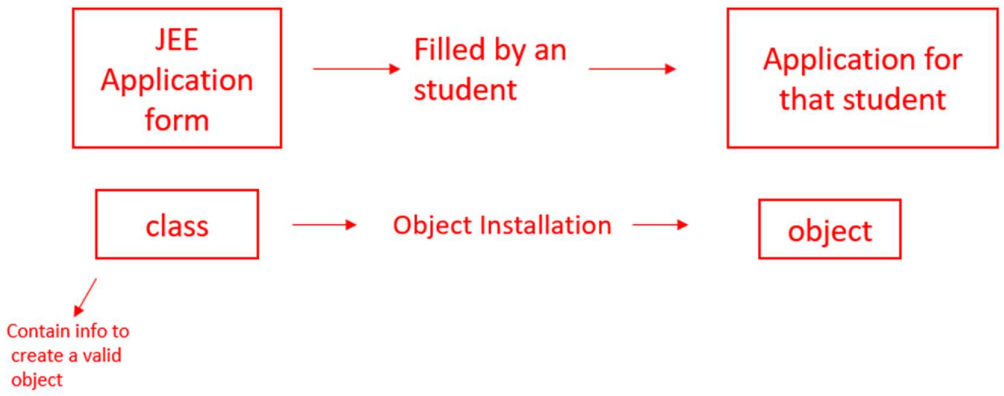
Java Tutorial: Introduction to Object Oriented Programming

- Object-Oriented Programming tries to map code instructions with real-world, making the code short and easier to understand.
- With the help of OOPs, we try to implement real-world entities such as object, inheritance, abstraction, etc.
- OOPs helps us to follow the DRY(Don't Repeat Yourself) approach of programming, which in turn increases the reusability of the code.

Two most important aspects of OOPs - Classes & Objects :

Class :

- A class is a blueprint for creating objects.
- Classes do not consume any space in the memory.
- Objects inherit methods and variables from the class.
- It is a logical component.



Objects :

- An object is an instantiation of a class. When a class is defined, a template (info) is defined.
- Every object has some address, and it occupies some space in the memory.
- It is a physical entity.

Take a look at the below example to get a better understanding of objects and classes :

How to model a problem in OOPs

We identify the following:

- Noun - Class - Employee
- Adjective - Attributes - name, age, salary
- Verb - Methods - getSalary(), increment()

This is all for this tutorial. We will do a detailed discussion on every aspect of OOPs in further tutorials.

Java Tutorial: Basic Terminologies in Object Oriented Programming

Four pillars of Object-Oriented-Programming Language :

1. *Abstraction :*
    - Let's suppose you want to turn on the bulb in your room. What do you do to switch on the bulb. You simply press the button and the light bulb turns on. Right? Notice that here you're only concerned with your final result, i.e., turning on the light bulb. You do not care about the circuit of the bulb or how current flows through the bulb. The point here is that you press the switch, the bulb turns on! You don't know how the bulb turned on/how the circuit is made because all these details are hidden from you. This phenomenon is known as abstraction.
    - More formally, data abstraction is the way through which only the essential info is shown to the user, and all the internal details remain hidden from the user.

Example :

→ Use this phone without bothering about how it was made

2. *Polymorphism :*

- One entity many forms.
- The word polymorphism comprises two words, poly which means many, and morph, which means forms.
- In OOPs, polymorphism is the property that helps to perform a single task in different ways.
- Let us consider a real-life example of polymorphism. A woman at the same time can be a mother, wife, sister, daughter, etc. Here, a woman is an entity having different forms.
- Let's take another example, a smartphone can work like a camera as well as like a calculator. So, you can see the a smartphone is an entity having different forms. Also :

→ Laptop is a single entity with Wifi + Speaker + storage in a single box!

3. *Encapsulation :*

- The act of putting various components together (in a capsule).
- In java, the variables and methods are the components that are wrapped inside a single unit named class.
- All the methods and variables of a class remain hidden from any other class.
- A automatic cold drink vending machine is an example of encapsulation.
- Cold drinks inside the machine are data that is wrapped inside a single unit cold drink vending machine.

4. *Inheritance :*

- The act of deriving new things from existing things.
- In Java, one class can acquire all the properties and behaviours of other some other class
- The class which inherits some other class is known as child class or sub class.
- The class which is inherited is known as parent class or super class.

- Inheritance helps us to write more efficient code because it increases the reusablity of the code.
- Example :
- Rickshaw → E-Rickshaw
- Phone → Smart Phone

Java Tutorial: Creating Our Own Java Class

Writing a Custom Class :

Syntax of a custom class :

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

Copy

Example :

```
public class Employee {
        int id;                         // Attribute 1
        String name;                    // Attribute 2
}
```

Copy

Note: The first letter of a class should always be capital.

- Any real-world object = Properties + Behavior
- Object in OOPs = Attributes + Methods

A Class with Methods :

We can add methods to our class Employee as follows:

```
/*
public class Employee {
        public int id;
        public String name;

        public int getSalary(){
                //code
```

```
                }
        public void getDetails(){
                //code
                }
        };
*/
```

Code as Described in the Video

```
package com.company;

class Employee{
    int id;
    int salary;
    String name;
    public void printDetails(){
        System.out.println("My id is " + id);
        System.out.println("and my name is "+ name);
    }

    public int getSalary(){
        return salary;
    }
}

public class cwh_38_custom_class {
    public static void main(String[] args) {
        System.out.println("This is our custom class");
        Employee harry = new Employee(); // Instantiating a new Employee Object
        Employee john = new Employee(); // Instantiating a new Employee Object

        // Setting Attributes for Harry
        harry.id = 12;
        harry.salary = 34;
        harry.name = "CodeWithHarry";
```

```
        // Setting Attributes for John
        john.id = 17;
        john.salary = 12;
        john.name = "John Khandelwal";

        // Printing the Attributes
        harry.printDetails();
        john.printDetails();
        int salary = john.getSalary();
        System.out.println(salary);
        // System.out.println(harry.id);
        // System.out.println(harry.name);
    }
}
```

Java Tutorial: Access modifiers, getters & setters in Java

Access Modifiers

Access Modifiers specify where a property/method is accessible. There are four types of access modifiers in java :

1. private
2. default
3. protected
4. public

Access Modifier	within class	within package	outside package by subclass only	outside package
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Default	Y	Y	N	N
private	Y	N	N	N

From the above table, notice that the private access modifier can only be accessed within the class. So, let's try to access private modifiers outside the class :

```
class Employee {
```

```
    private int id;
    private  String name;

}

public  class CWH {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        emp1.id = 3;
        emp1.name = "Shubham";

    }
}
```

Copy

Output :

```
java: id has private access in Employee
```

Copy

You can see that the above code produces an error that we're trying to access a private variable outside the class. So, is there any way by which we can access the private access modifiers outside the class? The answer is Yes! We can access the private access modifiers outside the class with the help of getters and setters.

Getters and Setters :

- Getter ➳ Returns the value [accessors]
- setter ➳ Sets / updates the value [mutators]

In the below code, we've created total 4 methods:

1. setName(): The argument passed to this method is assigned to the private variable name.
2. getName(): The method returns the value set by the setName() method.
3. setId(): The integer argument passed to this method is assigned to the private variable id.
4. getId): This method returns the value set by the setId() method.

```
class Employee {
```

```
    private int id;
    private  String name;

    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }
    public void setId(int i){
        id = i;
    }
    public int getId(){
        return id;
    }
}

public  class CWH {
    public static void main(String[] args) {
        Employee emp1 = new Employee();

        emp1.setName("Shubham");
        System.out.println(emp1.getName());
        emp1.setId(1);
        System.out.println(emp1.getId());

    }
}
```

Copy

Output :

```
Shubham
```

```
1
```

Copy

As you can see that we've got our expected output. So, that's how we use the getters and setters method to get and set the values of private access modifiers outside the class.

Source code as described in the video:

```
package com.company;

class MyEmployee{
    private int id;
    private String name;

    public String getName(){
        return name;
    }
    public void setName(String n){
        this.name = n;
    }
    public void setId(int i){
        this.id = i;
    }
    public int getId(){
        return id;
    }
}
public class cwh_40_ch9 {
    public static void main(String[] args) {
        MyEmployee harry = new MyEmployee();
        // harry.id = 45;
        // harry.name = "CodeWithHarry"; --> Throws an error due to private access modifier
        harry.setName("CodeWithHarry");
        System.out.println(harry.getName());
        harry.setId(234);
        System.out.println(harry.getId());
```

```
    }
}
```

Java Tutorial: Constructors in Java

Constructors in Java :

- Constructors are similar to methods,, but they are used to initialize an object.
- Constructors do not have any return type(not even void).
- Every time we create an object by using the new() keyword, a constructor is called.
- If we do not create a constructor by ourself, then the default constructor(created by Java compiler) is called.

Rules for creating a Constructor :

1. The class name and constructor name should be the same.
2. It must have no explicit return type.
3. It can not be abstract, static, final, and synchronized.

Types of Constructors in Java :

There are two types of constructors in Java :

1. Defaut constructor : A constructor with 0 parameters is known as default constructor.

Syntax :

```
<class_name>(){
//code to be executed on the execution of the constructor
}
```

Copy

Example :

```
class CWH {
    CWH(){
        System.out.println("This is the default constructor of CWH class.");
    }

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

```
        CWH obj1 = new CWH();


    }
}
```

Copy

Output :

```
This is the default constructor of CWH class.
```

Copy

In the above code, CWH() is the constructor of class CWH The CWH() constructor is invoked automatically with the creation of object ob1.

2. Paramerterized constructor : A constructor with some specified number of parameters is known as a parameterized constructor.

Syntax :

```
<class-name>(<data-type> param1, <data-type> param2,.....){
//code to be executed on the invocation of the constructor
}
```

Copy

Example :

```
class CWH {
    CWH(String s, int b){


        System.out.println("This is the " +b+ "th video of "+ " "+ s);
    }


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

```
        CWH obj1 = new CWH("CodeWithHarry Java Playlist",42);


    }
}
```

Copy

Output :

```
This is the 42th video of  CodeWithHarry Java Playlist
```

Copy

In the above example, CWH() constructor accepts two parameters i.e., string s and int b.

Constructor Overloading in Java :

Just like methods, constructors can also be overloaded in Java. We can overload the Employe constructor like below:

```
public Employee (String n)
        name = n;
    }
```

Copy

Note:

1. Constructors can take parameters without being overloaded
2. There can be more than two overloaded constructors

Let's take an example to understand the concept of constructor overloading.

Example :

In the below example, the class Employee has a constructor named Employee(). It takes two argument,i.e., string s & int i. The same constructor is overloaded and then it accepts three arguments i.e., string s, int i & int salary.

```
class Employee {
// First constructor
   Employee(String s, int i){
      System.out.println("The name of the first employee is : " + s);
      System.out.println("The id of the first employee is : " + i);
```

```
    }
//    Constructor overloaded
    Employee(String s, int i, int salary){
        System.out.println("The name of the second employee is : " + s);
        System.out.println("The id of the second employee is : " + i);
        System.out.println("The salary of second employee is : " + salary);
    }

}
public class CWH_constructors {
    public static void main(String[] args) {
        Employee shubham = new Employee("Shubham",1);
        Employee harry = new Employee("Harry",2,70000);




    }
}
```

Copy

Output :

```
The name of the first employee is : Shubham
The id of the first employee is : 1
The name of the second employee is : Harry
The id of the second employee is : 2
The salary of second employee is : 70000
```

Quick quiz: Overloaded the employee constructor to initialize the salary to Rs 10,000

Source code as described in the video:

```
package com.company;

class MyMainEmployee{
    private int id;
    private String name;
```

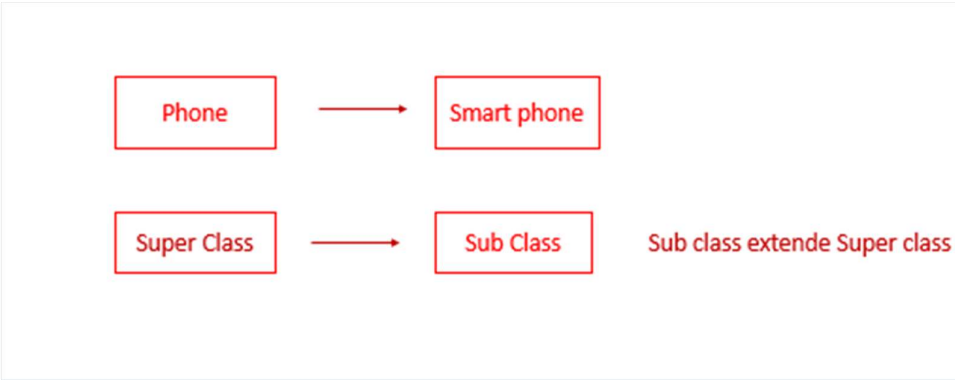
```
    public MyMainEmployee(){
        id = 0;
        name = "Your-Name-Here";
    }
    public MyMainEmployee(String myName, int myId){
        id = myId;
        name = myName;
    }
    public MyMainEmployee(String myName){
        id = 1;
        name = myName;
    }
    public String getName(){
        return name;
    }
    public void setName(String n){
        this.name = n;
    }
    public void setId(int i){
        this.id = i;
    }
    public int getId(){
        return id;
    }
}

public class cwh_42_constructors {
    public static void main(String[] args) {
        //MyMainEmployee harry = new MyMainEmployee("ProgrammingWithHarry", 12);
        MyMainEmployee harry = new MyMainEmployee();
        //harry.setName("CodeWithHarry");
        //harry.setId(34);
        System.out.println(harry.getId());
```

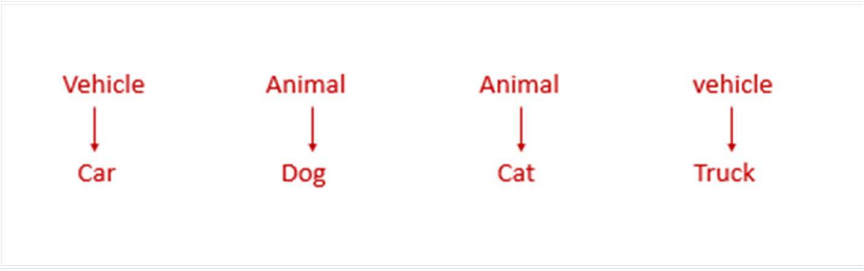
```
        System.out.println(harry.getName());
    }
}
```

Inheritance in Java

- You might have heard people saying your nose is similar to your father or mother. Or, more formally, we can say that you've inherited the genes from your parents due to which you look similar to them.
- The same phenomenon of inheritance is also valid in programming.
- In Java, one class can easily inherit the attributes and methods from some other class. This mechanism of acquiring objects and properties from some other class is known as inheritance in Java.
- Inheritance is used to borrow properties & methods from an existing class.
- Inheritance helps us create classes based on existing classes, which increases the code's reusability.

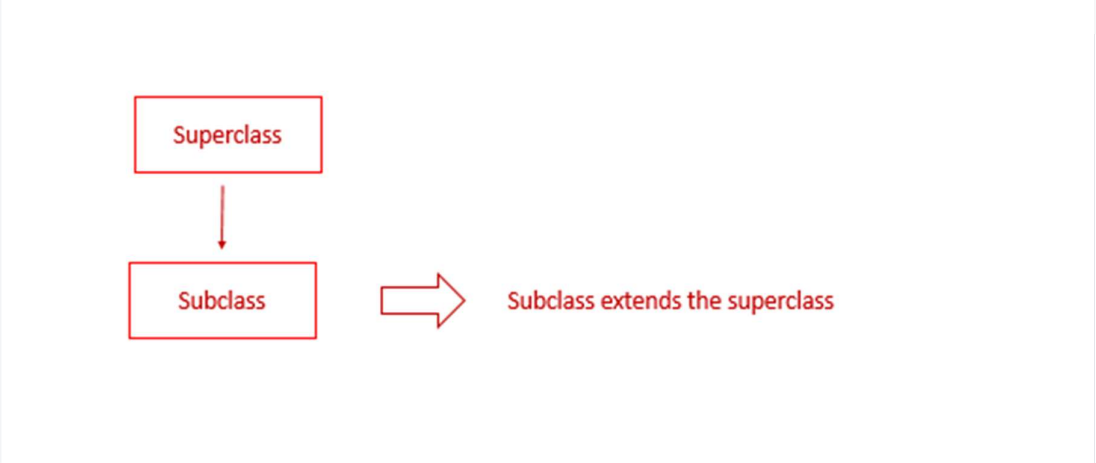
Examples :





Important terminologies used in Inheritance :

1. Parent class/superclass: The class from which a class inherits methods and attributes is known as parent class.
2. Child class/sub-class: The class that inherits some other class's methods and attributes is known as child class.



Extends keyword in inheritance :

- The extends keyword is used to inherit a subclass from a superclass.

Syntax :

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Copy

Example :

```
public class dog extends Animal {
            // code
            }
```

Copy

Note: Java doesn't support multiple inheritances, i.e., two classes cannot be the superclass for a subclass.
Quick quiz: Create a class Animal and Derive another class dog from it

```
package com.company;
```

```
class Base{
    public int x;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        System.out.println("I am in base and setting x now");
        this.x = x;
    }

    public void printMe(){
        System.out.println("I am a constructor");
    }
}

class Derived extends Base{
    public int y;

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

public class cwh_45_inheritance {
    public static void main(String[] args) {
        // Creating an Object of base class
        Base b = new Base();
        b.setX(4);
```

```
        System.out.println(b.getX());

        // Creating an object of derived class
        Derived d = new Derived();
        d.setY(43);
        System.out.println(d.getY());
    }
}
```

Constructors in Inheritance in Java

Constructors in Inheritance:

When a drived class is extended from the base class, the constructor of the base class is executed first followed by the constructor of the derived class. For the following Inheritance hierarchy , the constructors are executed in the order:

1. C1- Parent
2. C2 - Child
3. C3 - Grandchild

Constructors during constructor overloading :

- When there are multiple constructors in the parent class, the constructor without any parameters is called from the child class.
- If we want to call the constructor with parameters from the parent class, we can use the super keyword.
- super(a, b) calls the constructor from the parent class which takes 2 variables

Source code as described in the video:

```
package com.company;

class Base1{
    Base1(){
        System.out.println("I am a constructor");
    }
    Base1(int x){
        System.out.println("I am an overloaded constructor with value of x as: " + x);
    }
}
```

```
class Derived1 extends Base1{
    Derived1(){
        //super(0);
        System.out.println("I am a derived class constructor");
    }
    Derived1(int x, int y){
        super(x);
        System.out.println("I am an overloaded constructor of Derived with value of y as: " +
y);
    }
}

class ChildOfDerived extends  Derived1{
    ChildOfDerived(){
        System.out.println("I am a child of derived constructor");
    }
    ChildOfDerived(int x, int y, int z){
        super(x, y);
        System.out.println("I am an overloaded constructor of Derived with value of z as: " + z);
    }
}
public class cwh_46_constructors_in_inheritance {
    public static void main(String[] args) {
        // Base1 b = new Base1();
        // Derived1 d = new Derived1();
        // Derived1 d = new Derived1(14, 9);
        // ChildOfDerived cd = new ChildOfDerived();
        ChildOfDerived cd = new ChildOfDerived(12, 13, 15);
    }
}
```

his and super keyword in Java

this keyword in Java :

- this is a way for us to reference an object of the class which is being created/referenced.
- It is used to call the default constructor of the same class.
- this keyword eliminates the confusion between the parameters and the class attributes with the same name. Take a look at the example given below :

```
class cwh{
    int x;

//    getter of x
    public  int getX(){
        return  x;
    }

    // Constructor with a parameter
    cwh(int x) {
        x = x;
    }

    // Call the constructor
    public static void main(String[] args) {
        cwh obj1 = new cwh(65);
        System.out.println(obj1.getX());

    }
}
```

Copy

Output :

```
0
```

Copy

- In the above example, the expected output is 65 because we've passed x=65 to the constructor of the cwh class. But the compiler fails to differentiate between the parameter 'x' & class attribute 'x.' Therefore, it returns 0.
- Now, let's see how we can handle this situation with the help of this keyword. Take a look at the below code :

```
class cwh{
    int x;

//    getter of x
    public  int getX(){
        return  x;
    }

    // Constructor with a parameter
    cwh(int x) {
        this.x = x;
    }

    // Call the constructor
    public static void main(String[] args) {
        cwh obj1 = new cwh(65);
        System.out.println(obj1.getX());

    }
}
```

Copy

Output :

```
65
```

Copy

Now, you can see that we've got the desired output

Super keyword

- A reference variable used to refer immediate parent class object.
- It can be used to refer immediate parent class instance variable.
- It can be used to invoke the parent class method.

Source code as described in the video:

```
package com.company;
import javax.print.Doc;
class EkClass{
int a;
public int getA() {
return a;
}
EkClass(int a){
this.a = a;
 }
public int returnone(){
 return 1;
}
}
class DoClass extends EkClass{ DoClass(int c){ super(c);
System.out.println("I am a constructor"); }
}
public class cwh_47_this_super {
public static void main(String[] args) {
 EkClass e = new EkClass(65);
DoClass d = new DoClass(5);
System.out.println(e.getA()); } }
```

```
•
•            b.meth1();
•        }
•    }
```

- Copy

Output :

```
I am method 1 of class A
I am method 1 of class B
```

Copy

Source code as described in the video:

```
package com.company;

class A{
    public int a;
    public int harry(){
        return 4;
    }
    public void meth2(){
        System.out.println("I am method 2 of class A");
    }
}

class B extends A{
    @Override
    public void meth2(){
        System.out.println("I am method 2 of class B");
    }
    public void meth3(){
        System.out.println("I am method 3 of class B");
    }
}
public class cwh_48_method_overriding {
    public static void main(String[] args) {
        A a = new A();
        a.meth2();
```

```
        B b = new B();
        b.meth2();
    }
}
```

ynamic Method Dispatch in Java

- Dynamic method dispatch is also known as run time polymorphism.
- It is the process through which a call to an overridden method is resolved at runtime.
- This technique is used to resolve a call to an overridden method at runtime rather than compile time.
- To properly understand Dynamic method dispatch in Java, it is important to understand the concept of upcasting because dynamic method dispatch is based on upcasting.

Upcasting :

- It is a technique in which a superclass reference variable refers to the object of the subclass.

Example :

```
class Animal{}
class Dog extends Animal{}
```

Copy

```
Animal a=new Dog();//upcasting
```

Copy

In the above example, we've created two classes, named Animal(superclass) & Dog(subclass). While creating the object 'a', we've taken the reference variable of the parent class(Animal), and the object created is of child class(Dog).

Example to demonstrate the use of Dynamic method dispatch :

- In the below code, we've created two classes: Phone & SmartPhone.
- The Phone is the parent class and the SmartPhone is the child class.
- The method on() of the parent class is overridden inside the child class.
- Inside the main() method, we've created an object obj of the Smartphone() class by taking the reference of the Phone() class.

- When obj.on() will be executed, it will call the on() method of the SmartPhone() class because the reference variable obj is pointing towards the object of class SmartPhone().

```
class Phone{
    public void showTime(){
        System.out.println("Time is 8 am");
    }
    public void on(){
        System.out.println("Turning on Phone...");
    }
}

class SmartPhone extends Phone{
    public void music(){
        System.out.println("Playing music...");
    }
    public void on(){
        System.out.println("Turning on SmartPhone...");
    }
}
public class CWH {
    public static void main(String[] args) {

        Phone obj = new SmartPhone(); // Yes it is allowed
        // SmartPhone obj2 = new Phone(); // Not allowed

        obj.showTime();
        obj.on();
        // obj.music(); Not Allowed



    }
}
```

Copy

Output :

```
Time is 8 am
Turning on SmartPhone...
```

Copy

Note: The data members can not achieve the run time polymorphism.

Code as described/written in the video :

```
package com.company;
class Phone{
    public void showTime(){
        System.out.println("Time is 8 am");
    }
    public void on(){
        System.out.println("Turning on Phone...");
    }
}

class SmartPhone extends Phone{
    public void music(){
        System.out.println("Playing music...");
    }
    public void on(){
        System.out.println("Turning on SmartPhone...");
    }
}
public class cwh_49_dynamic_method_dispatch {
    public static void main(String[] args) {
        // Phone obj = new Phone(); // Allowed
        // SmartPhone smobj = new SmartPhone(); // Allowed
        // obj.name();

        Phone obj = new SmartPhone(); // Yes it is allowed
```

```
        // SmartPhone obj2 = new Phone(); // Not allowed

        obj.showTime();
        obj.on();
        // obj.music(); Not Allowed


    }
}
```

Java Tutorial: Abstract Class & Abstract Methods

What does Abstract mean?

Abstract in English means existing in through or as an idea without concrete existence.

Abstract class :

- An abstract class cannot be instantiated.
- Java requires us to extend it if we want to access it.
- It can include abstract and non-abstract methods.
- If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class phone Model {
        abstract void switch off ();
        || more code
        }
```

Copy

- Abstract class are used when we want to achieve security & abstraction(hide certain details & show only necessary details to the user)

Example :

```
abstract class Phone{
  abstract void on();
}
class SmartPhone extends Phone{
void run(){
```

```
    System.out.println("Turning on...");
    }
    public static void main(String args[]){
     Phone obj = new SmartPhone();
     obj.on();
    }
    }
```

Copy

Output :

```
Turning on...
```

Abstract method :

- A method that is declared without implementation is known as the abstract method.
- An abstract method can only be used inside an abstract class.
- The body of the abstract method is provided by the class that inherits the abstract class in which the abstract method is present.
- In the above example, on() is the abstract method.

Code as described/written in the video :

```
package com.company;


abstract class Parent2{
    public Parent2(){
        System.out.println("Mai base2 ka constructor hoon");
    }
    public void sayHello(){
        System.out.println("Hello");
    }
    abstract public void greet();
    abstract public void greet2();
}


class Child2 extends Parent2{
```

```
    @Override
    public void greet(){
        System.out.println("Good morning");
    }
    @Override
    public void greet2(){
        System.out.println("Good afternoon");
    }
}

abstract class Child3 extends Parent2{
    public void th(){
        System.out.println("I am good");
    }
}
public class cwh_53_abstract {
    public static void main(String[] args) {
        //Parent2 p = new Parent2(); -- error
        Child2 c = new Child2();
        //Child3 c3 = new Child3(); -- error
    }
}
```

Java Tutorial: Introduction to Interfaces

Interfaces in Java :

- Just like a class in java is a collection of the related methods, an interface in java is a collection of abstract methods.
- The interface is one more way to achieve abstraction in Java.
- An interface may also contain constants, default methods, and static methods.
- All the methods inside an interface must have empty bodies except default methods and static methods.
- We use the `interface` keyword to declare an interface.
- There is no need to write `abstract` keyword before declaring methods in an interface because an interface is implicitly abstract.
- An interface cannot contain a constructor (as it cannot be used to create objects)
- In order to implement an interface, java requires a class to use the `implement` keyword.

Example to demonstrate Interface in Java :

```
interface Bicycle {
        void apply brake ( int decrement );
        void  speed up ( int increment );
      }

class Avon cycle implements Bicycle {
        int speed = 7 ;
         void apply brake ( int decrement ) {
                speed = speed - decrement ;
                }
      void speedup ( int increment ){
              speed = speed + increment ;
              }
```

Copy

Code as described/written in the video :

```
package com.company;

interface Bicycle{
   int a = 45;
   void applyBrake(int decrement);
   void speedUp(int increment);
}

interface HornBicycle{
   int x = 45;
   void blowHornK3g();
   void blowHornmhn();
}

class AvonCycle implements Bicycle, HornBicycle{
   //public int x = 5;
```

```
    void blowHorn(){
        System.out.println("Pee Pee Poo Poo");
    }
    public void applyBrake(int decrement){
        System.out.println("Applying Brake");
    }
    public void speedUp(int increment){
        System.out.println("Applying SpeedUP");
    }
    public void blowHornK3g(){
        System.out.println("Kabhi khushi kabhi gum pee pee pee pee");
    }
    public void blowHornmhn(){
        System.out.println("Main hoon naa po po po po");
    }
}
public class cwh_54_interfaces {
    public static void main(String[] args) {
        AvonCycle cycleHarry = new AvonCycle();
        cycleHarry.applyBrake(1);
        // You can create properties in Interfaces
        System.out.println(cycleHarry.a);
        System.out.println(cycleHarry.x);

        // You cannot modify the properties in Interfaces as they are final
        // cycleHarry.a = 454;
        //System.out.println(cycleHarry.a);

        cycleHarry.blowHornK3g();
        cycleHarry.blowHornmhn();
    }
}
```

Abstract class	Interface
1. It can contain abstract and non-abstract method	It can only contain abstract methods. We do not need to use the "abstract" keyword in interface methods because the interface is implicitly abstract.
2. abstract keyword is used to declare an abstract class.	interface keyword is used to declare an interface.
3. A sub-class extends the abstract class by using the "extends" keyword.	The "implements" keyword is used to implement an interface.
4. A abstract class in Java can have class members like private, protected, etc.	Members of a Java interface are public by default.
5. Abstract class doesn't support multiple inheritance.	Multiple inheritance is achieved in Java by using the interface.

Why multiple inheritance is not supported in java?

Is multiple inheritance allowed in Java?

- Multiple inheritance faces problems when there exists a method with the same signature in both the superclasses.
- Due to such a problem, java does not support multiple inheritance directly, but the similar concept can be achieved using interfaces.
- A class can implement multiple interfaces and extend a class at the same time.

Some Important points :

1. Interfaces in java are a bit like the class but with a significantly different.
2. An Interface can only have method signatures field and a default method.
3. The class implementing an interface needs to declare the methods (not field)
4. You can create a reference of an interface but not the object
5. Interface methods are public by default

Java Interfaces Example & Default Methods

Default methods In Java:

- An interface can have static and default methods.
- Default methods enable us to add new functionality to existing interfaces.
- This feature was introduced in java 8 to ensure backward compatibility while updating an interface.
- A class implementing the interface need not implement the default methods.
- Interfaces can also include private methods for default methods to use.
- You can easily override a default method like any other method of an interface.

Example :

```
interface Animal{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void bark();
}
public class CWH implements Animal{

    @Override
    public void bark() {
        System.out.println("Dog barks!");
    }
    public static void main(String[] args) {
        CWH obj1 = new CWH();
        obj1.bark();
        obj1.say();

    }


}
```

Copy

Output :

```
Dog barks!
Hello, this is default method
```

Copy

Code as described/written in the video :

```
package com.company;
```

```
interface MyCamera{
    void takeSnap();
    void recordVideo();
    private void greet(){
        System.out.println("Good Morning");
    }
    default void record4KVideo(){
        greet();
        System.out.println("Recording in 4k...");
    }
}

interface MyWifi{
    String[] getNetworks();
    void connectToNetwork(String network);
}

class MyCellPhone{
    void callNumber(int phoneNumber){
        System.out.println("Calling "+ phoneNumber);
    }
    void pickCall(){
        System.out.println("Connecting... ");
    }

}

class MySmartPhone extends MyCellPhone implements MyWifi, MyCamera{
    public void takeSnap(){
        System.out.println("Taking snap");
    }
    public void recordVideo(){
        System.out.println("Taking snap");
```

```
    }
//    public void record4KVideo(){
//        System.out.println("Taking snap and recoding in 4k");
//    }
    public String[] getNetworks(){
        System.out.println("Getting List of Networks");
        String[] networkList = {"Harry", "Prashanth", "Anjali5G"};
        return networkList;
    }
    public void connectToNetwork(String network){
        System.out.println("Connecting to " + network);
    }
}
public class cwh_57_default_methods {
    public static void main(String[] args) {
        MySmartPhone ms = new MySmartPhone();
        ms.record4KVideo();
        // ms.greet(); --> Throws an error!
        String[] ar = ms.getNetworks();
        for (String item: ar) {
            System.out.println(item);
        }
    }
}
```

Inheritance in Interfaces

Interfaces can extend other interfaces as shown below :

```
public interface Interface 1 {
            void meth1 ();
        }



public interface Interface 2 extends Interface 1 {
```

```
        void meth 2( );
    }
```

Copy

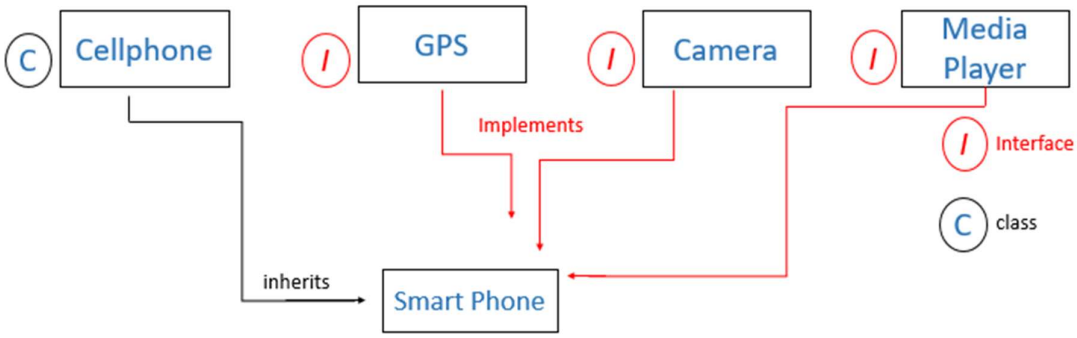
Note: Remember that interface cannot implement another interface only classes can do that!

Code as described/written in the video :

```
package com.company;

interface sampleInterface{
    void meth1();
    void meth2();
}
interface childSampleInterface extends sampleInterface{
    void meth3();
    void meth4();
}
class MySampleClass implements childSampleInterface{
    public void meth1(){
        System.out.println("meth1");
    }
    public void meth2(){
        System.out.println("meth2");
    }
    public void meth3(){
        System.out.println("meth3");
    }
    public void meth4(){
        System.out.println("meth4");
    }
}
public class cwh_58_inheritance_interfaces {
    public static void main(String[] args) {
        MySampleClass obj = new MySampleClass();
        obj.meth1();
```

```
        obj.meth2();
        obj.meth3();
    }
}
```

Java Tutorial: Polymorphism in Interfaces



GPS g = new Smartphone ( ); can only use GPS method
Smartphones = new Smartphone ( ); can only use smartphone methods
Implementing an Interface force method implementation
Code as described/written in the video :

```
package com.company;

interface MyCamera2{
    void takeSnap();
    void recordVideo();
    private void greet(){
        System.out.println("Good Morning");
    }
    default void record4KVideo(){
        greet();
        System.out.println("Recording in 4k...");
    }
}

interface MyWifi2{
```

```
    String[] getNetworks();
    void connectToNetwork(String network);
}

class MyCellPhone2{
    void callNumber(int phoneNumber){
        System.out.println("Calling "+ phoneNumber);
    }
    void pickCall(){
        System.out.println("Connecting... ");
    }

}

class MySmartPhone2 extends MyCellPhone2 implements MyWifi2, MyCamera2{
    public void takeSnap(){
        System.out.println("Taking snap");
    }
    public void recordVideo(){
        System.out.println("Taking snap");
    }
    //    public void record4KVideo(){
//        System.out.println("Taking snap and recoding in 4k");
//    }
    public String[] getNetworks(){
        System.out.println("Getting List of Networks");
        String[] networkList = {"Harry", "Prashanth", "Anjali5G"};
        return networkList;
    }
    public void connectToNetwork(String network){
        System.out.println("Connecting to " + network);
    }
    public void sampleMeth(){
        System.out.println("meth");
```

```
    }
}
public class cwh_59_polymorphism {
    public static void main(String[] args) {
        MyCamera2 cam1 = new MySmartPhone2(); // This is a smartphone but, use it as a camera
        // cam1.getNetworks(); --> Not allowed
        // cam1.sampleMeth(); --> Not allowed

        cam1.record4KVideo();

        MySmartPhone2 s = new MySmartPhone2();
        s.sampleMeth();
        s.recordVideo();
        s.getNetworks();
        s.callNumber(7979);
    }
}
```

Interpreted vs Compiled Languages!

Interpreter Vs Compliler :

The interpreter translates one statement at a time into machine code. On the other hand, the compiler scans the entire program and translates the whole of it into machine code

Interpreter :

1. one statement at a time
2. An interpreter is needed every time
3. Partial execution if an error occurs in the program.
4. Easy for programmers.

Compiler :

1. Entire program at a time
2. Once compiled, it is not needed
3. No execution if an error occurs
4. Usually not as easy as interpreted once

Is Java interpreted or compiled?

Java is a hybrid language both compiled as well as interpreted.

- A JVM can be used to interpret this bycode
- This bytecode can be taken to any platform (win/ mac / Linux) for education.
- Hence java is platform-independent (write once run everywhere).

Executing a java program

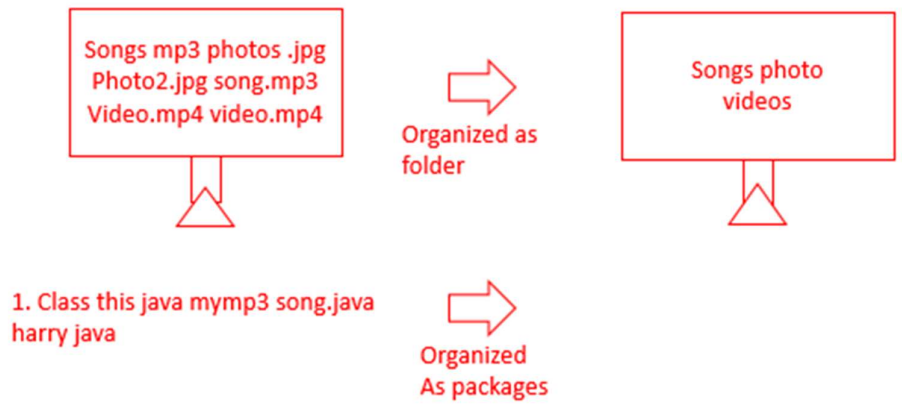
java Harry java - compiles
java Harry class - Interpreted
So far the execution of our program was being managed by IntelliJ idea.
we can download a source code like VS code to compile & execute our java programs

Packages in Java

- A package is used to group related classes.
- packages help in avoiding name conflicts

There are two types of packages :

- Build-in packages - java API
- User-defined packages - Custom packages



Using a java package :

Import keyword is used to import packages in the java program. Example :

- import java lang * - import

- import java string - import string from java long
- s= new java long string (" Harry ") - use without importing

ava Tutorial: Creating Packages in Java

How to create a package in Java :

```
javac -d Harry java
```

Copy

The above code creates a packages folder.

```
java Harry java
```

Copy

The above code creates Harry class.

- We can also create inner packages by adding packages inner as the package name.
- These packages once created can be used by other classes.

Code as described/written in the video :

```
package com.company;


import java.util.Scanner;
//import java.util.*;
public class cwh_65_packages {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // java.util.Scanner sc = new java.util.Scanner(System.in);
        int a = sc.nextInt();
        System.out.println("This is my scanner taking input as " + a);
    }
}
```

Access Modifiers in Java

Access modifiers determine whether other classes can use a particular field or invoke a particular method can be public, private, protected, or default (no modifier). See the table given below :

Access Modifier	within class	within package	outside package by subclass only	outside package
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Default	Y	Y	N	N
private	Y	N	N	N

Code as described/written in the video :

```
package com.company;


class C1{
    public int x = 5;
    protected int y =45;
    int z = 6;
    private int a = 78;
    public void meth1(){
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
        System.out.println(a);
    }
}
public class cwh_66_access_modifiers {
    public static void main(String[] args) {
        C1 c = new C1();
        // c.meth1();
        System.out.println(c.x);
        System.out.println(c.y);
        System.out.println(c.z);
        // System.out.println(c.a);
    }
}
```