__/__/____

JAVA

INTRODUCTION :-

→ Java is a high-level, robust, object-oriented and secure programming language.

→ Java was developed by Sun Microsystems (now a subsidiary of Oracle) in 1991.

→ James Gosling is known as the father of Java.

* Hello program :-

```
class Java {
    public static void main (String args []) {
        System.out.println ("Hello Java");
    }
}
```

Explanation :-

1. Class Java :- Syntax for initializing class.
   [Class classname].

2. public :- It is an access modifier. It has to be public so that java runtime can execute this method. It is made public so that JVM can invoke it from outside the class as it is not present in the current class

3. static :- It is a keyword which is when associated with a method, makes it a class related method.

The main() method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM.

4. Void :- It is a keyword used to specify that a method doesn't return anything.

5. main :- It is the name of the Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It's not a keyword.

6. String[] args :- Java main method accepts a single argument of type String array. It stores Java command line arguments and is an array of type java.lang.String class.

* Java Variables

A variable is a container which holds the value while the Java program is executed.

→ Types of Variables

1) Local variables → A variable declared inside the body of the method is called local variable. It cannot be defined with 'static' keyword.

Meaning of instance → A single occurence of something.

2. Instance variables → A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static. It is called instance variable because its value is instance specific and is not shared among instances.

3. Static variable → A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

* Data Types

- Primitive
  - → Boolean (V = false, S = 1 bit)
  - → char (V = '\u0000', S = 2 byte)
  - → byte (V = 0, S = 1 byte)
  - → short (V = 0, S = 2 byte)
  - → int (V = 0, S = 4 byte)
  - → long (V = 0L, S = 8 byte)
  - → float (V = 0.0f, S = 4 byte)
  - → double (V = 0.0d, S = 8 byte)
- Non-Primitive
  - → String
  - → Array

* V = Default value
S = Default size.

__/__/____

* Java Operators Precedence.

Operator Type	Category	Precedence
Unary	Postfix	~~++~~, expr++, expr--
	Prefix	++expr, --expr, +expr, -expr, ~, !.
Arithmetic	Multiplicative	*, /, %
	Additive	+, -
Shift	Shift	<<, >>, >>>
Relational	Comparison	<, >, <=, >=, instanceof
	Equality	==, !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	?, :
Assignment	assignment	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

__/__/____

* Java I/O (Basic Input and Output)

→ Java Input

In order to use the object of Scanner, we need to import java.util.Scanner package;.

```
// Creating an object of Scanner
    Scanner input = new Scanner (System.in);

// taking input from the user and printing.
    int number = input.nextInt();
    System.out.println (number);

// closing the Scanner object
    input.close();
```

Similarly, we can use nextLong(), nextFloat(), nextDouble(), next() methods to get long, float, double and string input respectively from the user. Also, nextLine() is used to read an entire line.

* It is recommended to close the scanner object once the input is taken.

→ Java Output

⇒ System.out

- System is a class
- out is a public static field; it accepts output data.

Now,

1> print() - It prints string inside the quotes.
2> println() - It prints string inside the quotes similarly but then the cursor moves to the beginning of next line.
3> printf() - It provides string formatting

* Java Keywords

1. abstract - Java abstract keyword is used to declare abstract class.

2. boolean - It is used to declare a variable as a boolean type.

3. break - It is used to break loop or switch statement. It breaks the current flow of the program at specified condition.

4. byte - It is used to declare a variable that can hold an 8-bit data value

5. case - It is used in switch statements to mark blocks of text.

6. catch - It is used to catch the exceptions generated by try statements. It must be used after the try block only.

7. char - It is used to declare a variable that can hold unsigned 16-bit characters.

8. class - It is used to declare a class.

9. continue - It continues the current flow of the program and skips the remaining code at the specified condition.

10. default - It is used to specify the default block of code in a switch statement.

~~11. do - It is used to specify the default to block of code :~~

11. do - It is used in control statement to declare a loop. It can iterate a part of the program several times.

12. double - It is used to declare a variable than can hold a 64-bit floating point numbers.

13. ~~It~~ else - It is used to indicate the alternative branches in an if statement.

14. enum - It is used to define a fixed set of constants. Enum constructors are always private or default.

15. extends - It is used to indicate that a class is derived from another class or interface.

16. final - It is used to indicate that a variable holds a constant value. It is used to restrict the user

17. finally - It indicates a block of code in a try-catch structure. This block is always executed whether exception is handled or not.

18. float - It is used to declare a variable that can hold a 32-bit floating point number.

19. for - It is used to execute a set of instructions / functions repeatedly when some conditions become true.

20. if - It executes the if block if condition is true.

21. implements - It is used to implement an interface.

22. import - It makes classes and interfaces available and accessible to the current source code.

23. instanceof - It is used to test whether the object is an instance of the specified class or implements an interface.

24. int - It is used to declare a variable that can hold a 32-bit signed integer.

25. interface - It is used to declare an interface. It can have only abstract methods.

__/__/____

26. long - It is used to declare a variable that can hold a 64-bit integer.

27. native - It is used to specify that a method is implemented in native code using Java Native Interface.

28. new - It is used to create new objects.

29. null - It is used to indicate that a reference does not refer to anything. It removes the garbage value.

30. package - It is used to declare a Java package that includes the classes.

31. private - It is an access modifier used to indicate that a method or variable may be accessed only in the class in which it is declared.

32. protected - It is an access modifier. It can be accessible within package and outside the package but through inheritance only.

33. public - It is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.

34. return - It is used to return from a method when its execution is complete.

35. Short - It is used to declare a variable that can hold a 16-bit integer.

36. switch - It contains a switch statement that executes code based on test value. It tests the equality of a variable against multiple values.

37. Static - It is used to indicate that a variable or method is a class method. It is used for memory management mainly.

38. super - It is a reference variable that is used to refer parent class object. It can be used to invoke immediate parent class method.

39. ~~strit~~ strictfp - It is used to restrict the floating point calculations to ensure portability

40. Synchronized - It is used to specify the critical sections or methods in multithreaded code.

41. this - It can be used to refer the current object in a method or constructor.

42. throw - It is used to explicitly throw an exception. It is mainly used to throw custom exception. It is followed by an instance.

43. throws - It is used to declare an exception. Checked exception can be propagated with throws.

44. transient - It is used in serialization. If you define any data member as transient, it will not be serialized.

45. try - It is used to start a block of code that will be tested for exceptions. It must be followed by either catch or finally block.

46. void - It is used to specify that a method does not have a return value.

47. volatile - It is used to indicate that a variable may change asynchronously.

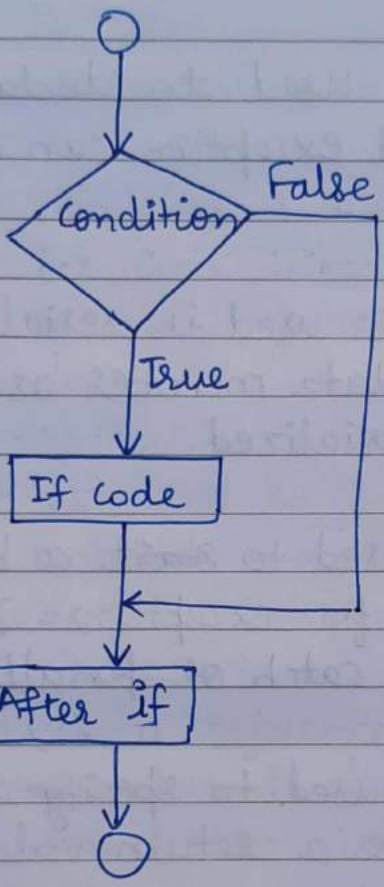
48. while - It is used to start a while loop. This loop iterates a part of the program several times. If the number of iterations is not fixed, it is recommended to use while loop.

JAVA FLOW CONTROL :-

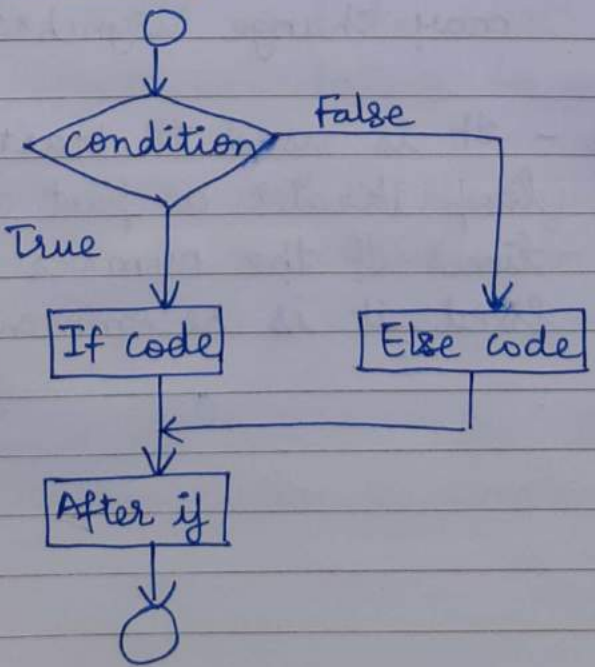
__/__/____

* Java If - Else Statement

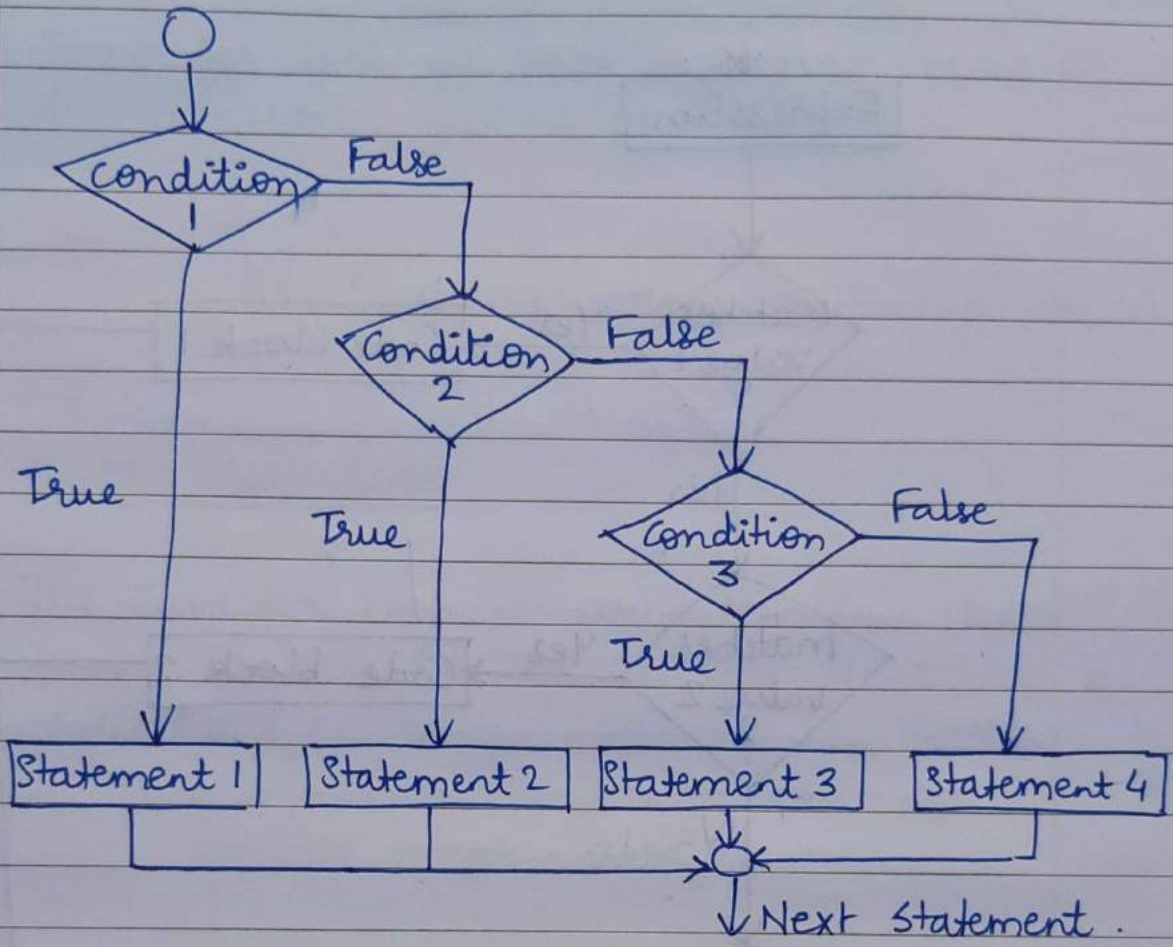
→ if statement



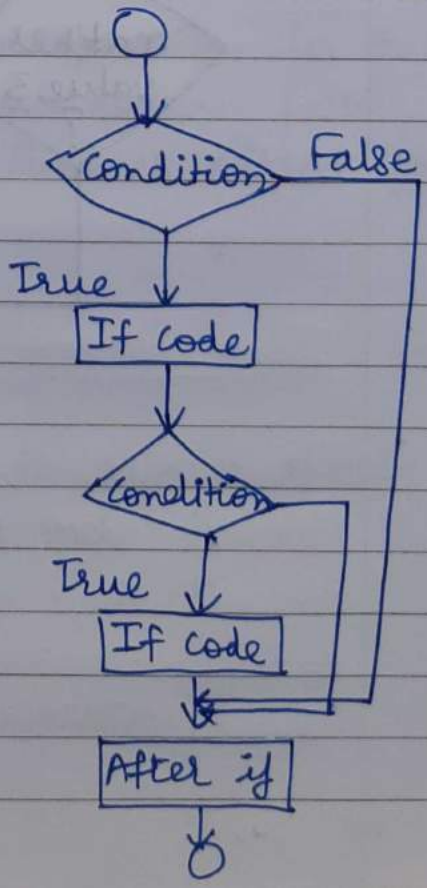
→ if - else statement



→ if - else - if ladder statement

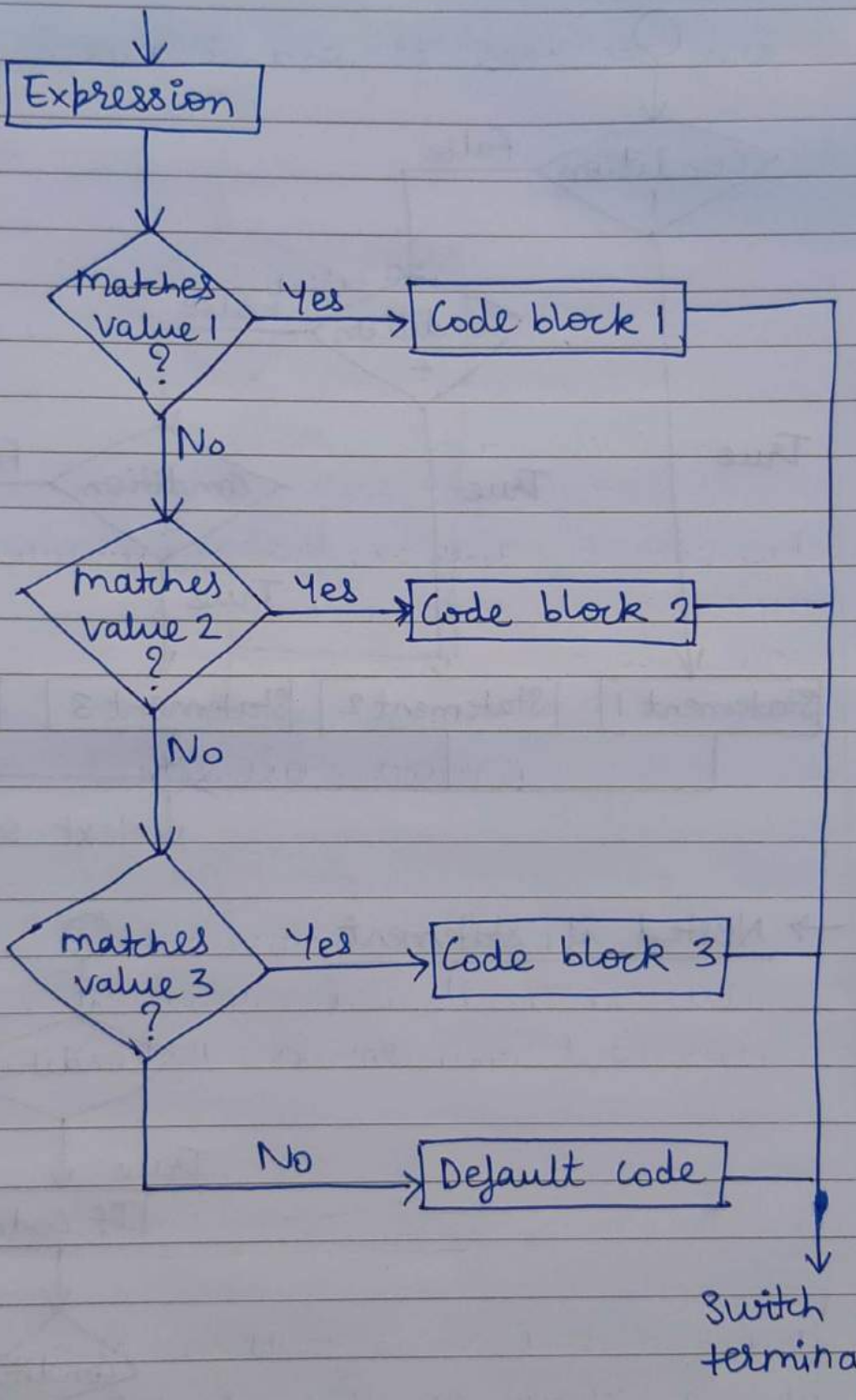


→ Nested if Statement



__/__/____

* Java Switch Statement -



__/__/____

* Java for Loop

Java for loop is used to run a block of code for a certain number of times.

Syntax :-

```
for (initialExpression; conditionExpression; updateExpr)
```

=> For-each loop

The Java for loop has an alternative syntax that makes it easy to iterate through arrays and collections.

```
for (int integer : array) {
}
```

=> If we set the condition expression in such a way that it never evaluates to false, the for loop will run forever. This is called infinite for loop.

* Java while loop

Java while loop is used to run a specific code until a certain condition is met.

Syntax :-

```
while (conditionExpression) {
}
```

Java do... while loop is similar to while loop. However, the body of do.. while loop is executed once before the test expression is checked.

Syntax :-

```
do {
//body of loop
}
while (conditionExpression)
```

The body of the loop is executed at first. Then, the ~~condition~~ conditionExpression is evaluated.

⇒ If the condition of a loop is always true, the loop runs for infinite times (until the memory is full).

NOTE :- The for loop is used when the number of iterations is known.
The while and do.. while loops are generally used when the number of iterations is unknown.

* Java break statement

The break statement in Java terminates the loop immediately, and the control of the program moves to the next statement outside the loop.

__/__/____

How it works?

1>
```
while (testExpression){
    //codes
    if (condition){
        break;
    }
    //codes
}
```

2>
```
while (testExpression){
    //codes
    while (testExpression){
        //codes
        if (condition){
            break;
        }
        //codes
    }
    //codes
}
```

=> Labelld break statement is used to terminate / access the loop labeled as by the user in the code.

3>
```
while (ConditionExpression){
  first:
    while(condition Expression){
      break first ;
    }
}
```

4>
```
first:
  while (ConditionExpression){
    second:
    while (condition Expression {
    break first ;
    }
}
```

* Java continue ~~loop~~ statement

The continue statement skips the current iteration of a loop

1>
```
while (test Expression){
    //codes
    if (testExpression){
      continue ;
    }
}
```

2>
```
for (init ; test Expression ; update){
      //codes
      if (testExpression){
        continue ;
      }
}
```

3> do {
    //codes
    if (testExpression){
        continue;
    }
    //codes
}
while (testExpression);

4> Nested :-

```
while (testExpression){
    //codes
    while (testExpression){
        //codes
        if (testExpression){
            continue;
        }
    }
}
```

5> Labeled statement :-

```
label:
while (testExpression){
    //codes
    while (testExpression){
        //codes
        if (testExpression){
            continue label;
        }
    }
}
```

JAVA CLASS :-

__/__/____

NOTE :-

Java is an object oriented programming language. The core concept of the object oriented approach is to break complex problems into smaller objects.

* Java Class

A class is a blueprint for the object.

Syntax :-

```
class className {
  //fields
  // methods
}
```

* Java Object

Any entity that has state and behaviour is known as object.
It is an instance of a class.
An object contains an address and takes up some space in memory.

Syntax :-

*

```
classname object = new classname();
```

We can use the name of objects along with '.' ~~operation~~ operator to access members of the class.

→ object.variable ; → object.method();
Here, variable is a field and method() is a method.

NOTE *

We can think of the class as sketch (prototype) of a house. It contains all the details about the floors, doors, window, etc. Based on these description we build the house. House is the object.

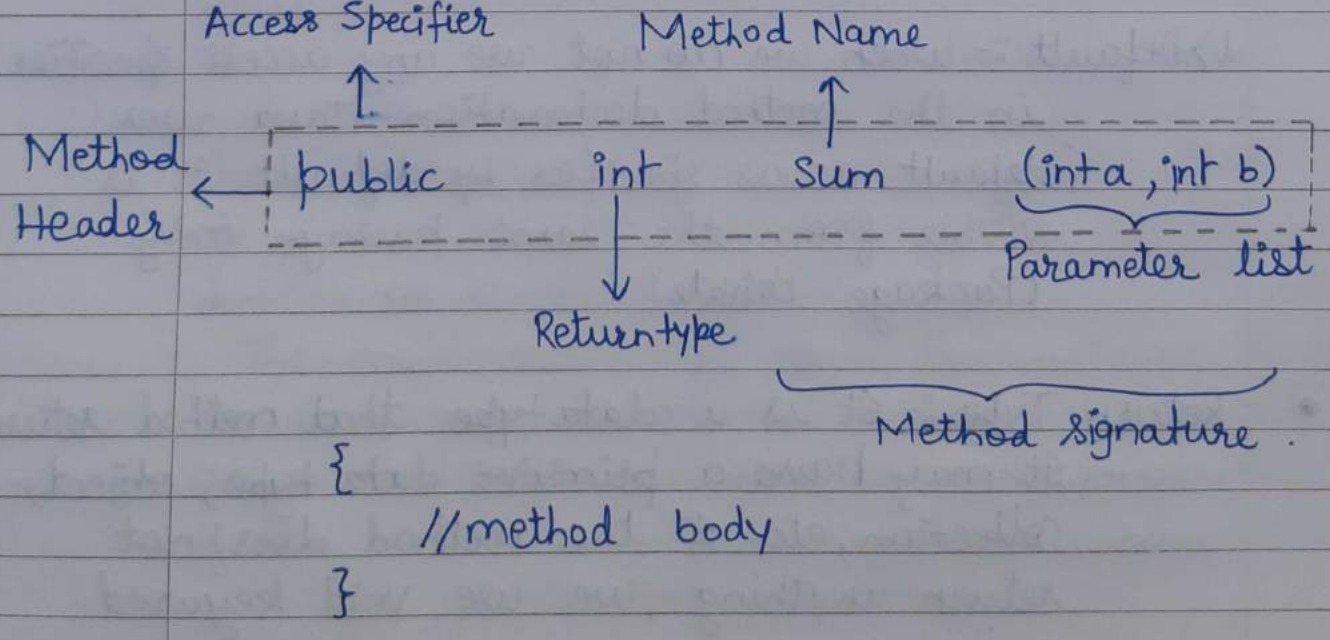
* Java Methods

A method is a block of code that performs a specific task.

In Java, there are two types of methods :-

1> User-defined Methods

Method declaration syntax :-




```
{
    //method body
}
```

__/__/____

- Method Signature :- Every method has a method signature. It is a part of method declaration. It includes the method name and parameter list.

Access Specifier :- It is the access type of the method. It specifies the visibility of the method.

a) Public :- The method is accessible by all classes when we use public specifier in our application. Declarations are visible everywhere.

b) Private :- The method is accessible only in the classes in which it is defined. Declarations are visible within the class only.

c) Protected :- The method is accessible within the same package or subclasses in a different package. Declarations are visible within the package or all subclasses.

d) Default :- When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible from the same package only. (Package Private)

- Return Type :- It is a data type that method returns. It may have a primitive data type, object, collection, etc. If the method does not return anything, we use void keyword.

Access Modifier	Within Class	Within Package	Outside package by subclass only	Outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

- Method Name :- It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of method.

- Parameter list :- It is the list of parameters separated by a comma and enclosed in the pair of ~~para~~ parenthesis. It contains the datatype and variable name.

2) Standard Library Methods :-

These are built-in methods that are readily available for use.
These standard libraries come with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

Eg :- Sqrt() is a method of Math class. It returns the square root of a number.

3) Static Method :-

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method.
The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it.

Eg :- public static void main();

4) Instance Method :-

It is a non-static method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of the class.

Eg :- public int add (int a, int b).

* Java Constructors

A constructor in Java is similar to a method that is invoked when an object of the class is created.

A constructor has the same name as that of the class and does not have any return type.

It is called when an instance of the class i.e. object is created.

A Java constructor cannot be abstract, static, final and synchronized.

Syntax :-

```
class Test {
    Test() {
        // constructor body
    }
}
```

NOTE i) Once a constructor is declared private, it cannot be accessed from outside the class. So, creating objects from outside the class is prohibited using private constructor.

// Default value of an object is 'Reference Null'

ii) A constructor can be overloaded but cannot be overriden.

⇒ Constructor types :-

a) No-Arg Constructors

A constructor that does not accept any arguements.

Eg:-
```
class Company {
    String name;
    public Company(){
        name = "Program";
    }
}
```

b) Parameterized Constructors

A constructor that accepts arguments.

Eg:-
```
class Main {
    String languages;
    Main (String lang){
        languages = lang;
    }
}
```

c) Default Constructor

If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values.

NOTE :- Use equalsIgnoreCase() method to ignore the UpperCase and Lowercase into consideration.

__/__/____

* Java Strings :-

String is a sequence of characters. Strings in Java are not primitive data types. Instead, all strings are objects of a predefined class name String. All string variables are instances of the String class

⇒ String Methods :-

1) length()

We use this method to find the length of the string.

→ String.length().

2) Concat()

To join two strings.

→ first.concat(second)

3) equals()

To compare two strings.

→ first.equals(second)

It returns boolean value.

4) substring()

It returns a substring from the given string.

→ string.substring(int startIndex, int endIndex)

- The substring begins with the character at the startIndex and extends to the character at index endIndex - 1

- If the endIndex is not passed, the substring begins with the character at the specified index and extends to the end of the string.

Eg:- Str1.Substring(3);
Str1.substring(3,7);

5) replace()

This method replaces each matching occurences of the old character in the string with the new character.

→ String.replace(char oldChar, char newChar);
or
→ String.replace(CharSequence oldText, CharSequence new);

- oldChar and newChar are characters.
- oldText and newText are substring.
- If the character to be replaced is not in the string, it returns the original string.

Eg:- Str1.replace('a','z');
"Lava".replace('L','J');
Str1.replace("C++","Java");
"aa bb aa zz".replace("aa","zz");

- In case of replacing substrings, the method replaces only the first occurence. Use the replaceAll() method replace all occurences.

6) Split()

The method divides the string at the specified regex and returns an array of substrings.

__/__/____

→ string.split (String regex, int limit)

- regex - the string is divided at this regex ((an be strings).

- limit - controls the number of resulting substrings.

If the limit parameter is not passed, split() returns all possible substrings.

⇒ String vowels = "a::b::c::d:e";
String[] result = vowels.split ("::");
Arrays.toString (result);

Here,
result = [a, b, c, d:e]

- If the limit parameter is 0 or negative, it returns an array containing all substrings.
- If the limit parameter is positive, lets say 'n', it returns the maximum of n substrings.

⇒ String vowels = "a:bc:de:fg:h";

Now,
vowels.split (":", -2); [a, bc, de, fg, h]
vowels.split (":", 0) ; [a, bc, de, fg, h]
vowels.split (":", 2) ; [a, bc:de:fg:h]
vowels.split (":", 4) ; [a, bc, de, fg:h]

NOTE - If you need to use the split() method on special characters such as: \, !, ^, *, +, etc. you need to escape these characters by '\\'
⇒ Str.split ("\\f");

7> compareTo()

This method compares two strings lexicographically (ASCII Value order)

→ string.compareto(String str)

- returns 0 if the strings are equal
- returns a negative integer if the string comes before the str argument in the dictionary order.
- returns a positive integer if the string comes after the str argument in the dictionary order.

⇒ Str1 = "Learn Java"
Str2 = "Learn Java"
Str3 = "Learn Kolin"

Str1.compareTo(Str2) //0
Str1.compareTo(Str3) //-1
Str3.compareTo(Str1) // 1

⇒ Str1 = "Learn Java"
Str2 = "learn Java"

Str1.compareTo(Str2) //-32.

compareTo() method takes Uppercase and Lowercase into consideration.
Use compareToIgnoreCase() to ignore that case.

__/__/____

8) Contains()

It checks whether the specified string is present in the string or not.

→ String.contains(CharSequence ch)

A charSequence is a sequence of characters such as: String, CharBuffer, StringBuffer.

It returns a boolean value.

9) IndexOf()

It returns the index of the first occurence of the specified character/substring within the string.

→ String.indexOf(char ch, int fromIndex)

or

String.indexOf(String str, int fromIndex)

- ch/str → The character/string whose starting index is to be found.
- fromIndex (optional) → If it is passed, the character/string is searched starting from this index.

- This method returns the index of the first occurence of the specified character/string and returns '-1' if it is not found.
- If the empty string is passed, indexOf() returns 0. (Found at first position. It is because the empty string is a subset of every substring).

__/__/____

10) trim()

It returns a string with any leading (starting) and trailing (ending) whitespace removed.

If there are not whitespace in the start or the end, it returns the original string.

NOTE :-

Whitespace is any character or series of characters that represent horizontal or vertical space.

Eg :- space, newline \n, tab \t, vertical tab \v, etc.

If you need to remove all whitespace characters from a string, you can use replaceAll() method.

→ String.replaceAll("\\S","");

11) charAt()

It returns the character at the specified index.

→ String.charAt(int index)

If the index passed to charAt() is negative or out of bonds, it throws an exception.

12) toLowerCase() - converts all characters in the string to lower case characters.

13) toUpperCase() - converts all characters in the string to upper case characters.

14) concat() - This method concatenates two strings and returns it.

→ String.concat(String str)

You can also use the '+' operator for concat^.

	concat()	'+' operator.
1>	Suppose, str1 is null and str2 is not null. Then, str1.concat(str2) throws NullPointerException.	Suppose, str1 is null and str2 is 'Java'. Then, str1 + str2 gives "nullJava".
2>	You can only pass a String to the method.	If one of the operand is a string and the other one is not a string, then the non-string value is internally converted to a string before concatenation. "Java" + 5 = "Java5".

15> valueOf()

It returns the string representation of the argument passed. It is a static method. So, we call the method using class name like this:-

→ String.valueOf(b);

The method belongs to the String class.

It takes only one parameter.

Eg:- int a = 5; String.valueOf(a) // "5"

long l = -2343834L; String.valueOf(l) //-2343834.

char ch[] = {'J', 'a', 'v', 'a'}.

String.valueOf(ch) // "Java"

16) matches()

It checks whether the string matches the given regular expression or not.

→ string.matches(String regex)

Eg:- String regex = "^a...s$";
//five letter word that starts with 'a' and end 's'.

"abs".matches(regex); // false
"alias".matches(regex); // true
"an abacus".matches(regex); // false.

17) startsWith()

It checks whether the string begins with specified string or not.

→ String.startsWith(String str, int offset)

offset (optional) - checks a in a substring of string starting from this index.

Eg:- String str = "Java Programming";
str.startsWith("Java") //true.
str.startsWith("J") // true
str.startsWith("java") //false.
str.startsWith("Java", 3) //false.
str.startsWith("a Pr", 3) //true.

18) endsWith()

It checks whether the strings end with the specified string or not.

→ String.endsWith(String str).

19) isEmpty()

It checks whether the string is empty or not. It returns true if the string is empty and false if not empty.

→ String.isEmpty()

20) intern().

It returns a canonical representation of the string object.

→ String.intern()

It does not take any parameters.

The String interning ensures that all strings having the same contents use the same ~~no~~ memory.

Since both str1 and str2 have the same contents, both these strings will share the same memory. Java automatically interns the string literals.

```
String str1 = "xyz" new String ("xyz")
String str2 = new String ("xyz")

System.out.println (str1 == str2) //false
  // Because they dont share the same
     memory pool.

//using intern() method.
str1 = str1.intern();
str2 = str2.intern();

System.out.println(str1 == str2) // true.
```

Both string have the same content, but they are not equal initially.
Then we use intern() method so that both strings use same memory pool. Now, they are equal.

21) contentEquals()

It checks whether the contents of the string is equal to the specified charSequence / StringBuffer or not.

→ string.contenEquals(StringBuffer Sb)

→ string.contentEquals(~~Stein~~ CharSequence cs)

22) hashCode()

It returns a hashcode for the string.

A hashcode is a number (object's memory address) generated from any object, not just strings.

This number is used to store / retrieve objects quickly in a hashtable.

It doesn't take any parameters.

→ String.hashCode()

The hashcode is computed using formula :-

$$s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1].$$

- n is the length of string.

Eg:- String str = "Java";
System.out.println(str.hashCode());
// 2301506.

NOTE :- For two strings to be equal, their hash code must be equal too.

23) join()

It returns a new string with the given elements joined with the specified delimiter.

→ String.join (CharSequence delimiter, Iterable elements)

OR

→ String.join (CharSequence delimiter, CharSequence...elements)

Here, ... denotes that there can be one or more CharSequence.

join() is a static method. You do not need to create a string object to call this method. Rather, we call the method using the class name String.

Parameter :-

delimiter - the delimiter to be joined with elements.

elements - elements to be joined.

The function returns a string.

- If an iterable is passed, its elements will be joined. The iterable must implement CharSequence

- String, StringBuffer, CharBuffer, etc. are CharSequence as these classes implement it.

Eg:- CharSequence()

```
result = String.join("-", "Java", "is", "fun");
System.out.println(result);  // Java-is-fun.
```

Eg:- Iterable()

```
ArrayList<String> text = new Arraylist<>();
text.add("Java");
text.add("is");
text.add("fun");

String result = String.join("-", text);
System.out.println(result)        // Java-is-fun.
```

24) replaceFirst()

It replaces the first substring that matches the regex of the string with the specified text.

→ String.replaceFirst(String regex, String replacement)

Refer replace() method given before

25) SubSequence()

It returns a character sequence from the string.

→ String.Subsequence(int StartIndex, int endIndex);

StartIndex is inclusive and endIndex is exclusive.

Eg:- Str = "Harappa"

(print) Str.Subsequence(2,5); //rap.

26) toCharArray()

It converts the string to a Char Array and returns it.

→ String.toCharArray(int start, int end)

Eg:- str = "Java Programming";
char[] result = ;
result = str.toCharArray();
System.out.println(result); // Java Program
-ming

27) format()

It returns a formatted string based on the argument passed.

→ String.format(String format, Object... args)

- format() is a static method. We call this method using the class name String.
- ... denotes you can pass more than one object.

Parameters :-

format - a format string.
args - 0 or more arguments.

Eg: String language = "Java";
int number = 30;

result = String.format("Language : %s", language);
// Language : Java.
result = String.format("Hexadecimal Number : %x", number);
//Hexadecimal Number : 1e

__/__/____

Here, "Language : %s" is a format string.
%s in the format string is replaced with the content of language. %S is a format specifier.
Similarly, %x is replaced with the hexadecimal value of number.

* Format Specifiers

Specifier	Description
%b, %B	"true" or "false" based on the argument.
%s, %S	a string.
%c, %C	a Unicode character.
%d	a decimal integer (used for integers only)
%o	an octal integer (used for integer only)
%x, %X	a hexadecimal integer (used for integer only)
%e, %E	for scientific notation (used for floating-point)
%f	for decimal ~~notation~~ numbers (used for floating-point)

You can use multiple format specifiers in one format string. Eg:- int n1 = 47 ; String text = "Result" ;
(printing) String.format("%s\n Hexadecimal : %x", text, n1) ;

// Result
Hexadecimal : 2f

→ Formatting of Decimal Numbers

```
float n1 = -452.534f
double n2 = -345.766d.

String.format("n1 = %f", n1)  //-452.53397
String.format("n2 = %f", n2)  //-345.766000

String.format("n1 = %.2f", n1)  //-452.53
String.format("n2 = %.2f, n2)  //-345.77.
```

- Java doesn't return the exact representation of floating-point numbers.
- When %.2f is used, it gives two numbers after the decimal point.

→ Padding numbers with Spaces and 0

```
int n1 = 46 , int n2 = -46;
```

		//Output
(i)	result = String.format("|%5d|", n1)	| 46|
(ii)	result = String.format("|%05d|, n1)	|00046|
(iii)	result = String.format("%+d", n1) result = String.format("%+d", n2)	+46 -46
(iv)	result = String.format("%(d", n2);	(46)

Here, (i) and (ii) is padding number with spaces and zeros. The length of the string will be 5 in both.
(iii) is using signs before numbers. (iv) is enclosing negative number within parenthesis and removing the sign.

The Object Class is the superclass for all classes in Java. Hence, every class can implement these methods.

* Object Methods

1) equals()

It checks whether two objects are equal.

→ object.equals(object obj)

It returns boolean value.

Eg:- obj1.equals(obj2)

2) toString()

It converts the object into a string and returns it.

→ object.toString()

Eg:- Object obj1 = new Object();
print obj1.toString() //java.lang.Object@6a6824be

Here,

- java.lang.Object - class name
- @ - at sign
- 6a6824be - hash code of object in hexadecimal form.

3) hashCode()

It returns the hash code value associated with the object.

→ object.hashCode()

Eg:- Object obj1 = new Object();
print obj1.hashCode() //1785210046.

5> getClass()

If returns the class name of the object.

→ object.getClass()

Eg:- Object obj1 = new Object();

printing → obj1.getClass() // java.lang.Object.

String obj2 = new String();

→ obj2.getClass() // java.lang.String.

ArrayList<Integer> obj3 = new ArrayList<>();

→ obj3.getClass() // java.util.ArrayList.

Eg:- Call method from Custom Class.

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

    Main obj = new Main();

    System.out.println(obj.getClass());
  }
}
```

Output :- class Main.

Here, we have created a class named Main. Note that we have called the getClass() method using the method of Main.

JAVA OOP :-

__/__/____

* final keyword :

It is used to denote constants. Once any entity (variable, method or class) is declared final, it can be assigned only once.

- The final variable cannot be reinitiated with another value.
- The final method cannot be overriden.
- The final class cannot be extended/inherited
- It is recommended to use uppercase to declare final variables in Java.

* INHERITANCE

It is one of the key features of OOP that allows us to create a new class from an existing class.

The new class that is created is known as subclass/child/derived and the existing class from where the child class is derived is known as parent/base/super class.

The extends keyword is used to perform inheritance in Java.

```
class Animal {
  // code
}
class Dog extends Animal {
  //
}
```

→ Super keyword in Java Inheritance.

Super keyword is used to call the method of the parent class from the method of the child class.

```
class Animal {
    public void eat(){
        System.out.println("I can eat");
    }
}

class Dog extends Animal{

    //overriding the eat method.
    public void eat(){
        super.eat();
        System.out.println("I eat dog food");
    }
}

class Main{
    public static void Main(String[] args){

        Dog labrodor = new Dog();

        labrodor.eat()

    }
}
```

Output :- I can eat
I eat dog food.

__/__/____

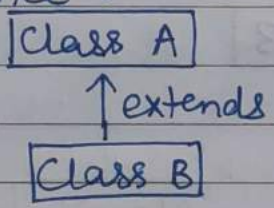
→ Protected Membess in Inheritance

If a class includes protected fields and methods, then these fields and methods are accessible from the subclass of the class.

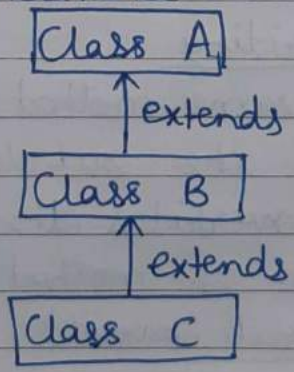
→ The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.

→ Types of Inheritance :-

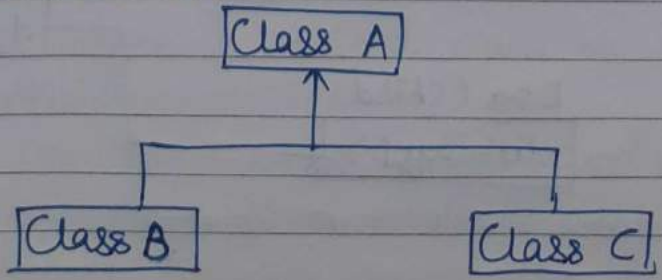
1> Single Inheritance :-



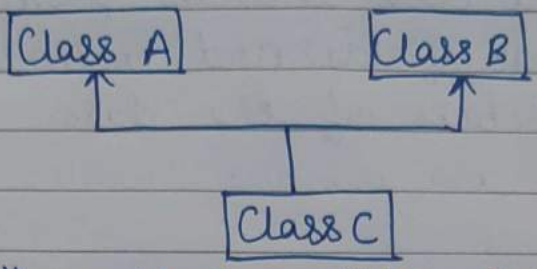
2> Multilevel Inheritance :-



3> Hierarchical Inheritance

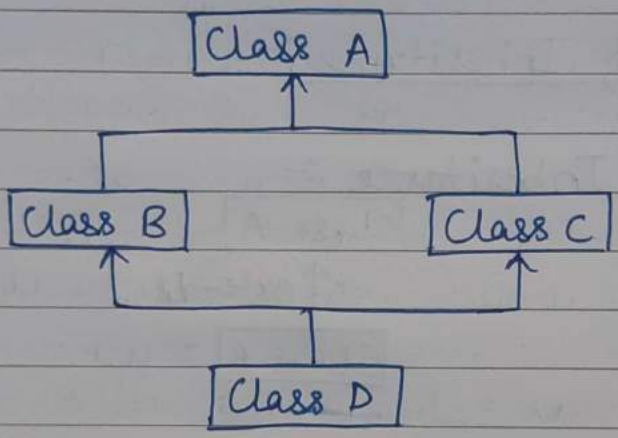


4> Multiple Inheritance



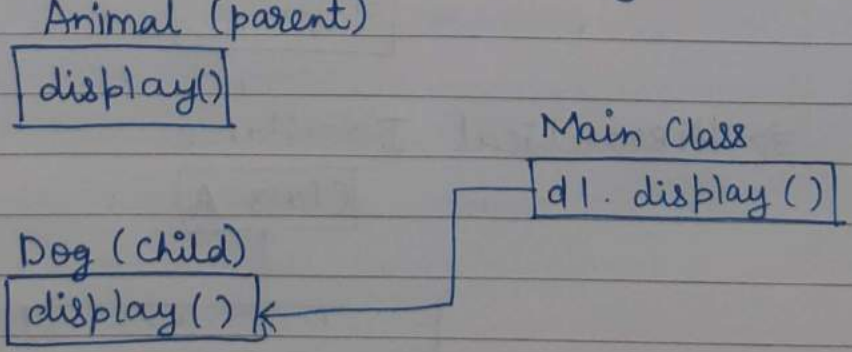
Java doesn't support multiple ~~inheritance~~ superclasses

5> Hybrid Inheritance



* Method Overriding.

If the same method is defined in both the superclass and the subclass, then the method of the subclass overrides the method of the superclass. This is known as method overriding.



Rules for overriding

- Both the superclass and subclass must have the same method name, return type and parameters.
- We cannot override the method declared as final and static.
- We should always override abstract methods of the superclass

* ABSTRACTION :-

It is a process of hiding the implementation details and showing using functionality to the user.

Abstraction lets you focus on what the object does instead of how it does it.

This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

There are two ways to achieve abstraction in Java

a) Abstract class (0 to 100%)

b) Interface (100%)

→ Abstract Class

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated (we cannot create objects of abstract classes)

- An abstract method must be declared with an abstract keyword.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

__/__/____

→ Abstract Method.

A method that doesn't have a body is known as abstract method.

We use the same abstract keyword to create abstract methods.

If a class contains an abstract method, then the class should be declared abstract. Otherwise it will generate an error.

- To implement features of an abstract class, we inherit subclasses from it create objects of the subclass.

- A subclass must override all abstract methods of an ~~abs~~ abstract class. However, if the subclass is declared abstract, its not mandatory to override abstract methods.

- We can access the static ~~fn~~-attributes and methods of an abstract class using the reference of the abstract class. Eg:- Animal.staticMethod();

Eg:-

```
abstract class Animal {
    abstract void makeSound();
}

class Dog extends Animal {
    public void makeSound(){
        System.out.println("Bark Bark");
    }
}
```

```
class Cat extends Animal {
    public void makeSound (){
        System.out.println ("Meows");
    }
}

class Main{
    public Static Void main (String[] args){
        Dog d1 = new Dog() ;
        d1. makeSound;

        Cat c1 = new Cat() ;
        c1 . makeSound ();
    }
}.
```

Output :-

Bark bark
Meows.

Here, the superclass Animal has an abstract method makeSound(). The makesound() method cannot be implemented inside Animal. So, the implementation of makeSound() is kept hidden.

So, Dog makes its own implementation of MakeSound() and Cat makes its own makeSound().

__/__/____

* INTERFACES.

i) An interface is a fully abstract class that includes a group of methods without a body.
ii) In Java, an interface defines a set of specifications that other classes must implement.

→
```
interface Language {
        public void getName();
}
```

Now, every class that uses this interface should implement the getname() specification.

iii) An interface in Java is a blueprint of a class. It has static constants and abstract methods.
iv) It is a mechanism to achieve abstraction.
v) There can be only abstract methods in the Java interface, but no body of the method because implementation is done in the class.
vi) We can have default, static and private methods in an Interface.
vii) By interface, we can support the functionality of multiple inheritance.
viii) It can be used to achieve loose coupling.
ix) Interface keyword is used to declare an interface.
x) It provides total abstraction; means all the methods in the inferface are declared with the empty body, and all the fields are public, static and final by default.

__/__/____

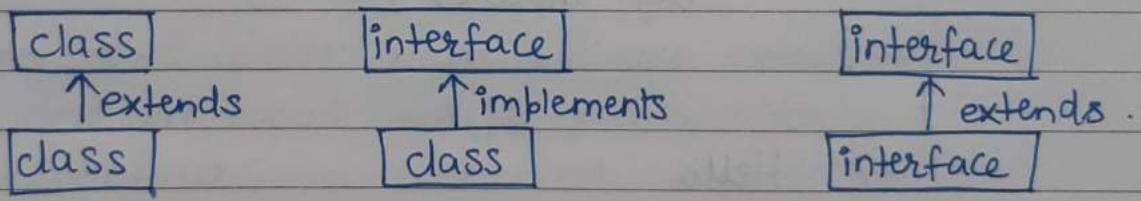
→ Extending an Interface OR Interface Inheritance

i> Similar to classes, interfaces can extend other interfaces by using extends keyword.

```
→ interface Line{
        //body
  }
  interface Polygon extends Line{
        // members of Polygon interface
        //members of Line interface
  }
```

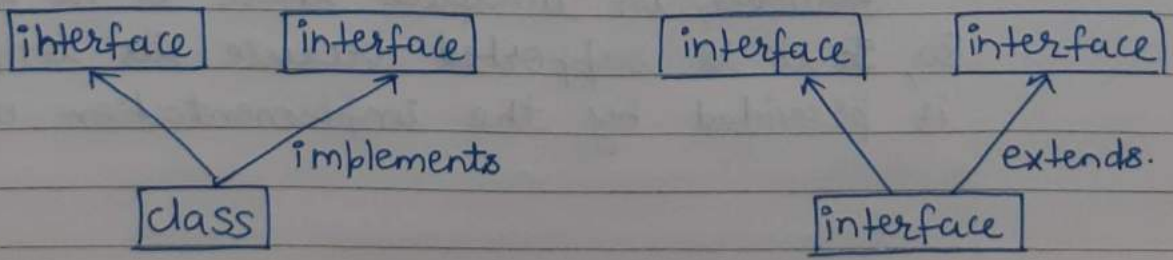
We can extend multiple interfaces from an interface

→ Relationship between classes and interfaces.



→ Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple ~~interfa~~ inheritance.



```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class A implements Printable, Showable {
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
        A obj = new A();
        obj.print();
        obj.show();
    }
}
```

⇒ Hello
Welcome.

NOTE:- Multiple inheritance is not supported through class in Java, but it is possible by an interface, why?

ANSWER:- Multiple inheritance is not supported in the case of class because of ambiguity. However, in interface there is no ambiguity. So, it is supported because its implementation is provided by the implementation class.

__/__/____

* Differentiate between abstract class and interface

	Abstract Class	Interface.
1)	It can have abstract and non-abstract methods.	It can have only abstract methods.
2)	It doesn't support multiple inheritance	It supports multiple inheritance.
3)	It can have static, final, non-final and non-static variables.	It has only static and final variables.
4)	It can provide the implementation of interface.	It can't provide implementation of abstract class.
5)	It can extend another Java class and implement multiple Java interfaces.	It can extend other interfaces only.
6)	It can have class members like private, protected, etc.	Members of interface are public by default.

__/__/____

* POLYMORPHISM :-

It is a concept by which we can perform a single action in different ways. It simply means more than one form.

Types of polymorphism :-

1) Compile time polymorphism :-

This type of polymorphism is implemented using the following :-

a) Method Overloading :-

If a class has multiple methods having same name but different parameters, it is known as Method Overloading.

It increases the readability of the program. There are two ways to overload the method in Java :-

→ By changing the number of arguments.
→ By changing the data type.

Eg:-
```
void func (){....}
void func (int a){....}
float func (double a){....}
float func (int a, float b){....}
```

- Method overloading is not possible by changing the return type of the method because of ambiguity.

__/__/____

- We can also overload main() method. You can have any number of main methods in a class by method overloading. But JVM call main() method which receives string array as arguments only.

b> Operator Overloading

Some operators in Java behave differently with different operands. Like,

→ + operator is overloaded to perform numeric addition as well as string concatenation.

→ operators like &, | and ! are overloaded for logical and bitwise operations.

2> Run time Polymorphism

Runtime Polymorphism or Dynamic Method Dispatch is a process in which a call to an overriden method is resolved at runtime.

Method Overriding is an implementation of Run time Polymorphism.

__/__/____

* ENCAPSULATION

It is a process of wrapping code and data together into a single unit.

It refers to the bundling of fields and materials (methods) inside a single class.

It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve data hiding.

We can create a fully encapsulated class in Java by making all the data members of the class private.

Encapsulation helps us to keep related fields and methods together, which makes our code cleaner and easy to read.

NOTE:- Encapsulation in itself is not data hiding. It is a way to provide data hiding.

→ By providing a getter or setter method, you can make the class read-only or write only.

Eg:- Encapsulation -

```
public class Student {
        private String name;
        public String getName(){
//getter method             return name;
        }
//setter method         public void setName(String name){
                            this.name=name;
                        }
}
```

```
class Test {
    public static void main (String[] args){
        Student s = new Student();
//setting value    s.setName ("Vijay");
//getting value    System.out.println (s.getName());
    }
}
```

Eg:- Read Only class

```
public class Student{
    private String college = "AKG";
    public String getCollege(){
        return college;
    }
}
```

Now, you can't change the value of the college data member which is "AKG".

```
s.setCollege ("KITE"); //will render compile time error.
```

Eg:- Write Only Class.

```
public class Student{
    private String college;
//getter method    public void setCollege (String college){
        this.college = college;
    }
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

__/__/____

* Java Enums

Enum (short for enumeration) is a type ~~of~~ that has a fixed set of constant values. We use the enum ~~keyboard~~ keyword to declare enums.

```
enum Size {
    A, B, C, D
}
```

A, B, C, D are fixed values. They are also called enum constants.

Eg:-
```
enum Size {
    SMALL, MEDIUM, LARGE, EXTRALARGE.
}
class Main {
    public static void main (String[] args){
        System.out.println (Size.SMALL) ;
        System.out.println (Size.MEDIUM) ;
    }
}
```
=> SMALL
MEDIUM.

→ Enum class in Java

Enum types are considered to be a special type of class.

An enum class can include methods and fields.

When we create an enum class, the compiler will create instances (objects) of each enum constants.

__/__/____

Also, all enum constant is always public static final by default.

Eg:- Enum class.

```
enum Size {
    SMALL, MEDIUM, LARGE, EXTRALARGE;

    public String getSize(){

        Switch (this){
        case SMALL :-
            return "small";
        case MEDIUM:
            return "medium";
        case LARGE:
            return "large";
        case EXTRALARGE:
            return "extra large";
        default :
            return null;
        }
    public static void main (String [] args){
        System.out.println (Size.SMALL.getSize());
```

=> small.

→ Methods of Enum class

1) Enum ordinal()
It returns the position of an enum constant.
→ ordinal(SMALL) // returns 0.

2) Enum compareTo()
It compares the enum constants based on their ordinal value.
→ Size.SMALL.compareTo(Size.MEDIUM)
// ordinal(SMALL) - ordinal(MEDIUM)

3) Enum toString()
It returns the string representation of the enum constants.
→ SMALL.toString()
// returns "SMALL"

4) Enum name()
It returns the defined name of an enum constant in string form. The returned value from the name() method is final.
→ name(SMALL)
// returns "SMALL".

5) Enum valueOf()
It takes a string and returns an ~~en~~enum constant having the same string name.
→ Size.valueOf("SMALL")
//returns constant SMALL.

__/__/____

6> Enum values()

It returns an array of enum types containing all the enum constants.

→ Size [] enumArray = Size.value();

* Java Reflection

In Java, reflection allows us to inspect and manipulate classes, interfaces, constructors, methods and fields at run time.

There is a class in Java named Class that keeps all the information about objects and classes at runtime. The object of Class can be used to perform reflection.

In order to reflect a Java class, we first need to create an object of Class. This can be done in 3 ways.

a> forName()

It takes the name of the class to be reflected as its argument.

→ class Dog { ... }

Class a = Class.forName("Dog");

b> getClass()

Dog d1 = new Dog(); //Create an object of Dog

Class b = d1.getClass(); //Create an object of Class by reflecting Dog.

c> Using .class extension.

Class c = Dog.class;

* Reflection of Methods :-

→ Notice the statement,

Class obj = d1.getClass();

Here, we are creating an object obj of class using the getClass() method.

a> obj.getName() - returns the name of class

b> obj.getModifiers() - returns the access modifier of the class.

c> obj.getSuperclass() - returns the super class of the class.

a> obj.getDeclareMethod();

It returns all the methods present inside the class.

Also, we have created an object m of the Method class.

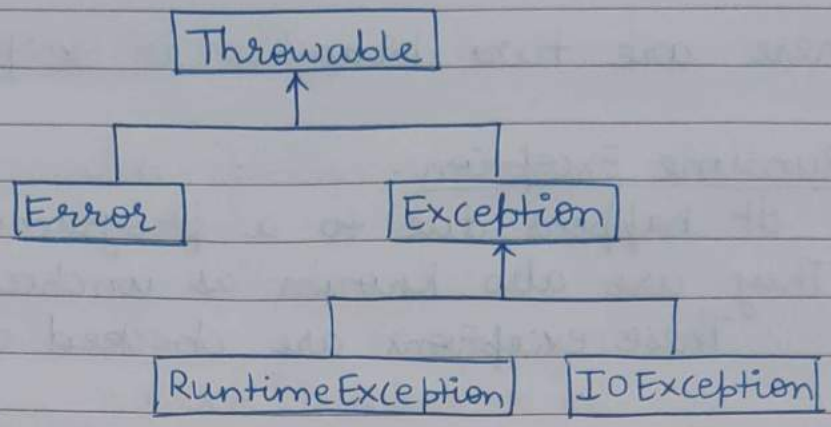
→ m.getName() - returns the name of a method.

→ m.getModifiers() - returns the access modifier of methods in integer form.

→ m.getReturnType() - returns the type return type of methods.

JAVA EXCEPTION HANDLING

__/__/____



The Throwable class is the root class in the ~~hiera~~ hierarchy

→ Errors

Error represent irrecoverable conditions such as Java Virtual Machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

They are usally beyond the control of the programmer and we should try not to handle them.

→ Exceptions

An exception is an unexpected error that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

When an exception occurs within a method, it creates a method. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

There are two branches in exception hierarchy

1) Runtime Exception.

It happens due to a programming error. They are also known as unchecked exceptions. These exceptions are checked at runtime.

- Improper use of an API → IllegalArgumentException
- Null pointer access (missing the initialization of a variable) → NullPointerException.
- Out-of-bounds array access → ArrayIndexOutOfBounds
- Dividing a number by 0 → ArithmeticException.

→ You can think about it in this way - "If it is a runtime exception, it is your fault."

2) IOException

It is also known as a checked exception. They are checked by the compiler at the compile time and the programmer is prompted to handle these exceptions.

- Trying to open a file that doesn't exists → FileNotFoundException.
- Trying to read past the end of a file.

__/__/____

* Exception Handling

1) try... catch block.

→
```
try {
    //code
}
catch (Exception e) {
    //code
}
```

Here, the code might generate an exception is placed in the try block. Every try block is followed by a catch block.
Every try block is followed by a catch block. When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

Eg:-
```
class Main {
    public static void main (String [] args) {
     try {
        int divideByZero = 5/0;
     System.out.println ("Try block");
     }

     catch (ArithmeticException e){
     System.out.println("Arithmetic Exception => " +
                        e.getMessage());
     }
  }
}
```

Output :-

Arithmetic Exception => / by zero.

To handle the exception, we have put the code 5/0 inside try block. Now when an exception occurs, the rest of the code inside the try block is skipped.
The catch block catches the exception and statements inside the catch block are executed.
If none of the statements in the try block generate an exception, the catch block is skipped.

2> finally block.

The finally block is always executed no matter there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

```
→ try{
      // code
  }
  catch (Exception e){
    // catch block
  }
  finally{
     // finally block always executes
  }
```

If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block.

Eg:-
```
class Main{
    public static void main (String[] args){
        try{
            int divideByZero = 5/0;
        }
        catch (ArithmeticException e){
            System.out.println("Catch block");
        }
        finally{
            System.out.println("This is finally block");
        }
    }
}
```

=> Catch block
This is finally block.

The exception is caught by catch block then the finally block is executed.

It is ~~becau~~ a good practice to use the finally block. It is because it can include important cleanup code like,

→ code that might be accidentally skipped by return, continue or break.

→ ~~or~~ closing a file or connection.

3) throw and throws keyword.

The throw keyword is used to explicitly throw a single exception.

When we throw an exception, the flow of the program moves from the try block to the catch block.

It is mainly used to throw custom exception.

__/__/____

Eg:-

```
public class Test {
    static void validate (int age) {
        if (age<18)
            throw new ArithmeticException ("Invalid")
        else
            System.out.println ("Welcome");
    }
    public static void main (String[] args){
        validate (13);
        System.out.println ("Rest of the code");
    }
}
```

=> Exception in thread main java.lang.ArithmeticException : not valid.

→ Throws keyword

It is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that the normal flow can be maintained.

```
→ return-type method () throws exception.name{
        //method code
  }.
```

Eg:-

```
import java.io.IOException;
class Testthrows1 {
    void m() throws IOException {
```

```
    throw new IOException("device error");
}
void n()throws IOException{
    m();
}
void p(){
  try{
    n();
  }catch(Exception e){System.out.println("handled");}
}
public static void main(String[] args){
   Testthrows1 obj = new Testthrows1();
   obj.p();
   System.out.println("normal flow");
  }
}
```

=> exception handled
normal flow.

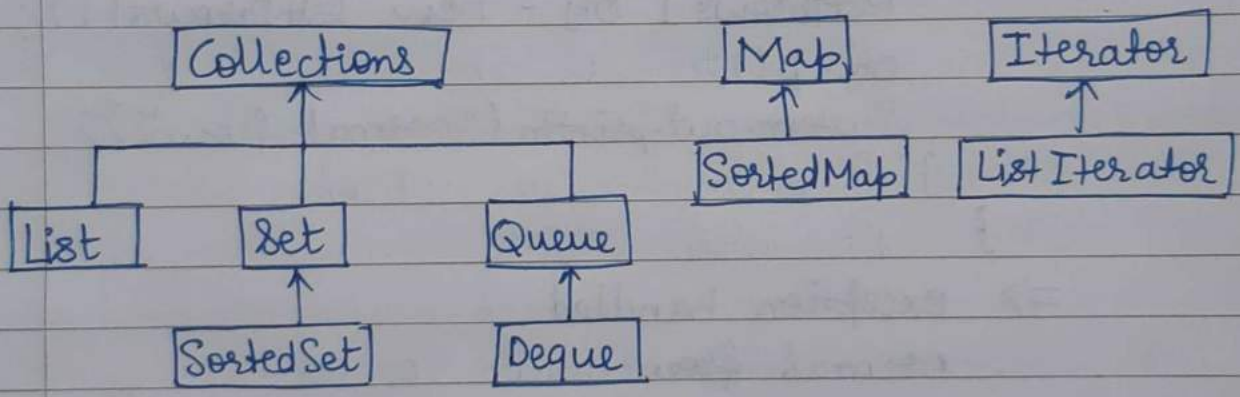
JAVA COLLECTIONS

The collections in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation and deletion.

Java Collection means a single unit of objects.

Java Collection Framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



* Java Collection Interface

The Collection Interface is the root interface of the collections framework.

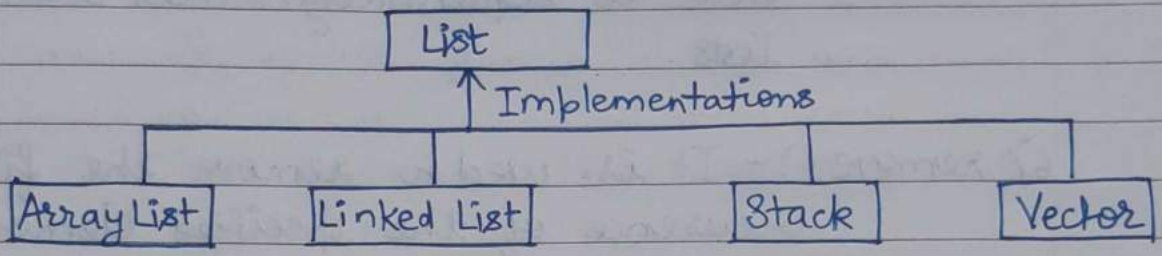
The Collection interface includes subinterfaces that are implemented by Java classes. All the methods of the Collection interface are also present in its subinterfaces.

__/__/____

1) List Interface

The List interface is an ordered collection that allows us to add and remove elements like an array.

It extends the Collections Interface.



In Java, we must use java.util.List package in order to use List.

```
//ArrayList Implementation of List.
    List <String> list1 = new ArrayList <>();

// LinkedList Implementation of List.
    List <String> list2 = new LinkedList <>();
```

- Methods used in List :-

1) add → It is used to append the element in the list.

2) addAll() → Adds all elements of one list to other
   boolean addAll (Collection c, T... elements)
   Eg:- Collections.addAll (arrlist, "1", "2", "3");

3) get() → helps to randomly access elements from lists.

4) Set() → It is used to replace the specified element in the list, present at the specified location.

5) iterator() - returns iterator object that can be used to sequentially access elements of lists.

6) remove() - It is used to remove the first occurence of the specified element.

7) removeAll() - It is used to remove all the elements from the list.

8) clear() - Removes all the elements from the list and is more efficient than removeAll.

9) Size() - Returns the length of lists.

10) toArray() - Converts a list into an array.

11) Contains() - It returns true if the list contains the specified element.

12) containsAll() - It returns true if the list contains all the specified element.

13) indexOf() - Searches a specified element in an arraylist and returns the index of the element.

Eg:- (ArrayList Class)

```
class Main {
    public static void Main (String[] args) {
        List <Integer> numbers = new Arraylist<>();
        numbers.add (1);
        numbers.add (2); numbers.add (3);
        System.out.println ("List: " + numbers);

        int number = numbers.get(2);
        System.out.println (number);

        int removednumber = numbers.remove (1);
        System.out.println (removednumber);
```

=> List : [1, 2, 3]
3
2

→ We can convert an ArrayList to an array or String by the toArray() or toString() respectively. Similarly, we can convert an Array to ArrayList by the asList() method.

→ Vectors.

The Vector class synchronizes each individual operation. This means that whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModification Exception

Type indicates the type of List.
Eg. Integer, String

is generated. Hence, this continuous use of block for each operation makes vectors less efficient.

However, in array lists, methods are not synchronized. Instead, it uses the Collections.synchronizedList() method that synchronizes the list as a whole. It is recommended to use ArrayList in place of Vector because vectors are not threadsafe and are less efficient.

```
Vector <Type> vector = new Vector<>();
```

→ **Stack**

It provides the functionality of the stack data structure.

The Stack class extends the Vector class. Elements are stored and accessed in Last in First Out manner.

```
Stack <Type> stacks = new Stack<>();
```

Stack methods -

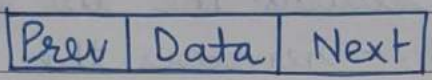
1) push()
2) pop()
3) peek()
4) search() - returns the position of the element from the top of the stack.
5) empty() - To check whether the stack is empty or not. [Note that it is not "isEmpty"]

__/__/____

It is not recommended to use the Stack class ~~for~~. Instead, use the ArrayDeque class (implements the Deque interface) to implement the stack data structure in Java.

→ Linked List

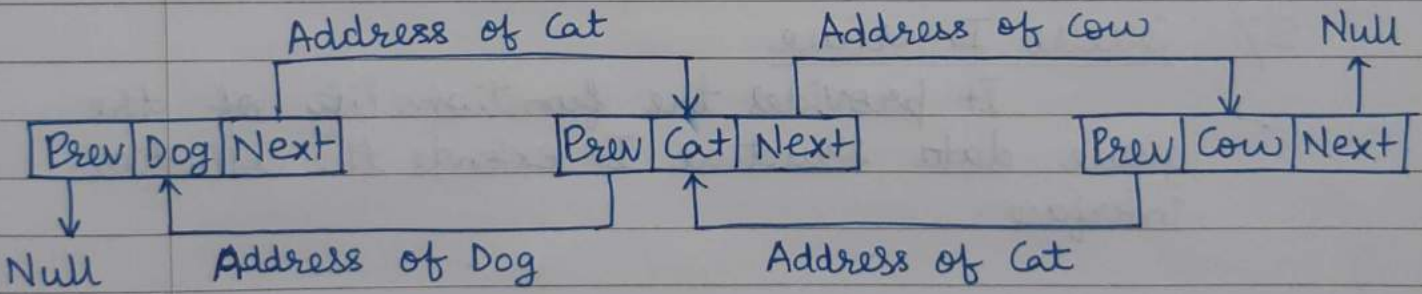
It provides the functionality of the linked list data structure.



Node Structure.

```
LinkedList<Type> linkedList = new LinkedList<>();
```

Working of a Java LinkedList :-



Methods used :-

1) add() - to add an element or node.
2) get() - to access an element.
3) set() - to change elements.
4) remove() - to remove an element
5) contains() - checks if it contains an element.
6) indexOf() - returns the index of the first occurrence.
7) lastIndexOf() - returns the index of the last occurrence.

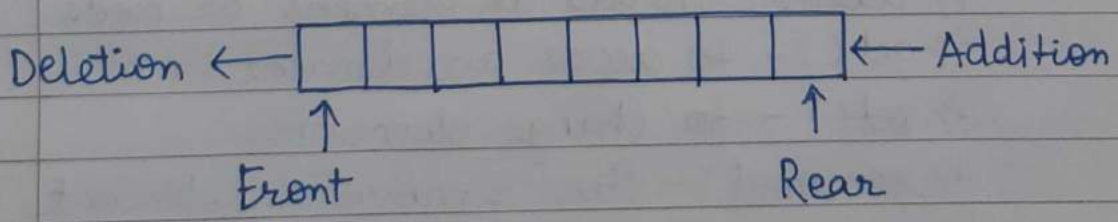
Since, the Linked List class also implements the Queue and the Deque interface, it can implement methods of these interfaces as well.

1) addFirst() - adds the element at the beginning of the linked list.
2) addLast() - adds the element at the end of the list.
3) getFirst() - returns the first element.
4) getLast() - returns the last element.
5) removeFirst() - removes the first element.
5) removeLast() - removes the last element.
6) poll() - returns and removes the first elements from the linked list.
7) offer() - adds the specified element at the end of the linked list.

2) Queue Interface

It provides the functionality of the queue data structure. It extends the Collection interface.

Working of Queue Data Structure :-

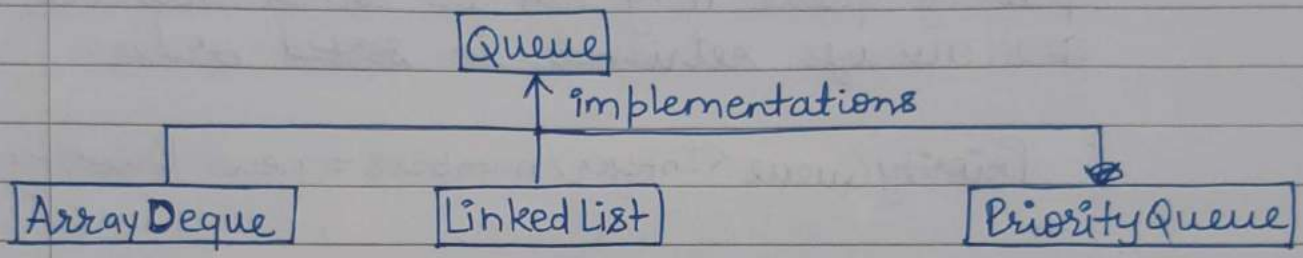


Queue

~~Queue<Type> queue =~~

Since, the Queue is an interface, we cannot provide the direct implementation of it. In order to use the functionalities of Queue, we need to use classes that implement it :-

→ ArrayDeque
→ LinkedList.
→ PriorityQueue.




```
Queue <Type> queue = new LinkedList <>();
Queue <Type> queue = new ArrayDeque <>();
Queue <Type> queue = new PriorityQueue <>();
```

Methods of Queue

1) add() -
2) offer() -
3) element() - returns the head of the queue. Throws an exception if the queue is empty.
4) peek() - returns the head of the queue. Returns null if the queue is empty.
5) remove() - returns the and removes the head of the queue. Throws an exception if the queue is empty.
6) poll() - returns and removes the head of the queue. Returns null if the queue is empty.

→ Priority Queue.

It provides the functionality of heap data structure.

Unlike normal queues, ~~prori~~ priority queue elements are always retrieved in sorted order. It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

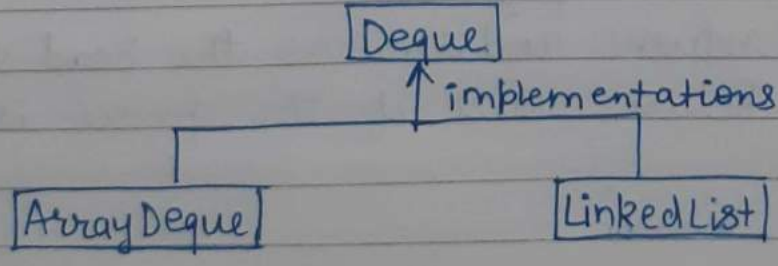
```
PriorityQueue<Integer> numbers = new PriorityQueue<>();
```

Methods of Priority Queue :-

1) add() - Adds element (throws exception if queue is full)
2) offer() - Adds element (returns false if queue is full)
3) peek() - returns head of the queue.
4) remove() - removes specified element.
5) poll() - returns and removes the head of queue.
6) Contains(element) - returns true if element is found. Else, false.
7) Size() - returns the length of the priority queue.
8) toArray() - Converts the priority queue to an array.

→ Deque Interface

It provides the functionalities of a double ended queue. It extends the queue interface.

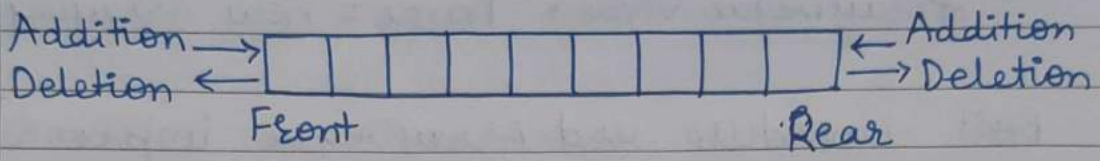


__/__/____

```
Deque <String> queue1 = new ArrayDeque <>();

Deque <String> queue2 = new LinkedList <>();
```

Working of a Deque :-



Methods of Deque :-

1> addFirst()	7> peckFirst()
2> add Last()	8> peekLast()
3> offer First()	9> removeFirst()
4> offerLast()	10> removeLast()
5> getFirst()	11> pollFirst()
6> getLast()	12> pollLast()

=> Deque as Stack Data Structure.

It is ~~recommeded~~ recommended to use Deque as a Stack instead of the Stack class. It is because the methods of Stack class are synchronized.

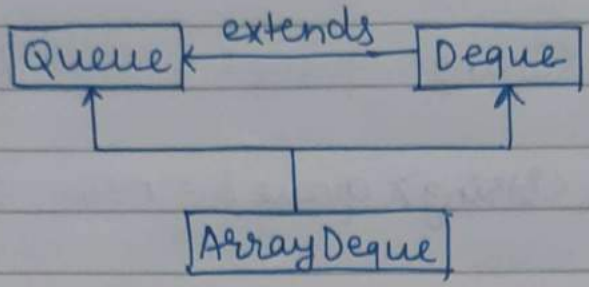
Methods provided by Deque :

a> push() b> pop() c> peek()

→ ArrayDeque

It is used to implement queue and deque data structures using arrays. The ArrayDeque class implements two interfaces :-

1> Queue interface
2> Deque interface.



ArrayDeque <Type> name = new ArrayDeque<>();

Both LinkedList and ArrayDeque implements the Deque interface. However, there exists some differences in them.

→ LinkedList supports null elements, whereas ArrayDeque doesn't.

→ Each node in a linked list includes links to other nodes. That's why LinkedList requires more storage than ArrayDeque.

→ If you are implementing the queue or the deque data structure, an ArrayDeque is likely to be faster than a LinkedList.

* Java Map Interface

It provides the functionality of the map data structure.

Working of Map :-

Elements of Map are stored in key/value pairs.
Keys are unique values associated with individual values.
A map cannot duplicate keys. And, each key is associated with a single value.

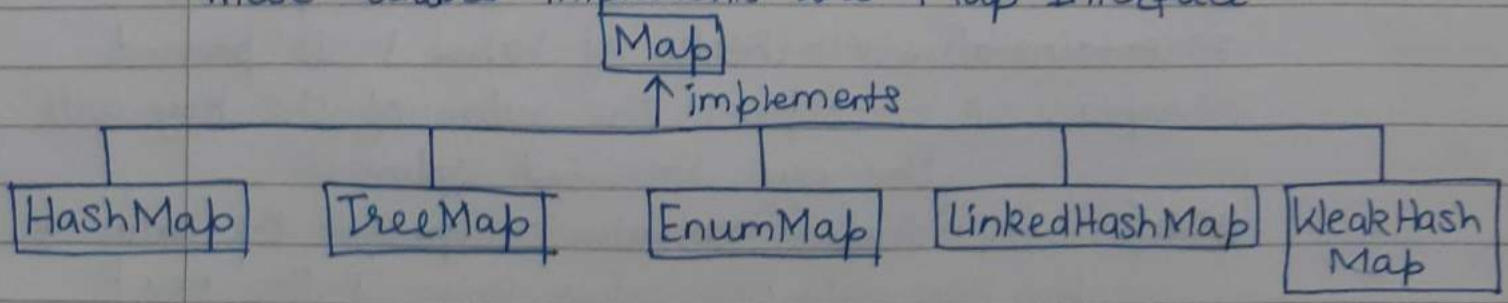
Keys		Values
US	⟶	United States
br	⟶	Brazil
es	⟶	Spain

The Map interface maintains 3 different sets :-

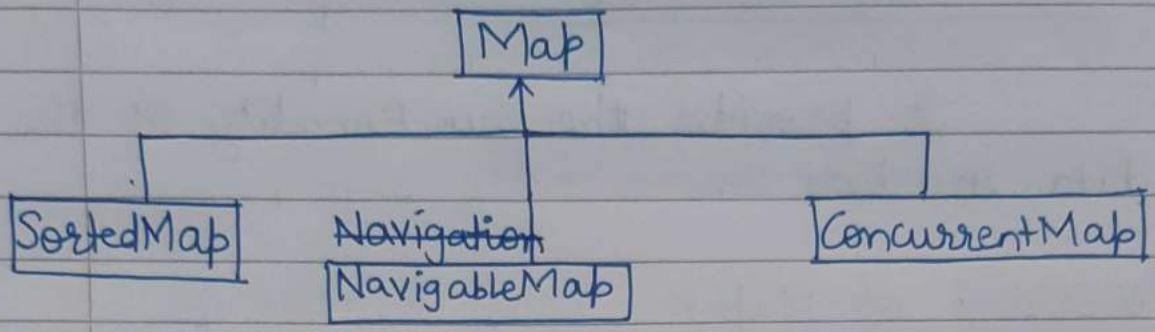
1) The set of keys.
2) The set of values.
3) The set of mappings (key/value associations)

Hence, we can access them individually.

These classes implements the Map Interface.



The Map Interface is also extended by these :-



→ Map<Key, Value> numbers = new HashMap<>();

Methods of Map Interface :-

1) put (K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.

2) putAll - Inserts all the entries from the specified map to this map.

3) putIfAbsent (K, V) - Inserts the association if the key K is not already associated with the value V.

4) get (K) - Returns the value associated with the specified key K. Returns null if not found.

5) ~~get~~ getOrDefault (K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns default value.

6) containsKey (K) - Checks if key K is present.

7) containsValue (V) - Checks if value V is present.

8) replace (K, V) - Replace the value of the key with the new specified value V.

9) replace (K, oldValue, newValue) - Replaces the value of the key with newValue only if the key K is associated with the value oldValue.

10) remove(K) - Removes the entry represented by the K
11) remove(K,V) - Removes the entry that has K associated with value V.
12) keySet() - Returns a set of keys present in the map.
13) values() - Returns a set of all values present in the map
14) entrySet() - Returns a set of all the key/value mapping present in the map.

Eg:-

```
class Main{
    public static void main (String[] args){
      Map<String, Integer> numbers = new HashMap<>();

      numbers.put("One", 1);
      numbers.put("Two", 2);
      System.out.println("Map: " + numbers);

      System.out.println("Keys: " + numbers.keySet());
      System.out.println("Values: " + numbers.values());
      System.out.println("Entries: " + numbers.entrySet());

      int value = numbers.remove("Two")
      System.out.println("Removed: " + value);
    }
}
```

=> Map: {One=1, Two=2}
Keys: [One, Two]
Values: [1, 2]
Entries: [One=1, Two=2]
Removed: 2

1> HashMap

It provides the functionality of the heap table data structure.

```
HashMap <K, V> numbers = new HashMap<>();
```

2> LinkedHashMap

It provides the hash table and linked list implementation of Map Interface.

It extends the HashMap class to store its entries in a hash table. It internally maintains a doubly-linked list among all of its entries ~~to~~ order its entries.

```
LinkedHashMap <Key, Value> numbers = new LinkedHashMap
                                     <>(8, 0.6f);
```

Here, 8 is capacity. Capacity is the number of entries it can store.

0.6 is load Factor. It means whenever our hashmap is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

By default, capacity is 16 and loadFactor is 0.75.

- It maintains a doubly linked list internally. Due to this, it maintains the insertion order of its elements.
- It requires more storage than HashMap. It is because it maintains linked lists internally.
- The performance of LinkedHashMap is slower than HashMap.

__/__/____

3) EnumMap

It provides a map implementation for elements of an enum.

Here, enum maps are used as keys.

```
enum Size {
        SMALL, MEDIUM , LARGE, EXTRALARGE
}
```

```
EnumMap<Size, Integer > Sizes = new EnumMap< >
                                (Size.class);
```

Here, Size - keys of the enum that map to values.
Integer - values of the enum map.

3) SortedMap

It provides sorting of keys stored in a map.

Since, it is an interface, we cannot create objects from it. In order to use its functionalities, we need to use the TreeMap class that implements it.

```
SortedMap<Key, Value> numbers = new TreeMap <>();
```

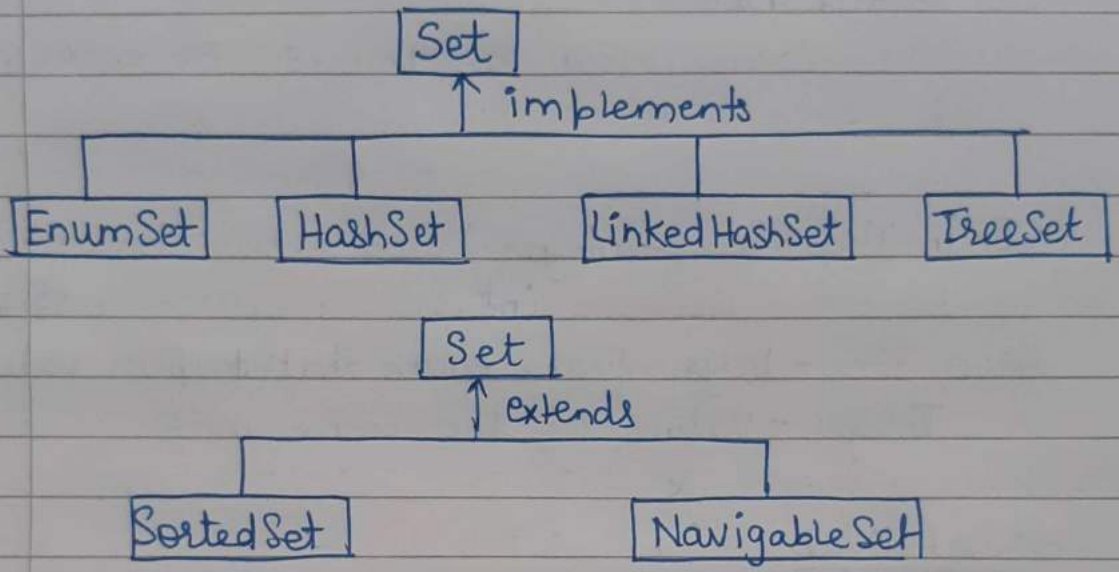
No matter how you insert the keys in the SortedMap, when you access the elements, the keys will be in sorted manner.

__/__/____

* Java Set Interface.

The Set Interface provides the features of the mathematical set in Java.

Unlike the List Interface, sets cannot contain duplicate elements.




```
Set<String> animals = new HashSet<>();
```

Methods to Set :-

1) add()
2) addAll()
3) iterator()- returns an iterator that can be used to access elements of the set sequentially.
4) remove()
5) removeAll()
6) retainAll()- retains all the elements in the set that are also present in another specified set.
7) clear()- removes all the elements
8) size()- returns the length of the set.
9) toArray()

10) contains() - returns true if the set contains the specified element.

11) containsAll().

12) hashCode() - returns a hash code value

Set Operations :-

Union - to get the union of two sets x and y, we can use x.addAll(y)

Intersection - to get the intersection of two sets x and y, we can use x.retainAll(y).

Subset - to check if x is a subset of y, we can use y.containsAll(x).

1) HashSet

It provides the functionalities of the hash table data structure.

```
HashSet<Integer> numbers = new HashSet<>(8,0.75);
```

We have created a hash set named numbers. First parameter is capacity. Second parameter is loadFactor.

HashSet is commonly used if we have to randomly access elements. It is because elements in a hash table are accessed using hash codes.

The hashcode of an element is a unique identity that helps to identify the element in a hash table.

HashSet cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

HashSet is not synchronized. That is if multiple threads access the hash set at the same time and one of the threads modifies the hash set. Then it must be externally synchronized.

2) EnumSet

It provides a set implementation of elements in a single enum.

Creating EnumSet :-

```
//enum
enum Size { SMALL, MEDIUM, LARGE, EXTRALARGE }
```

1) using allOf (Size)

```
EnumSet<Size> sizes = EnumSet.allOf (Size.class);
```

2) using noneOf (Size)

```
EnumSet<Size> sizes = EnumSet.noneOf (Size.class);
```

3) Using range (e1, e2)

```
EnumSet<Size> sizes = EnumSet.range (Size.MEDIUM,
                                     Size.EXTRALARGE);
```

4) Using of()

```
EnumSet<Size> sizes = EnumSet.of (Size.SMALL,
                                  Size.LARGE);
```

Methods of EnumSet :-

1) add()
2) addAll()
3) remove()
4) removeAll()
5) copyOf() - creates a copy of EnumSet.
6) contains()
7) isEmpty()
8) size()
9) clear()

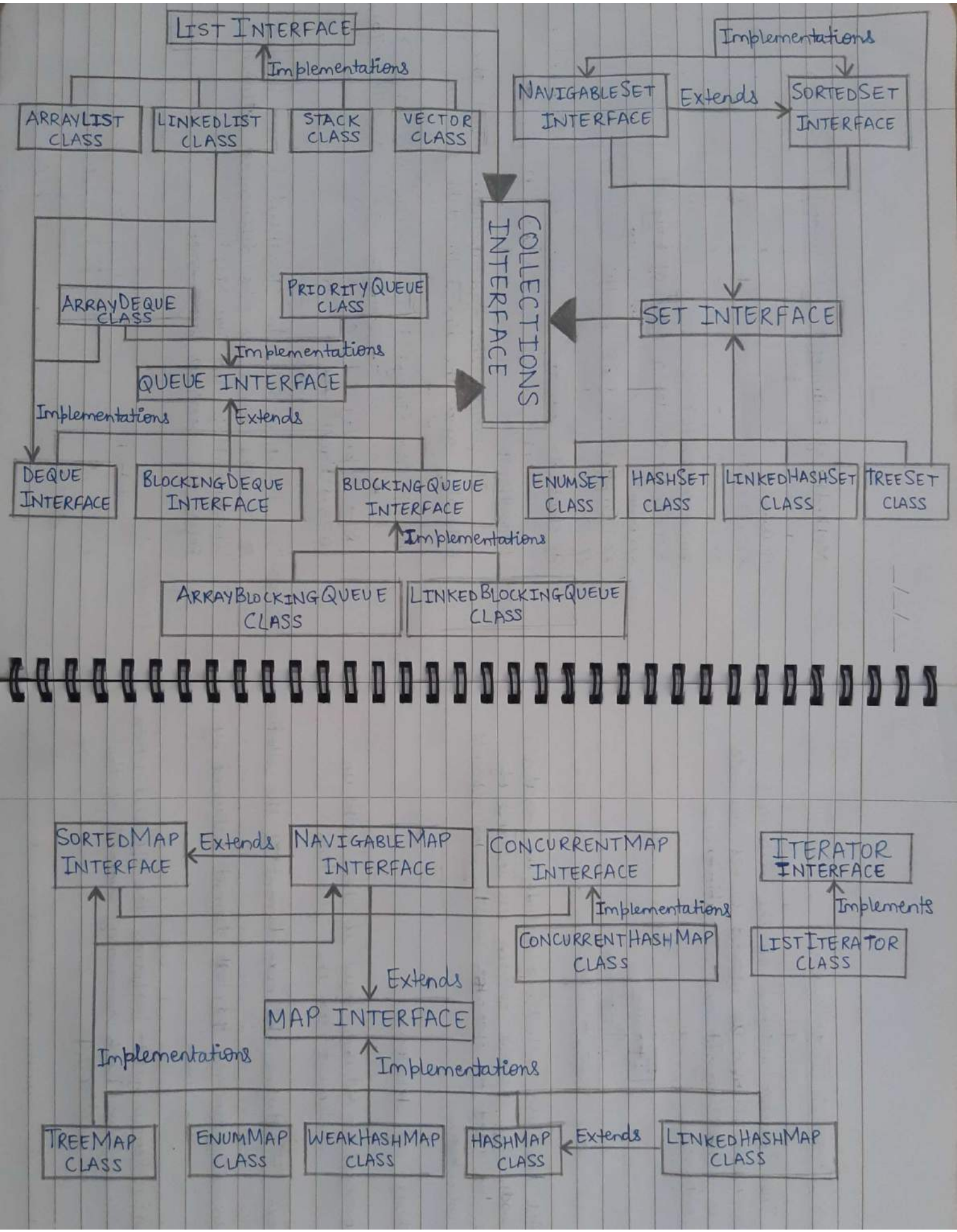
3) LinkedHashSet

It provides the functionalities of both the hashtable and the linked list data structure.

```
LinkedHashSet<Integer> numbers = new LinkedHashSet<>
                                        (8, 0.75);
```

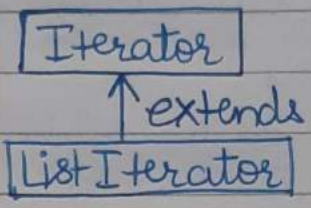
8 → capacity
0.75 → loadFactor.

NOTE:- The differences between LinkedHashSet and HashSet is are the same as the differences between LinkedHashMap and HashMap.
Refer the previous pages once.


LIST INTERFACE
Implementations
ARRAYLIST CLASS
LINKEDLIST CLASS
STACK CLASS
VECTOR CLASS
Implementations
NAVIGABLESET INTERFACE
Extends
SORTEDSET INTERFACE
COLLECTIONS INTERFACE
ARRAYDEQUE CLASS
PRIORITYQUEUE CLASS
Implementations
QUEUE INTERFACE
SET INTERFACE
Implementations
Extends
DEQUE INTERFACE
BLOCKINGDEQUE INTERFACE
BLOCKINGQUEUE INTERFACE
ENUMSET CLASS
HASHSET CLASS
LINKEDHASHSET CLASS
TREESET CLASS
Implementations
ARRAYBLOCKINGQUEUE CLASS
LINKEDBLOCKINGQUEUE CLASS
SORTEDMAP INTERFACE
Extends
NAVIGABLEMAP INTERFACE
CONCURRENTMAP INTERFACE
ITERATOR INTERFACE
Implementations
Implements
CONCURRENTHASHMAP CLASS
LISTITERATOR CLASS
Extends
MAP INTERFACE
Implementations
Implementations
TREEMAP CLASS
ENUMMAP CLASS
WEAKHASHMAP CLASS
HASHMAP CLASS
Extends
LINKEDHASHMAP CLASS

* Java Iterator Interface

The Iterator interface of the Collections framework allows to ~~create~~ access elements of a collection. It has a subinterface ListIterator.



All the Java collections include an iterator() method. This method returns an instance of iterator used to ~~iterator~~ iterate over elements of collections.

Methods of Iterator :-

1) hasNext() - returns true if there exists an element in the collection.

2) next() - returns the next element of the collection

3) remove() - removes the last element returned by the next()

4) forEachRemaining() - performs the specified action for each remaining element of the collection.

Eg:-

```
import java.util.ArrayList;
import java.util.Iterator;

class Main{
    public static void main(String[] args){
        ArrayList<Integer>numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println(numbers);

        Iterator<Integer> iterate = numbers.iterator();

        int number = iterate.next();
        System.out.println(number);

        iterate.remove();
        System.out.println(number);

        System.out.println("Updated ArrayList: ");

        while(iterate.hasNext()){
         iterate.forRemaining((value)->System.out.print
                                        (value + ", "));
        }
    }
}
```

Output:- [1, 3, 2]
1
1
3, 2,

Here, we have passed the lambda expression as an argument of the forEachRemaining() method.

→ is the lambda operator.

→ List Iterator Interface

It provides the functionality to access elements of a list.

It is bidirectional. This means it allows us to iterate elements of a list in both the directions.

It extends the Iterator Interface.

The List interface provides a listIterator() method that returns an instance of the ListIterator interface.

Methods of List Iterator:-

1) hasNext()

2) next()

3) nextIndex() - returns the index of the element that the next() method will return.

4) previous() - returns the previous element of the list.

5) previousIndex() - returns the index of the element that the previous() method will return.

6) remove() - removes the element returned by either

next() or previous()

7) set() - replaces the element returned by either next() or previous() with the specified element.

Eg:-

```
import java.util.ArrayList;
import java.util.ListIterator;

class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(3);
        numbers.add(2);
        System.out.println("ArrayList: " + numbers);

        ListIterator<Integer> iterate = numbers.listIterator();

        int number1 = iterate.next();
        System.out.println(number1);

        int index1 = iterate.nextIndex();
        System.out.println("Position of Next Element: " +
                           index1);

        System.out.println("Is there any next element? "+
                           iterate.hasNext());
```

Output

ArrayList : [1, 3, 2]
1

Position of Next Element : 1

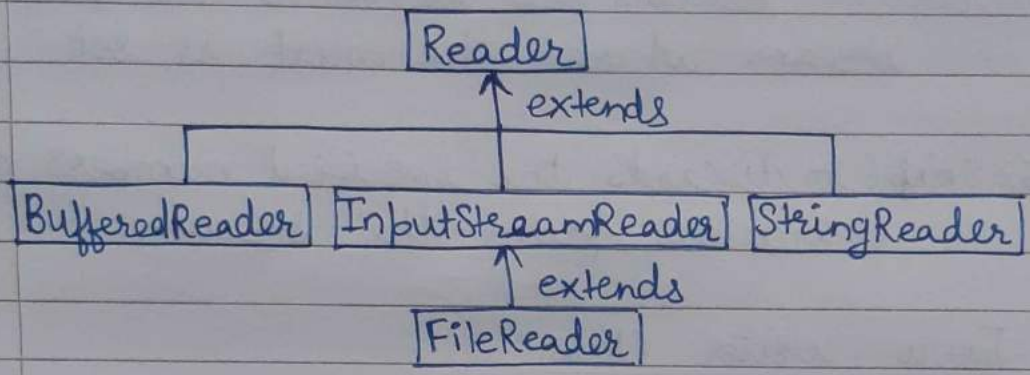
Is there any element ? true.

JAVA READER-WRITER

* Java Redder class

The Reader class of the java.io package is an abstract superclass that represents a stream of characters.

Since, Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.




```
Reader input = new FileReader();
```

Here, we have created a reader using the FileReader class. It is because Reader is an abstract class. Hence we cannot create an object of Reader. We can also create readers from other subclasses of Reader.

Methods of ~~Redor~~ Reader :-

1) ready() - checks if the reader is ready to be used.

2) read(char[] array) - reads the characters from the stream and stores in the specified array.

3) read(char[] array, int start, int length) - reads the numbers of characters equal to length from the stream and stores in the specified array starting from the start.

4) mark() - marks the position in the stream up to which data has been read.

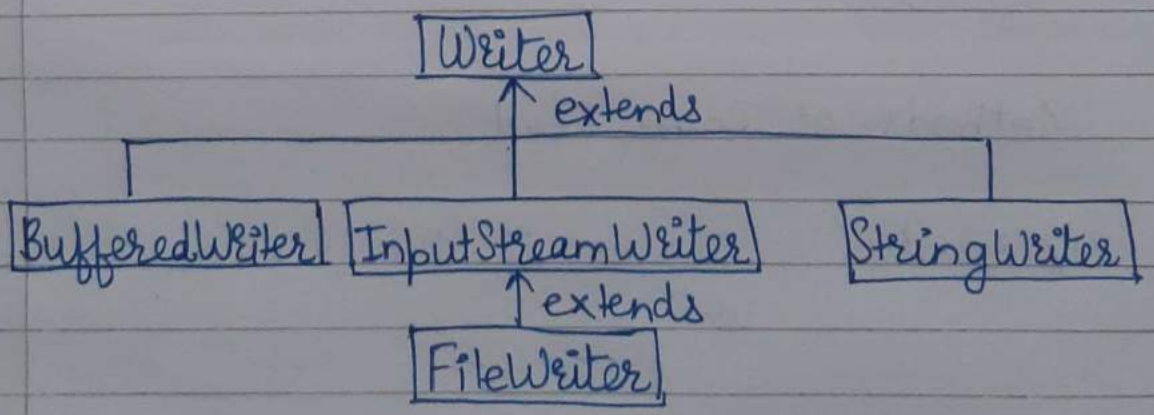
5) reset() - returns the control to the point in the stream where the mark is set.

5) skip() - discards the specified numbers of characters from the stream.

* Java Writer Class

The Writer class of the Java.io package is an abstract superclass that represents a stream of characters.

Since, Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.




```
Writer output = new FileWriter();
```

Here, we have created a writer named output using the FileWriter class. It is because the Writer is an abstract class. Hence, we cannot create an object of Writer. We can also create Writers from other subclasses of the Writer class.

Methods of Writer :-

1) write (char[] array) - writes the character from the specified array to the output stream.

2) write (String data) - writes the specified string to the writer.

3) append (char c) - inserts the specified character to the current writer.

4) flush() - forces to write all the data present in the writer to the corresponding destination.

5) close() - closes the writer.

1) Java InputStreamReader Class

It can be used to convert data in byte form into data in characters form.

2) Java OutputStreamWriter Class

It can be used to convert data in character form into data in bytes form.

3) Java FileReader Class

It can be used to read data (in characters) from files.

4) Java FileWriter Class

It can be used to write data (in characters) to files.

5) Java BufferedReader Class

It can be used with other readers to read data (in characters) more efficiently.

6) Java BufferedWriter Class

It can be used with other writers to write data (in characters) more efficiently.

7) Java StringReader Class

It can be used to read data (in characters) from strings.

8) Java StringWriter Class

It can be used to write data (in characters) to the string buffer.

9) Java PrintWriter Class

It can be used to write output data in a commonly readable form (text).

* Java Type Casting

The process of converting the value of one data type to another data type is known as Typecasting.

In Java, there are 13 types of data conversion. Here, we will only focus on the major 2 types :-

1) Widening Type Casting

In this type, Java automatically converts one datatype to another data type.

Here, the lower datatype is (having smaller size) is converted into the higher data type (having larger size). Hence, there is no loss in data.

This is why this type of conversion happens automatically.

This is also known as Implicit Type Casting.

Eg:-

```
public static void main (String [] args){
    int num = 10;
    System.out.println ("Integer: " + num);
```

```
        double data = num;
        System.out.println("Double: " + data);
    }
```

⇒ Integer: 10
Double : 10.0.

Here, we are assigning the int type variable named num to a double type variable named data.
Here, the Java first converts the int type data into the double type. And then assign it to the double variable.

2) Narrowing Type Casting

In this type, we manually convert one data type into another using the "parenthesis".
Here, the higher data type (having larger size) are converted into lower data types (having smaller size). Hence there is loss of data. This is why this type of conversion does not happen automatically.
This is also known as Explicit Type Casting.

Eg:-

```
public static void main (String[] args){
    double num = 10.99 ;
    System.out.println("Double: " + num);

    int data = int(num);
    System.out.println("Integer: " + data);
}
```

=> Double : 10.99
Integer : 10.

→ Consider an integer num and String data.
Now,
1) Type conversion from String to int :-

```
int num = Integer.parseInt(data);
```

2) Type conversion from int to String :-

```
String data = String.valueOf(num);
```

* Command Line Arguments

Consider a java file named Main.java.

1) To compile the code :-
javac Main.java

2) To run the code :-
java Main