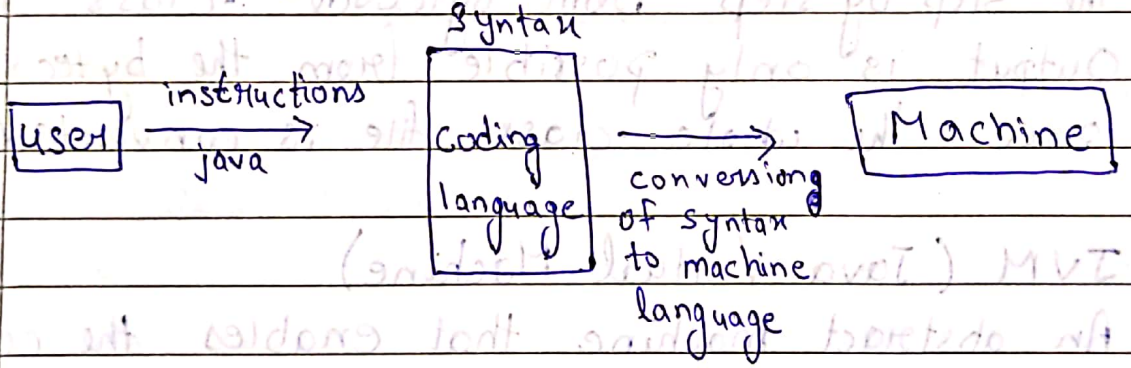
L1 - Java

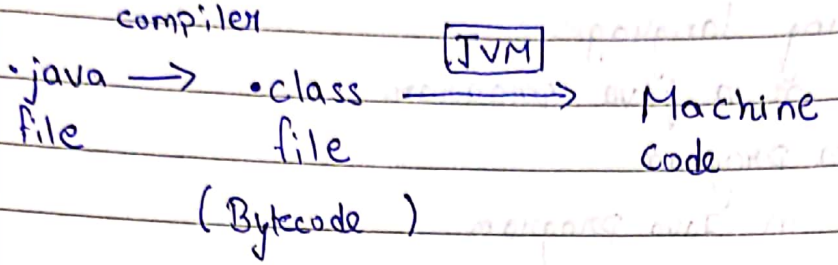
1- Programming language
2- Working of a java program
3- Basic java program
4- Keywords in Java program
5- Variables in java
6- Data types in java
7- Types conversion in java
8- Java comments

Programming language



Programming language is just the set of instruction that a user give to a machine. The instructions are coded in java for a human readable form and then gets converted into machine language using compiler so that machine can understand the instructions.

Working of a Java Program



First we write the code in .java file which gets converted into byte code by the compiler with the extension of .class. Later this .class file goes through JVM and gets converted into machine code. This machine code is the result of JVM interpreter that reads all the step by step from bytecode .class file. Output is only possible from the bytecode once the whole .class file is converted.

JVM (Java Virtual Machine)

An abstract machine that enables the computer to run java program.
When we run the code (Java program), java compiler first compiles your Java code to byte code. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

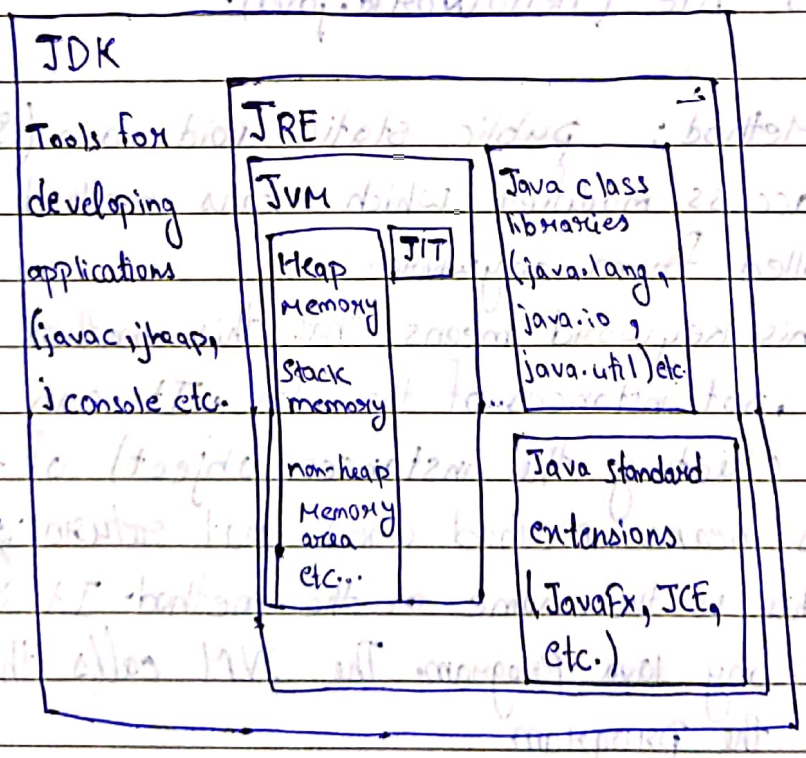
JRE (Java Runtime Environment)

A software package that provides Java class libraries, JVM and other components that

are required to run Java applications.

JDK (Java Development Kit)
A software development kit required to develop applications in Java. In addition to JRE, JDK also contains a numbers of development tools (compilers, Java Doc, Java Debugger, etc).

JVM performs following operations-
1- Loads source code.
2- Checks and verifies the source code.
3- Provides runtime environment to execute the code.



JDK → JRE → JVM

Basic Java Program

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

1 Class Definition: public class HelloWorld

- public: Public is a access modifier which makes class accessible by any other class.
- class: Keyword which is used to define a class.
- HelloWorld: name of class, also should be the name of file (HelloWorld.java).

2. Main Method: public static void main(String[] args)

- public: access modifier which means method can be called from anywhere.
- static: This keyword means that this method belongs to the class, not instances of the class. It can be called without creating the instances (object) of the class.
- void: This means method does not return any value.
- main: This is the name of the method. It is the entry point of any Java Program. The JVM calls this method to start the program
- String[] args: This is a parameter passed to the main method. It's an array of String objects. It allows command-line arguments to be passed to the program.

3. Printing to Console: System.out.println("Hello World")
- System: Class in java.lang package. It contains several useful class fields and methods. One of it is "out".
- out: static member of the system class. It is an instance of 'PrintStream' which has methods for printing text to the console.
- Println: This method of 'PrintStream' prints a line of text to the console. The text to be printed is passed as a parameter to this method. like "HelloWorld".

When you run this program the JVM looks for the 'main' method and starts executing it.

To run the program:
1- Save file with name HelloWorld.java
2- Open terminal to the absolute directory path of the file.
3- compile the program using:
javac HelloWorld.java

Run the program using:
java HelloWorld

Java keywords

These are some pre-defined words in java which cannot be used to give identifier name and these words are reserved to perform certain function in the java code.

Some keywords are-

1- abstract
2- Continue
3- for
4- new
5- switch
6- assert
7- default
8- goto
9- package
10- synchronized
11- boolean
12- do
13- if
14- private
15- this
16- break
17- double
18- implements
19- protected
20- throw
21- byte
22- else
23- import
24- public
25- throws
26- case
27- enum
28- instanceof
29- return
30- transient
31- catch
32- extends
33- int
34- short
35- try
36- char
37- final
38- interface
39- static
40- void
41- class
42- finally
43- long
44- strictfp
45 volatile
46- const
47- float
48- native
49- super
50- while

Java ~~Keywords~~ Variables

Variables are containers for storing data values.
Rules for naming variables in java:

- Java is case-sensitive. Hence, age and AGE are two different variables.
- Variables must start with either a letter or an underscore, _ or a dollar, $ sign.
- Variable names can't use whitespace.
- Keywords cannot be used in variable names.

Data Types in Java

A data type is an attribute associated with a piece of data that tells a computer system how to interpret its value.
A Primitive datatype specifies size & type and has no methods.
8 Types of Primitive Data Types

Type	Size	Default	Explaination
boolean	1 bit	false	Stores true or false values.
byte	1byte/8bits	0	Stores whole numbers from -128 to 127
short	2byte/16bits	0	stores whole no. from -32768 to 32767
int	4byte/32bits	0	stores whole no. from -2147483648 to 2147483647
long	8byte/64bits	0L	stores whole no. from -9223372036854775808 to 9223372036854775807
float	4byte/32bits	0.0f	Stores factorial no. Sufficient for 6 to 7 decimal digits.
double	8byte/64bits	0.0d	stores factorial no. sufficient for 15 decimal digits.
char	2bytes/16bits	'\u0000'	Stores single character/letter or ASCII value.

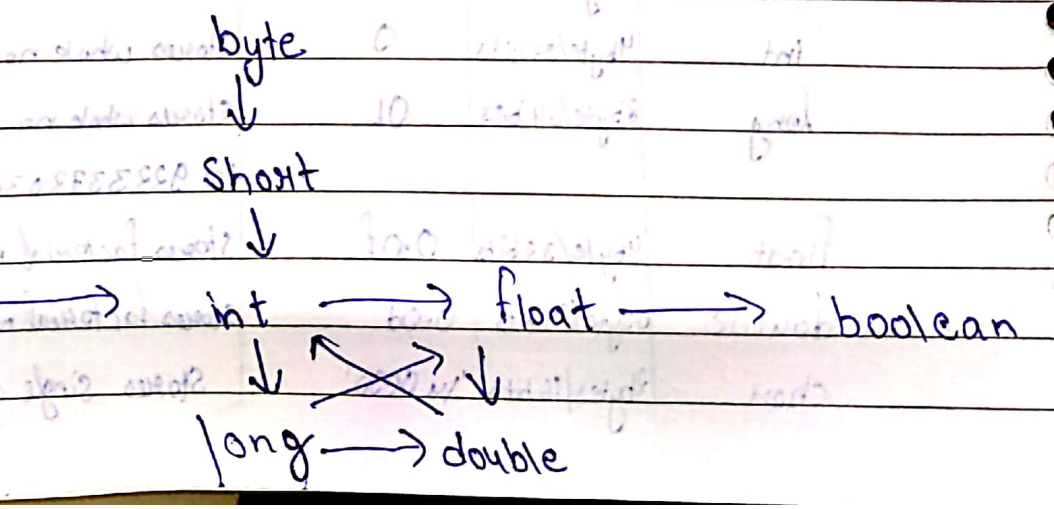
Non-primitive datatypes are called reference types because they refer to objects.

3 main non-primitive datatypes:
- String
- Arrays
- Classes

Differences in primitive	& non-primitive datatypes:
• Primitive are pre-defined	Defined by programmer (except string datatype).
• Cannot call methods to perform operations.	Can call methods.
• Always needs a value.	Can be null.
• Starts with lowercase.	Starts with uppercase.

Primitive datatypes are building blocks of non-primitive datatypes and one or more primitive datatype together creates a non-primitive datatypes.

Datatype Implicit Conversion



Implicit conversion are direct conversion which means we do not need to write any extra code for these conversions.

Explicit conversion

It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

```
int age = 12;
short newAge = (short) age;
```

Java Comments

Comments are portion of program which is completely ignored by the compiler.

Single line comments
Ctrl + /

Multiple line comments
ctrl + shift + /

Method comments for params description type /** then press Enter key.

L2 Binary Number System, Java Operators & User Input

Binary Number System

System only understands 0 or 1 so everything we provide gets converted into binary 0,1 to get stored in the system.

Convert decimal to Binary

2	26	0
2	13	1
2	6	0
2	3	1
	1	

$(26)_{10} \updownarrow (11010)_2$

2	49	1
2	24	0
2	12	0
2	6	0
2	3	1
	1	

$(49)_{10} \updownarrow (110001)_2$

Convert Binary to Decimal

$(110010)_2$

5	4	3	2	1	0
1	1	0	0	1	0
2^5	2^4	2^3	2^2	2^1	2^0

$32 + 16 + 0 + 0 + 2 + 0 = (50)_{10}$

$(1\ 0\ 1\ 0\ 1)_2 \Rightarrow$

1	0	1	0	1
2^4	2^3	2^2	2^1	2^0

$16 + 0 + 4 + 0 + 1 = (21)_{10}$

4 types of Number System

1- Decimal (Base 10) 0 - 9
2- Binary (Base-2) 0, 1
3- Octal (Base-8) 0-7
4- Hexadecimal (Base-8) 0-9, A-F

1 byte / 8 bit

Most Significant bit ← | 1 | 0 | 0 | 0 | 1 | 0 | 1 | → least significant bit

Sign bit ↓

1 → -ve
0 → +ve

⇓

-5

01111111 ⇒ +127

10000000 ⇒ -128 if stored in datatype byte

↓

128 if stored in datatype int

Binary Addition

1	1	0	1	0	→ 5
+	1	0	0	1	→ 9
0	1	1	1	0	→ 14

0	1	1	1	0	→ 14
0	1	0	1	1	→ 11
1	0	0	1		→ 25

Binary Subtraction

13 - 6 = 7

(13) 1101 ⟶

(6) 0110 ⟶ 1001 1's compliment

+ 1

(-4) 1010 2's compliment

1 0 1 0 1

1 0 1 0

0 1 1 1 → 7

21 - 8 = 13

2	21	1
2	10	0
2	5	1
2	2	0
	1	

$(10101)_2 = 21$

2	8	0
2	4	0
2	2	0
	1	

$(01000)_2 = 8$

01000 ⟶ 10111

+ 1

11000

```
  10101
+ 11000
  01101   (13)10
```

```
10000000  = (128)10
01111111
        +1
10000000  = -(128)10
```

Types of Operators in Java

1- Arithmetic operators
2- Assignment Operators
3- Relational Operators
4- Logical Operators
5- Unary Operators
6- Bitwise Operators

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Mudulo Operation (Remainder after division)

Operands
↓ ↓
12 % 7 = 5
12 / 7 = 1
12 + 7 = 19
12 - 7 = 5
12 * 7 = 84
↑
operator

Assignment Operators:

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Relational Operators

Operator	Description	Example
==	is equal to	3 == 5 (false)
!=	Not equal to	3 != 5 (true)
>	greater than	3 > 5 (false)
<	less than	3 < 5 (true)
>=	greater or equal to	3 >= 5 (false)
<=	Less or equal to	3 <= 5 (true)

Logical Operators

Operator	Example	Meaning
&& Logical AND	exp1 && exp2	true only if both true
|| Logical OR	exp1 || exp2	true if either one is true
! Logical NOT	!expression	true if expression is false

3==5 && 3>5 ⇒ false
3<=5 && 3==3 ⇒ True
3>=5 || 3==3 ⇒ True
!(3<5) ⇒ false
!(3>5) ⇒ True

Bitwise Operators (Bit Manipulation)

Operator	Description
~	Bitwise Complement
<<	Left shift
>>	Right shift
>>>	unsigned Right shift
&	Bitwise AND
^	Bitwise exclusive OR

Other Operators

++	Increment	a++ ⇒ a+1;
--	Decrement	a-- ⇒ a-1;
?:	Ternary operator	(if else)

__/__/16

Taking User Input using Scanner

To use scanner object, import java.util.Scanner package

```
Scanner sc = new Scanner(System.in);
  sc.nextInt( )
  sc.nextFloat( )
  sc.nextLine( )
  sc.next( )
```

Types of Input

```
nextLong()
nextFloat()
nextDouble()
next()  for string input
close()  for closing scanner after use (recommended)
```

Example:

```
main() {

  Scanner sc = new Scanner(System.in);
      System.out.print("Enter your age:");
      int age = sc.nextInt();
      System.out.println("You entered your age:" + age);
}
```

Output:

```
Enter your age: 18 ← provided by user
You entered your age: 18
```

__/__/17

Java Conditional Statements

1- if-else statement
2. if-else if-else statement
3- Nested if-else statement
4- Working with the logical operators
5. Java Ternary Operator
6- Java switch statement.

if-else Statements

This statement executes a certain section of code if the test expressing evaluated to true. Statement inside the body of else block are executed if the test expression is evaluated to false. This is known as the if-else statement in Java.

```
if (expression) {
    true;
} else {
    false;
}
```

if- else if - else statements

In java, we have an if...else... if ladder, that can be used to execute one block of code among multiple other blocks.

```
if (exp ){

}else if( ) {

} else if( ) {

} else {

}
```

Nested if-else statements

conditional statement inside conditional statement

```
if ( )  {

   if () {

   } else {

   }
} else {

}
```

__/__/19

Using Ternary operators

condition ? expression1 : expression2

Switch Statements

If you have many else-if statements then switch statements are better.

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the value of each case.
- If there is a match, the associated block of code is executed.
- The break and default keywords are optional, and will be described as to break the execution and give a default case if all the above cases do not match.

```
int day = 4;
switch (day) {
  case 1:
    Sout("Monday");
  break;
  case 4:
    Sout("Tuesday");
  break;
  default:
    Sout("No match");
```

No need of break as it's last statement.

__/__/20

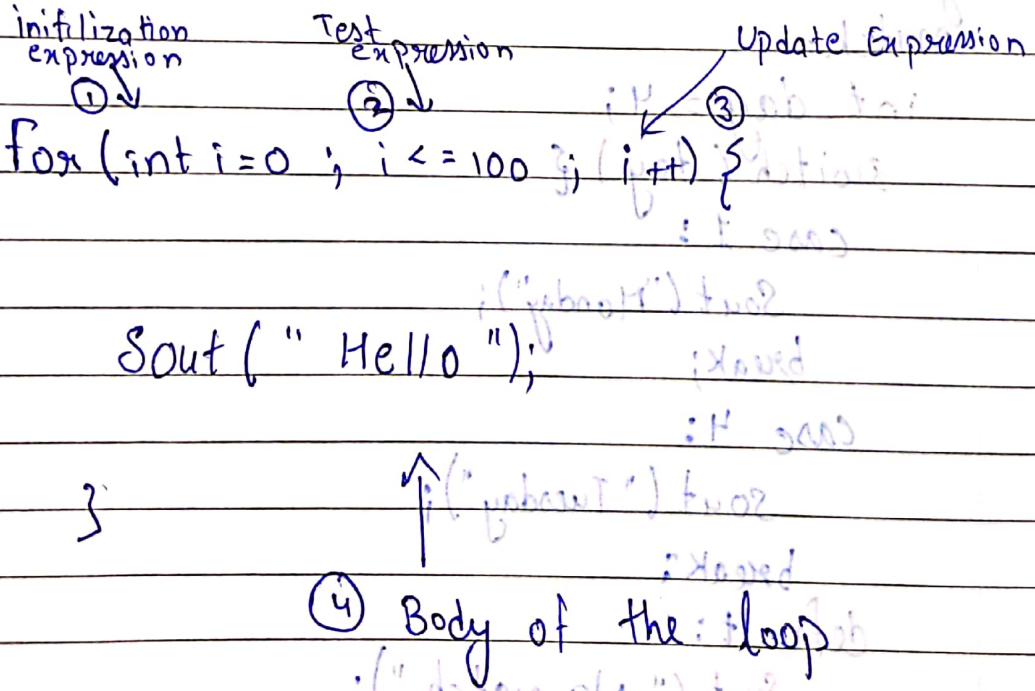
Loops in Java

1- for Loop
2- while loop
3- do-while loop
4- break & continue statements
5- Nested loops
6- Labeled break & continue statements

Elements of Java loops:

- Initialization Expression (s)
- Test Expression (condition)
- Update Expression (s)
- Body of the loop

Java for loop



Sum of n natural numbers

```
int finalnumber=0;
For (int sum=0, i=1; i<=n ; i++){

    sum = sum+i+;
    finalnumber = sum';
}

Sout( finalnumber);
```

Sum of n even numbers

```
int sum=0;
for(int i=0 ; i=i ; i<=n; i++){

    sum = sum + 2*i;

}
```

if n=5 then sum=30

While loop

can be used as for loop but one specific case is when we don't know how many time we want to do a task but we know when to stop.

```
int i=0;
while (i<10){

    i++;

}
```

- used to loop the linked list or trees ase we want to stop the iteration at the end of list & true:

do-while loop
will be executed atleast once.

```
int i = 8;

do {
    Sout (i);
    i++;
} while (i<=8);
```

The above example will be executed once.

break statement in Java
used to get out of the block of code:

```
while(testExpression) {
    if(condition to break) {
        break;
    }
}
```

Continue statement in Java

It is used to skip a certain condition in the loop.

```
for(int i=0; i<=5; i++){
    if (i == 4){
        continue;
    }
    sout(i);
}
```

Output : 0 1 2 3 5

Nested loops

loop inside another loop

```
for(int i=0; i<=5; i++){
    for(int j=0; j<=5; j++){
        sout(i, j);
    }
}
```

Output:

```
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 34 35 36
 :  :  :  :  :  :
```

Labeled break & Continue statements

When you're in a nested loop and writes break or continue statement then it affects only the inner loop but suppose you have a scenario where you want to break or continue to the outer loop also then you can use label of that loop or name of that loop to perform the action. label is just giving name to the loop.

```
label:
while (expression) {

    while (expression) {

        if (expression) {
            break label;
        }
    }
}
```

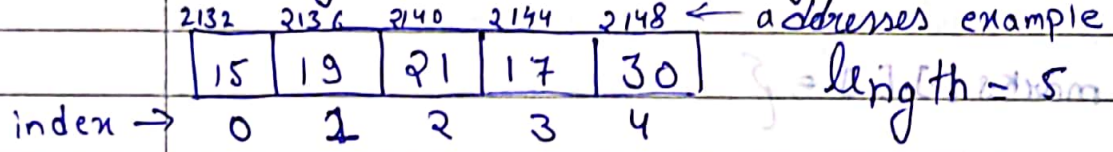
Arrays in Java

1. How Array works?
2. Creating and declaring Arrays.
3. for-each loop
4. Multi-dimensional Arrays

How Arrys Work

Arrays are stored in contigruous memory [consecutive memory locations].

Array → same data type



Creating Arrays

```
int[] array;   or   int array[];

intArray = new int[20] // allocating memory

int age[] = new int[20]; // declaration & initilization
```

for-each loop

```
String names[] = new String[10];

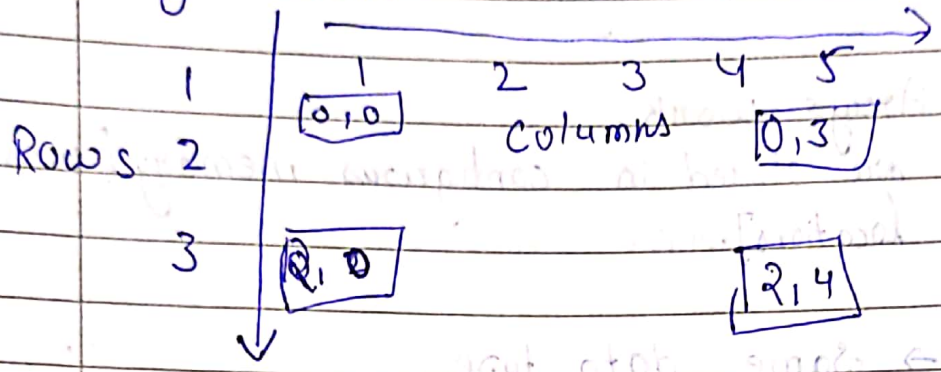
    for (String singleName : names){
        Sout (SingleName);
    }
```

Multi-dimensional Array

```
int arr[][] = new int[3][5];
```

3 x 5 = 15 ⇒ 15 integers storage

2D Array / Matrix




```
int marks[][] = {
        {12, 8, 16},
        {15, 19, 24},
        {14, 21, 29}
};
Sout(marks[1][1]);   // 19
```

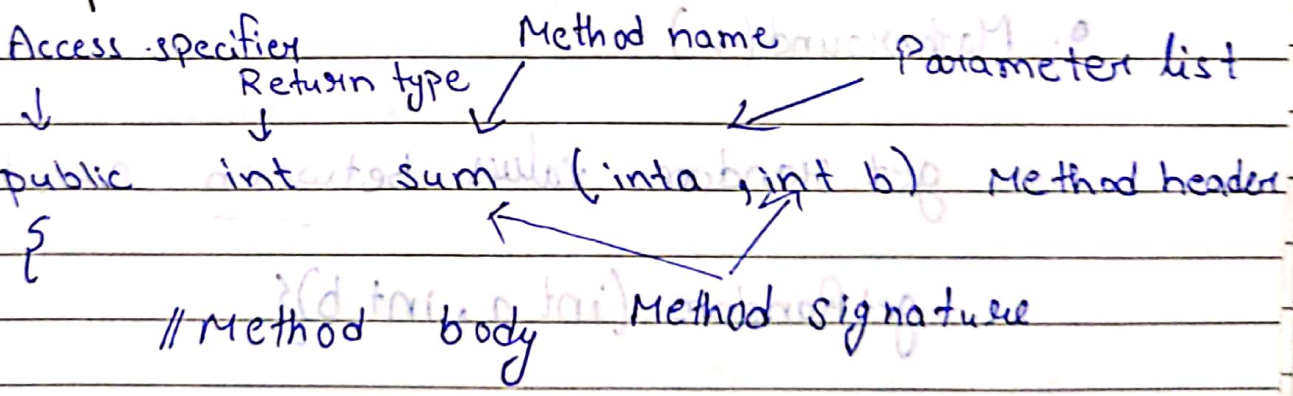
Methods in Java

1. How functions work?
2. Declaring Java Method
3. Calling a Method
4. Method return type
5. Method Parameters
6. Math library methods

How Methods work?

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code.

Components of a Method




```
Sum(5,10); // Calling a Method
```

Method Parameters

A method parameter is a value accepted by the method. As mentioned, a method can have any number of parameters.

Math Class Methods

1- Math.min(x, y)
2- Math.max(x, y)
3- Math.sqrt(x)
4- Math.pow(x, y)
5- Math.abs(x) gives absolute value. Negative to positive
6- Math.random() [0, 1] // 0 is inclusive, 1 is excluded
7- Math.floor(x)
8- Math.ceil(x)
9- Math.round(x)

get random values between a, b

```
getRandom(int a, int b){
    return (int) (Math.random()*(b-a+1)+a);
```

Example: value between 2, 10

Math.random()*(10-2+1)+2

Java String

1. Basics of String
2. How to create String in Java
3. How Strings are stored in Java
4. Immutability in strings
5. Comparing two Strings in Java
6. Java String Methods

String is an object that represents a sequence of char values. An array of characters works same as Java String.

String name = "Vishal";

similar to

char c[] = {'V', 'I', 'S', 'H', 'A', 'L'};

Primitive (char)

↓

Non-primitive (String)

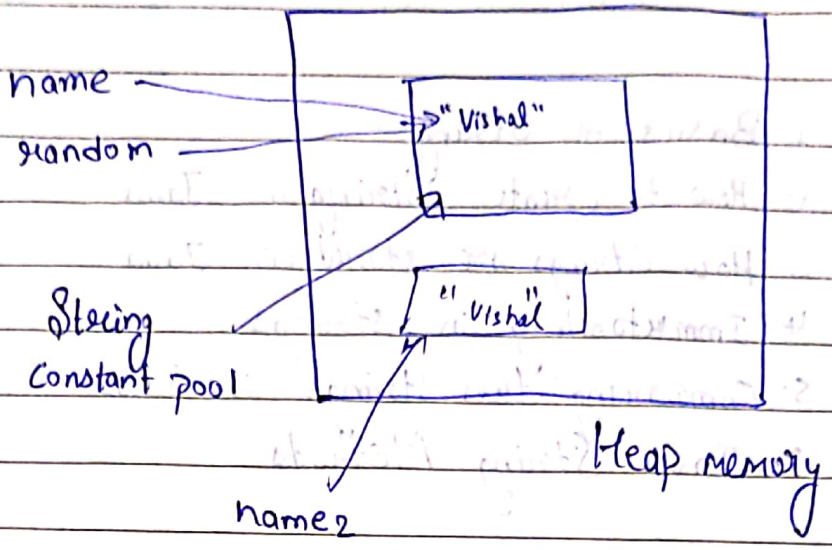
How to create String

By new keyword

String str = new String("Vishal");

By String literal.

String str = "Vishal";



If we use String literal to declare the String variable then it's value will be stored inside String constant pool and if we create one more String variable with same value then both variables will share same value from String constant pool to save memory.
If we use new keyword then the value of String variable will be stored in heap but not inside string constant pool and it'll not share it.

Immutable String in Java
String objects are immutable. Immutable simply means unmodifiable or unchangeable.
Once a string object is created its data or state can't be changed.

```
String name = "Vishal";
name = "Nikhil";
name = name + "Sharma";
```

Variable name with value "Vishal" will be created and string pool will store it.
Variable name is then changed to "Nikhil" but string "Vishal" is still in the memory and "Nikhil" will be created & stored in a different location. Same goes for name + "Sharma".

The unused values ("Vishal" & "Nikhil") will be cleared by garbage collector when it runs and finds out that the string "Vishal" & "Nikhil" is not referencing to any variable.

Comparing two Strings

== checks references and not the exact value so in case of String we use .equals() method.

equalsIgnoreCase(name) : compares ignoring case.

String Methods

1- toUpperCase()
2- toLowerCase()
3- trim()
4- StartsWith()
5- endsWith()
6- equals()
7- equalsIgnoreCase()
8 charAt()
9- valueOf()
10- replace()
11- contains()
12- substring()
13- split()
14- toCharArray()
15- isEmpty()
16- isBlank() // checks spaces also

String name = " Vishal";

- Sout(name.trim()) // "Vishal"
  Spaces removed
- Sout ("Carpet".startsWith("car")); // true
- Sout ("Carpet".endsWith("pet")); // true
- Sout("Carpet".charAt(3)); // 'p'
- int age = 123;
  String strAge = String.valueOf(age);

  Sout(strAge) // "123"

  strAge + 2 => "1232"
- String str = "Java is a good lang. Java is recommended"
  str.replace("Java", "Cpp");
  Sout(str) // "Cpp is a good lang. Cpp is recommended"
- Sout(str.contains("is") // true
- Sout(str.substring(2, 5) // "p_i"
  included ↑ excluded

endIndex in substring method is optional.

- ~~Sout~~ str.split(" "); returns String array

Object Oriented Programming

1- Classes & Objects
2- Constructors
3- Methods + constructor Overloading
4- this keyword
5- Inheritance
6- Method overriding
7- Super keyword
8- this vs Super keyword
9- final keyword
10- Java Packages
11- Access modifiers
12- Encapsulation
13- Data Hiding
14- Static keyword
15- abstract keyword
16- Abstraction
17- Interfaces
18- Inner class & Nested static classes
19- Anonymous Classes
20- Functional Interfaces
21- Lambda Expressions
22- How Java Memory works
23- Object Class
24- Polymorphism

Classes & Objects

1. Class is a blueprint which defines some properties and behaviours. An objects is an instance of a class which has those properties and behaviours attached.
2. A class is not allocated memory when it is defined. An object is allocated memory when it is created.
3. Class is a logical entity whereas objects are physical entities.
4. A class is declared only once. On the other hand, we can create multiple objects of a class.
5. A class is a way to arrange data and behaviour information. It is a template that must be implemented by its objects.
6. A class can also be seen as a user-defined datatype where any object of defined data type has some predefined properties and behaviours.

```
class Dog{

    String name;
    String age;        > Properties

    void walk(){       <- Behaviour
      Sout("walking");
    }
}
```

- The definition of class is stored in a memory called metaspace.
- The object gets its memory allocation from heap memory.

Method Overloading

1- Two or more methods can have same name inside the same class if they accepts different arguments.

2- Method overloading can be achieved by either:
a. changing the number of arguments.
b. or changing the data type of arguments.

3- It is not method overloading if we only change the return type of methods. There must be difference in parameters.

```
void func() { }

void func(int a) { }

float func(double a) { }

float func(int a, float b) { }
```

Constructors

1. Constructors are invoked implicitly when you instantiate objects.
2. Two rules of creating constructor are:
   a. The name of the same as class name.
   b. Constructor must not have a return type.
3. If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructors initializes instance variables with default values.
4. Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
5. A constructor cannot be abstract or static or final.
6. A constructor can be overloaded but cannot be overridden.

```
class Complex {

    int a, b;
    public Complex() {
        a = 10;
        b = 15;
    }
    public Complex (int a, int b) {
        this.a = a;
        this.b = b;
    }
}
```

this Keyword

- In Java, this keyword is used to refer to the current object inside a method or a constructor.
- We mostly use this keyword to remove any Ambiguity in Variable names. We can also use this keyword to invoke methods of the current class or to invoke a constructor of the current class.

Inheritance.

Inheritance 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 or derived class) and the existing class from where the child class is derived is know as superclass (parent or base class).

Class A
↓ extends
Class B

```
public class Vehicle{
    int tyres;
    int gates;
}
```

```
public class car extends Vehicle {
    int seats;
    public static void main() {
      Car c = new Car();
    }
}
```

Mehod Overriding

If a subclass provides implementation of the method that has been declared by one of its parent class, it is known as method overriding.

- Method overoiding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

Points to Remember:

1- Constructor cannot be inherited in Java.
2- Private members do not get inherited in Java.
3- Cyclic inheritance is not permitted in Java.
4- Assing parant reference to child objects.
5- Constructors get executed because of super() present in the constructor.

Types of inheritance in Java

- Single : sub-class derived form only one superclass.
- Multi-level: class derived from class which is derived already.
- Hierarchial: number of class derived from single base class.
- Hybrid : combination of two or more types of inheritance.

Super Keyword in Java

super is used to refer to the instance of the immediate parent class.

Uses of Super Keyword in Java:

- Used to refer to an instance value of the immediate parent class.
- Used to invoke a method of the immediate parent class.
- Used to invoke constructor of immediate parent class.

This vs Super Keyword

	this	Super
1-	this is an implicit reference variable keyword used to represent the current class.	Super is an implicit reference variable keyword used to represent the immediate parent class.
2-	this is to invoke the method of current class.	super is used to invoke methods of immediate parent class.
3-	used to invoke constructor of current class.	used to invoke constructor of the immediate parent class.
4-	this refers to the instance and static variables of the current class.	super refers to the instance and static variables of the immediate parent class.
5-	this can be used to return and pass as an argument in the content of a current class object.	super can be used to return and pass as an argument in the content of an immediate parent class object.

In brief -
this keyword is used to call constructor or methods of class inside another method.
Example -

```
class Car {

    void call {
        sout("car is starting");
    }

    void Car() {
        this.call();
        // initiaze the variables
    }

}
```

Super keyword is used to call constructor or methods of immediate parent.
Example -

```
class BMW {

    BMW() {
        super();
        //initialize variables
    }
```

Important → super needs to be the first statement in the method or constructor body.

final Keyword in Java

In Java, the final keyword is a non-access modifier that is used to define entities that cannot be changed or modified.

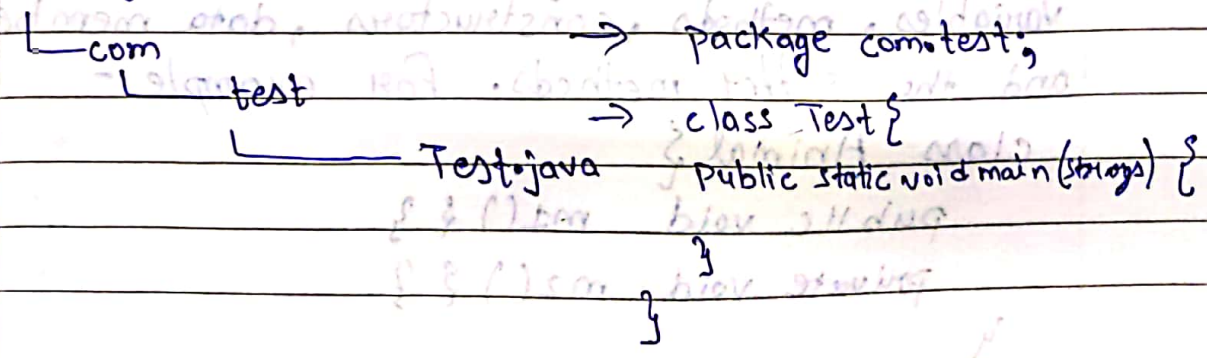
- final variable
  Variable with final keyword cannot be assigned again ~~or it value cannot be changed~~.

- final method
  method with final keyword cannot be overridden by its subclass.

- final class
  Class with final keyword cannot be extended or inherited from other classes.

Java Packages

A package is simply a container that groups related types (Java classes, interfaces, enumerations, and annotations). To define a package in Java, you use the keyword package. Java uses file system directories to store packages.



Importing a Package

- Java has an import statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.

```
import java.util.Date;    // imports only Date class
import java.io.*;  // imports everything inside java.io package
```

- In Java, the import statement is written directly after the package statement (if it exists) and before the class definition.

```
package package.name;
import package.ClassName;  // only import a Class

class MyClass{

    // body

}
```

Java Access Modifiers

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example -

```
class Animal{
    public void m1(){}
    private void m2(){}
}
```

Types of Access Modifiers

1 Default:
Same class → Yes
Same Package or Subclass → Yes
Same Package non-subclass → Yes
Different Package subclass → No
Different Package non-subclass → No
Sub-class ⇒ inherited class

2 Private:
Yes
No
No
No
No

3- Protected: (cannot be applied on class)
Yes
Yes
Yes
Yes
No

4- Public:
Yes
Yes
Yes
Yes
Yes

Encapsulation:

Encapsulation refers to the bundling of fields and 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.
A fully encapsulated class means making all data members private and using getter and setter methods to access.

Java Bean is the example of fully encapsulated class.

Advantages

- Can make any class read-only or write-only by providing only getter or setter methods.
- Provides control over what data will be inserted.
- Encapsulation is a way to achieve data hiding.
- Encapsulated class is easy-to-test. Better for unit testing.

Data Hiding

Data hiding is a way of restricting the access of our data membery by hiding the implementation details. Encapsulation also provides a way for data hiding. We can use access modifiers to achieve data hiding.

Static Keyword

If we want to access class members without creating an instance of the class, we need to declare the members static.

static variables can be accessed by calling by the class name of the class.
There is no need to create a class for accessing the static variables because static variables are the class variables and shared by all instance of class.

Static variables

- Only a single copy of the static variable is created and shared among all the instances of the class.
- Because it is a class-level variable, memory allocation of such variables only happens once when the class is loaded in the memory.
- If an object modifies the value of a static variable, the change is reflected across all objects..
- static variable can be used in any type of method: static or non-static.
- Non-static variables cannot be used inside static methods. It will throw a compile-time error.

```
public static int count = 0;
```

Static Methods

- The static members and methods belong to the class rather than the instance of the class. When the implementation of the particular method is not dependent on the instance variable and instance methods, In this case, we can make that method to be static.
- They can be accessed by the name of the class.
- The keywords such as this and super are not used in the body of the static method.
- The modification of the static field value is not ~~allowed~~ recommended but allowed.

Static Block

It will be the first thing to be execute in the class.

```
static {

    Sout ("Class loaded");

}
```

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare the abstract class.

- An abstract class can have both the regular methods and abstract methods.
- A method that doesn't have its body is known as an abstract method.
- Though abstract classes cannot be instantiated, we can create child class from it. We can then access members of the abstract class using the object of the subclass.
- If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

```
abstract class Vehicle {
    abstract void method1();
    abstract void method2();          // Abstract methods
             void method3(){          // complete method
}                          }

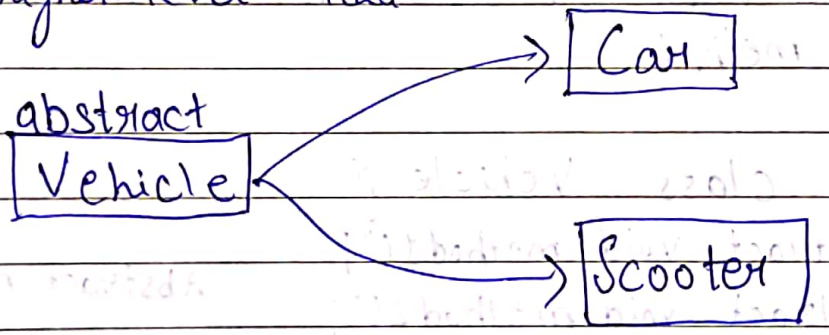
class Car extends Vehicle{
    @Override
     void method1(){
      // body
     }
     void method2(){
       // body
}  }
```

- Complete method are already defined with body in parent class so, overriding is optional.
- Abstract methods must be defined in child class.
- @Override annotation is optional and only used to spell-check the method names.
- If you define even single method as abstract then the class needs to be defined abstract.

What is Abstraction?

Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.

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



Java Interfaces

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

- interface keyword is used to create an interface in Java.
- Like abstract classes, we cannot create objects of interfaces.
- To use an interface, other classes must implement it. We use the implements keyword to implement an interface.

```
interface Language{
    public void getType();
    Public void getVersion();
}
```

All method are by default public & abstract so optional to provide these keywords.

Advantages

- Similar to abstract classes, interfaces help us to achieve abstraction in java.
- interfaces are also used to achieve multiple inheritance in java.
- Note: All the methods inside an interface are implicitly public and all fields are implicitly public static final.
- Interface now allow default methods inside interface which means providing implementation.

Inner classes & Nested classes

- A non-static nested class is a class within another class. It has access to members of the enclosing class (outer class). It is commonly known as inner class. Since the inner class exists within the outer class, you must instantiate the outer class first, in order to instantiate the inner class.

- Unlike inner class, a static nested class cannot access the members & variables of the outer class. It is because the static nested class doesn't require you to create an instance of the outer class.

- Using the nested class makes your code more readable and provide better encapsulation.

Example→

```
public class LearnInnerClass {
    class Toy {
        int price;
    }
    static class Playstation {
        int price;
    }
    public static void main(String [] args){
        LearnInnerClass obj = new LearnInnerClass();
        Toy toy = obj.new Toy();
        toy.price = 45;
        PlayStation playstation = new LearnInnerClass.Playstation();
    }
}
```

Anonymous Classes in Java

In java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.
A nested class that doesn't have any name is known as an anonymous class.

Anonymous classes usually extend subclasses or implement interfaces.
Here, Types can be
1- a superclass that an anonymous class extends.
2- an interface that an anonymous class implements.

```
mainClass {

    OuterClass obj = new OuterClass(){
        // methods
    }

    SuperInterface obj2 = new SuperInterface(){
        @Override
        public void interfaceMethod(){
        }
    }
}

class OuterClass {
}

interface SuperInterface{
}
```

Anonymous class cannot have multiple objects as it like a ~~variable~~ object.

Functional Interface
An interface that contains exactly one abstract method.
Functional Interface introduced in Java8 allow us to use a lambda expression to initiate the Interface's method and avoid using lengthy codes for the anonymous class implementation.

```
@FunctionalInterface
interface Sample{
    int calculate(int val);
}

Sample obj3 = (int x) -> {
    }
obj3.calculate(50);
```

Lambda Expression
(int x) -> x+1 // Single declared-type argument
(int x) -> {return x+1;} // Same as above
(x) -> x+1 // Single inferred-type argument, same as above
x -> x+1 // Paranthesis optional for single inferred-type case
(String s) -> s.length() // Single declared-type argument
(Thread t) -> {t.start();} // Single declared-type argument.

```
s -> s.length();  // Single inferred-type argument
t -> {t.start();} // Single inferred-type argument

(int x, int y) -> x+y // Multiple declared-type parameters
(x,y) -> x+y;         // Multiple inferred-type parameters
```

Marker Interface

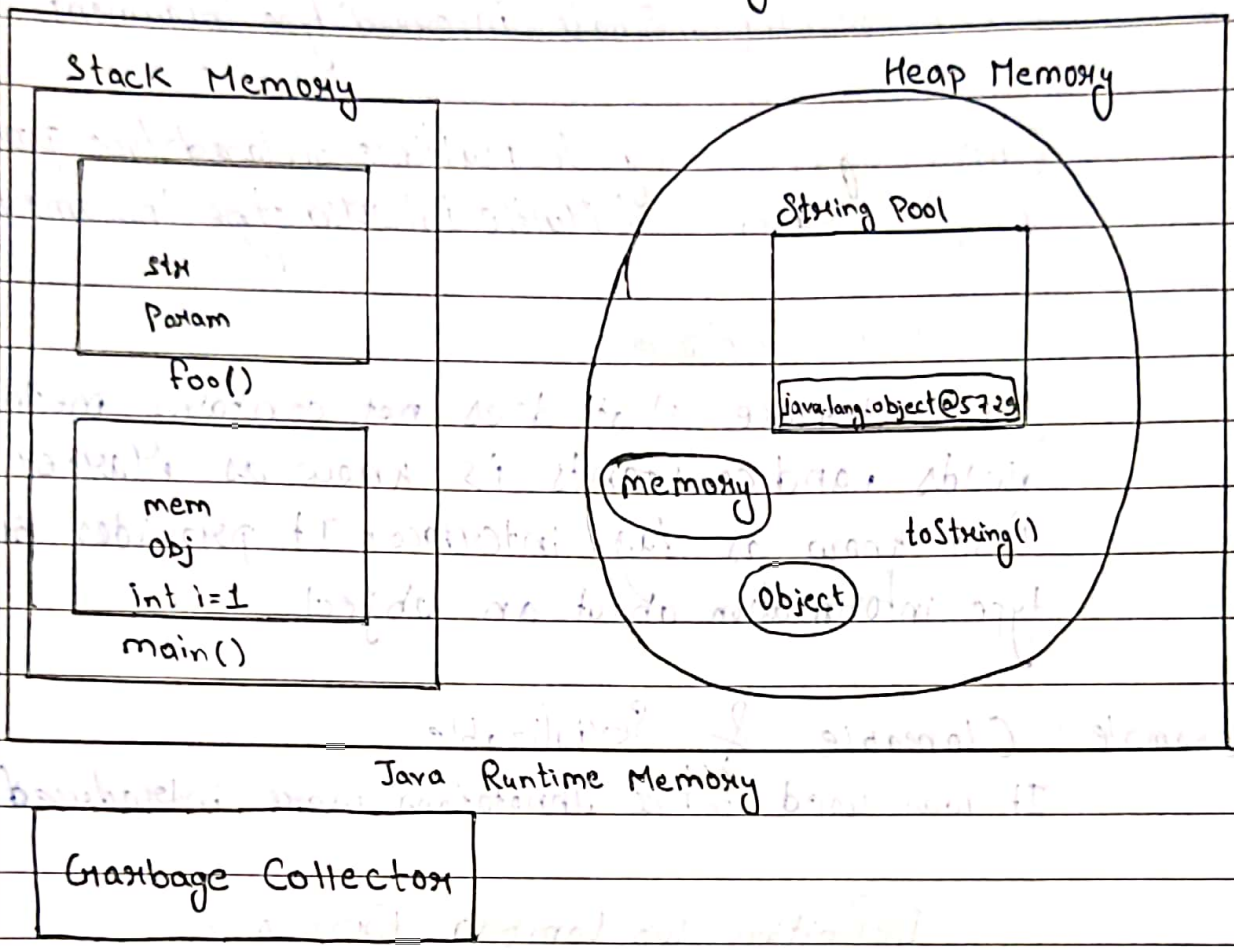
An interface that does not contains methods, fields, and constants is know as Marker interface. Also know as tag interface. It provides run-time type information about an object.

Example- Cloneable & Serializable
It was used before annotations were introduced.

Definition for Lambda Expressions

A Lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.
Lambda expressions were introduced in Java 8.
Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, curly braces can be used.

Java Heap and Stack Memory




```
public class Memory {
    public static void main(String[] args) {
        int i=1; //local variable
        Object obj = new Object();
        Memory mem = new Memory();
        mem.foo(obj);
    }
    private void foo(Object param) {
        String str = param.toString();
        System.out.println(str);
    }
}
```

Java Heap Memory

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create an object, it's always created in the Heap space.

Any object created in the heap space has global access and can be referenced from anywhere of the application. (if you have access). Heap is not thread safe.

Java Stack Memory

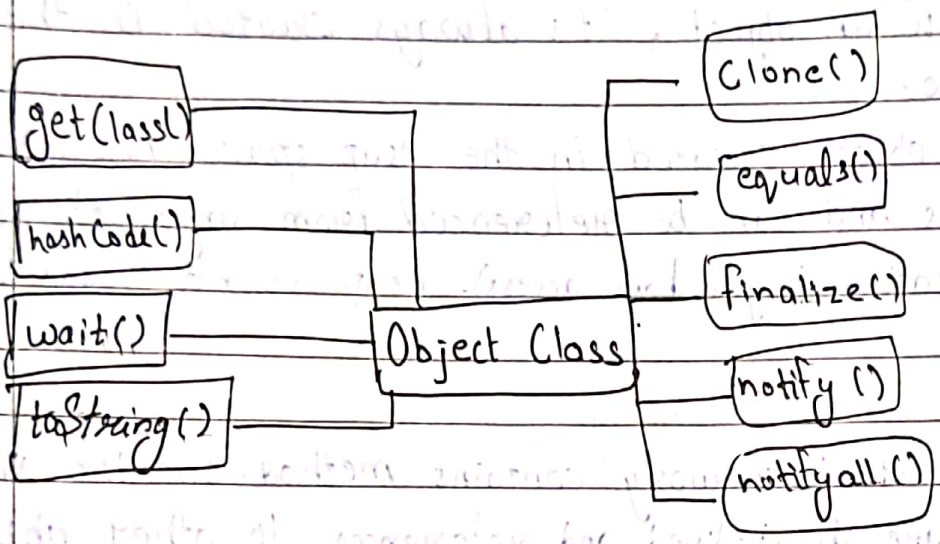
Java stack memory contains method-specific values that are short-lived and references to other objects in the heap that is getting reffered from the method.

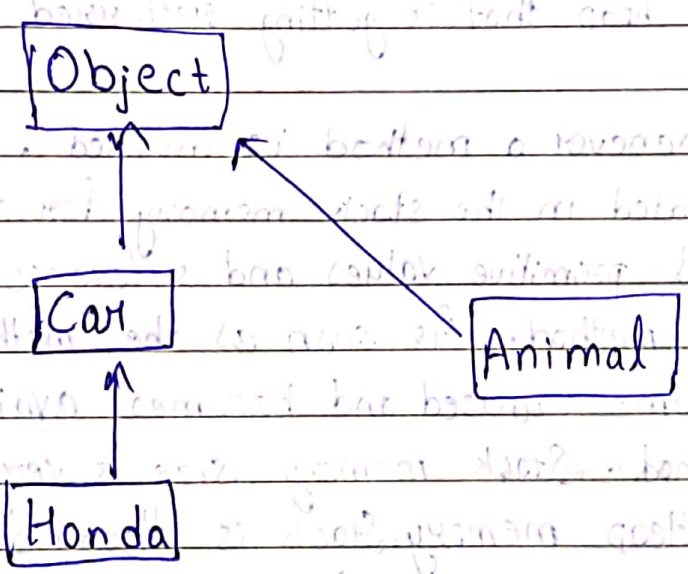
Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as the method ends, the block becomes unused and becomes available for the next method. Stack memory size is very less compared to Heap memory. Stack is thread safe (every thread creates it's own stack memory).

Object Class

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the object Class methods are available to all

Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.





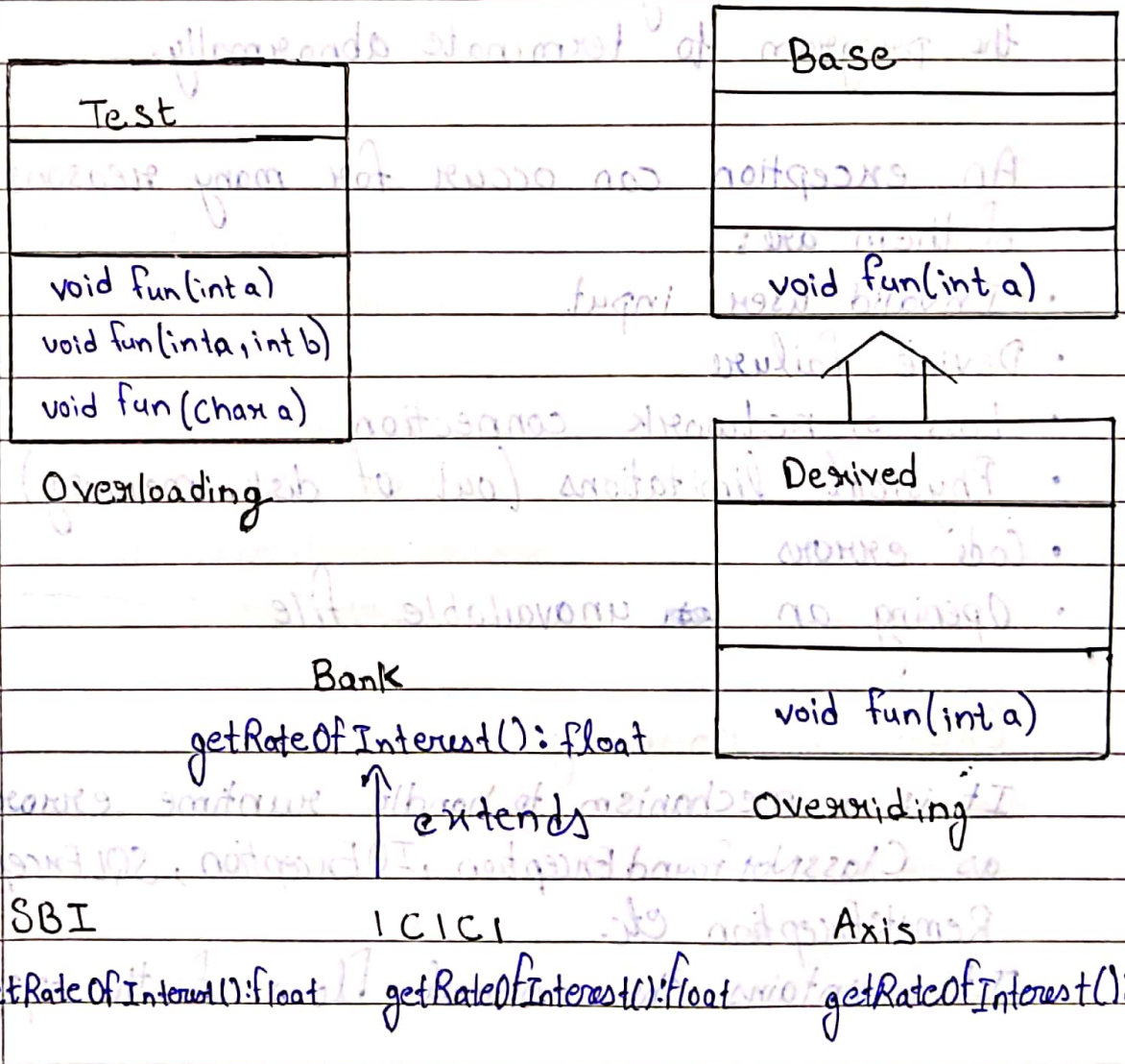
If Obj1 equals Obj2 that means their hashcode will be same.

Collision ⇒ When two objs get similar hashcode

Polymorphism

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, so it means many forms. There are two types of Polymorphisms.

- Compile-time Polymorphism
- Runtime Polymorphism (Dynamic method dispatch)



Exception Handling in Java

1- Java Exceptions
2- Java Exception Handling
3- try catch block
4- finally block
5- throw and throws keyword

Java Exceptions

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

An exception can occur for many reasons. Some of them are;

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Exception Handling

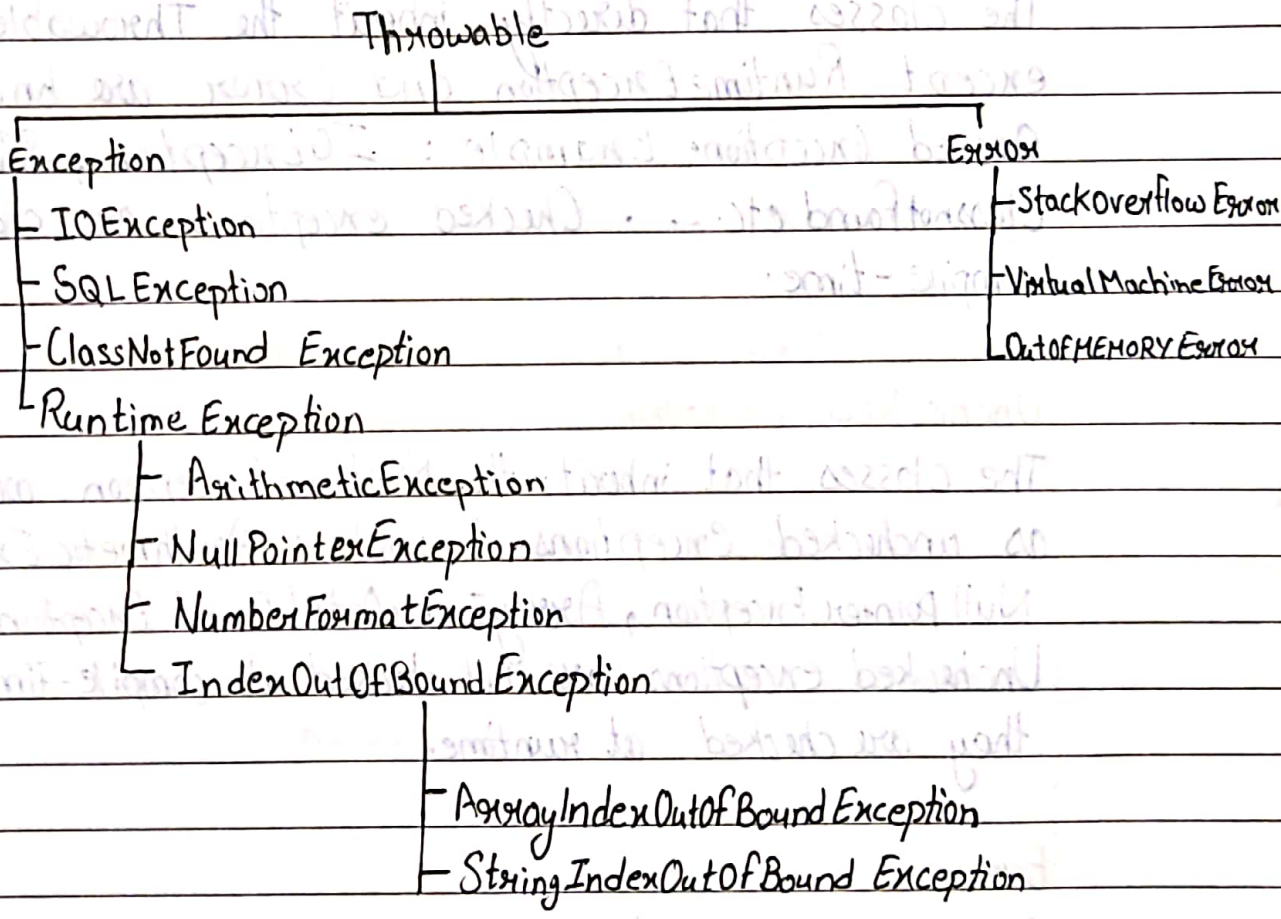
It is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException etc.

It maintains the normal flow of the application.

Suppose there are 10 statements in a Java program and an exception occurs at statement 5, the rest of the code will not be executed i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed.

Hierarchy of Java Exception Classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses. Exception and error.



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely.

1- Checked Exception
2- Unchecked Exception
3- Error

Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exception. Example: IOException, SQLException, ClassnotFound etc.... Checked exceptions are checked at compile-time.

Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundException. etc.... Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error

Errors are irrecoverable.
Example: OutOfMemoryError, StackOverflowError, VirtualMachineError

Keywords used in Exception handling

try: The 'try' keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

catch: The 'catch' block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by Finally block later.

finally: The 'finally' block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

throw: The 'throw' keyword is used to throw an exception.

throws: The 'throws' keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Example -

```
public class JavaExceptionExample {
   public static void main(String args[]){
    try{
        // code that may rise exception
         int data = 100/0;
      } catch(ArithmeticException e){
         System.out.println(e);
      }
        // rest of code in the program
      Sout("Rest of code");
   }
}
```

- `int a = 50/0; // ArithmeticException`

- ```
  String s = null;
  Sout( s.length()); // NullPointerException
  ```

- ```
  String s = "abc";
  int i= Integer.parseInt(s); // NumberFormatException
  ```

- ```
  int a[] = new int[5];
  a[10] = 50; // ArrayIndexOutOfBoundException
  ```
  ```

```
public class TestThrows{
   static void method() throws ArithmeticException{
       Sout("Inside the method()");
    throw new ArithmeticException("throwing ArithmeticException");
   }
   public static void main(){
    try{
       method();
      }
    catch(ArithmeticException e)
    {
      Sout("caught in main() method");
    }
  }
}
```

final, finally and finalize

- final is an access modifier
- finally is the block in Exception Handling.
- finalize is the method of object class.

Definition

final: final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.

finally: finally is the block in Java Exception Handling to execute the important code whether the exception

occurs or not.

finalize: finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.

Applicable to

final: final keyword is used with the classes, methods and variables.

finally: finally block is always related to the try and catch block in exception handling.

finalize: finalize() method is used with the objects.

Functionality

final: (1) Once declared, final variable becomes constant and cannot be modified.

(2) final method cannot be overridden by sub class.

(3) final class cannot be inherited.

finally: (1) finally block runs the important code even if exception occurs or not.

(2) finally block cleans up all the resources used in try block.

finalize: finalize method performs the cleaning activities with respect to the object before its destruction.

Execution

final: final method is executed only when we call it.

finally: finally block is executed as soon as the try-catch block is executed.
It's execution is not dependant on the exception.

finalize: finalize method is executed just before the object is destroyed.

final Example

```
public class FinalExampleTest{
    final int age = 18;
  void display(){
     age = 55;
  }
  public static void main(String[] args){
    FinalExampleTest obj = new FinalExampleTest();
    obj.display();
  }
}
```

fainally Example

```
public class FinallyExample{
    public static void main(String args[]);
      try{
          Sout("Inside try block");
          int data = 25/0;
          Sout(data);
      }
```

```
catch (ArithmeticException e){
        Sout("Exception handled");
        Sout(e);
    }
    finally{
        Sout("finally block is always executed");
    }
    Sout("rest of the code");
  }
}
```

finalize Example

```
public class FinalizeExample{
    public static void main(String[] args){
        FinalizeExample obj = new FinalizeExample();
        Sout("Hascode is: " + obj.hashcode());
        obj=null;
        System.gc();
        Sout("End of the garbage collection");
    }

    protected void finalize()
    {
        Sout("Called the finalize() method");
    }
}
```

Rules for Exception Handling with Method Overriding

- If the superclass method does not declare an exception
  If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare the unchecked exception.

- If the superclass method declares an exception
  If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

parent exception means not superclass, like if you declare ArithmeticException in superclass then you cannot override it with parent Exception class or RuntimeException class

Java Custom Exception

Custom exception means creating your own exception by deriving the class from the Exception parent class.

Reason for custom exception:

1- to catch and provide specific treatment to a subset of existing Java exceptions.
2- Business logic exceptions: These are the exceptions related to business logic and workflow. Useful for developer to understand the exact problem

Example 1

InvalidAgeException

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String str) {
        super(str);
    }
}

public class TestCustomException1 {
    static void validate(int age) throws InvalidAgeException{
        if(age<18){
            throw new InvalidAgeException("age is not valid");
        }
        else{
            Sout("Welcome to vote");
        }
    }
    public static void main(String args[])
    {
        try
        {
            validate(13);
        }
        Catch(InvalidAgeException e){
            Sout(e);
        }
        Sout("Rest of the code");
    }
}
```

Generics and Wrapper Classes

1- Wrapper Classes
2- Autoboxing & Un-boxing
3- Generics
4- Bounded Generics

Wrapper Classes

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

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

Need of Wrapper Class

1- The classes in java.util package handles only objects and hence wrapper classes help in (this) case also.
2- Data structures in the collection framework, such as ArrayList and Vector, store only objects (reference type) and not primitive types.
3- An object is needed to support synchronization in multithreading.

Autoboxing and Unboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example- conversion of int to Integer, long to Long, double to Double, etc.

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding datatype. is known as unboxing.

```
Integer obj3 = 12; // syntax for autoboxing

int age = obj3; // syntax for unboxing
```

Generics

Generics means parameterized types. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Example - Create a generic class

```
class GenericsClass <T> {

    // variable of T type
    private T data;
    public GenericsClass (T data) {
        this.data = data;
    }
    // method that returns T type variable
    public T getData(){
        return this.data;
    }

}
```

If you want more than one generic data types in your class then define it as given:

```
class GenericsClass <T, X, Y> {

    }
```

```
Dog<String> d1 = new Dog<>("Pablo");
```

This is how we use generic class

Reason of using Wrapper classes over Primitive types

- It is used if you need a nullable value.
- When you need an object and can't use primitive type, such as Generics.

Java Generic Method

Similar to the T generics class, we can also create a method that can be used with any type of data. Such as method is known as Generics Methods.

```
public <T> void genericMethod (T data) {....}

demo.<String>genericMethod ("Hello");
```

↑ Optional as IDE is able to identify from parameter itself.

Bounded Generics Types

In general, the type parameter can accept any data types (except primitive types). However, if we want to use generics for some specific types (such as accept data of number types only), then we can use bounded types.

In case of bounded types, we use the extends keyword.

Here, GenericClass is created with bounded type. This means GenericClass can only work with data types that are children of Number (Integer , Double and so on---).

```
class GenericClass <T extends Number> {
public void display() {

System.out.println("This is bounded generic class");
}
}
```

Collections in Java

- The collection in java is a framework that provides an architecture to store and manipulate the group of objects.
- Java collection can do operations like searching, sorting, insertion, manipulation and deletion.
- Java collection means a single unit of objects. Java Collection framework provides many different interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

- A Collection represents a single unit of objects. i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

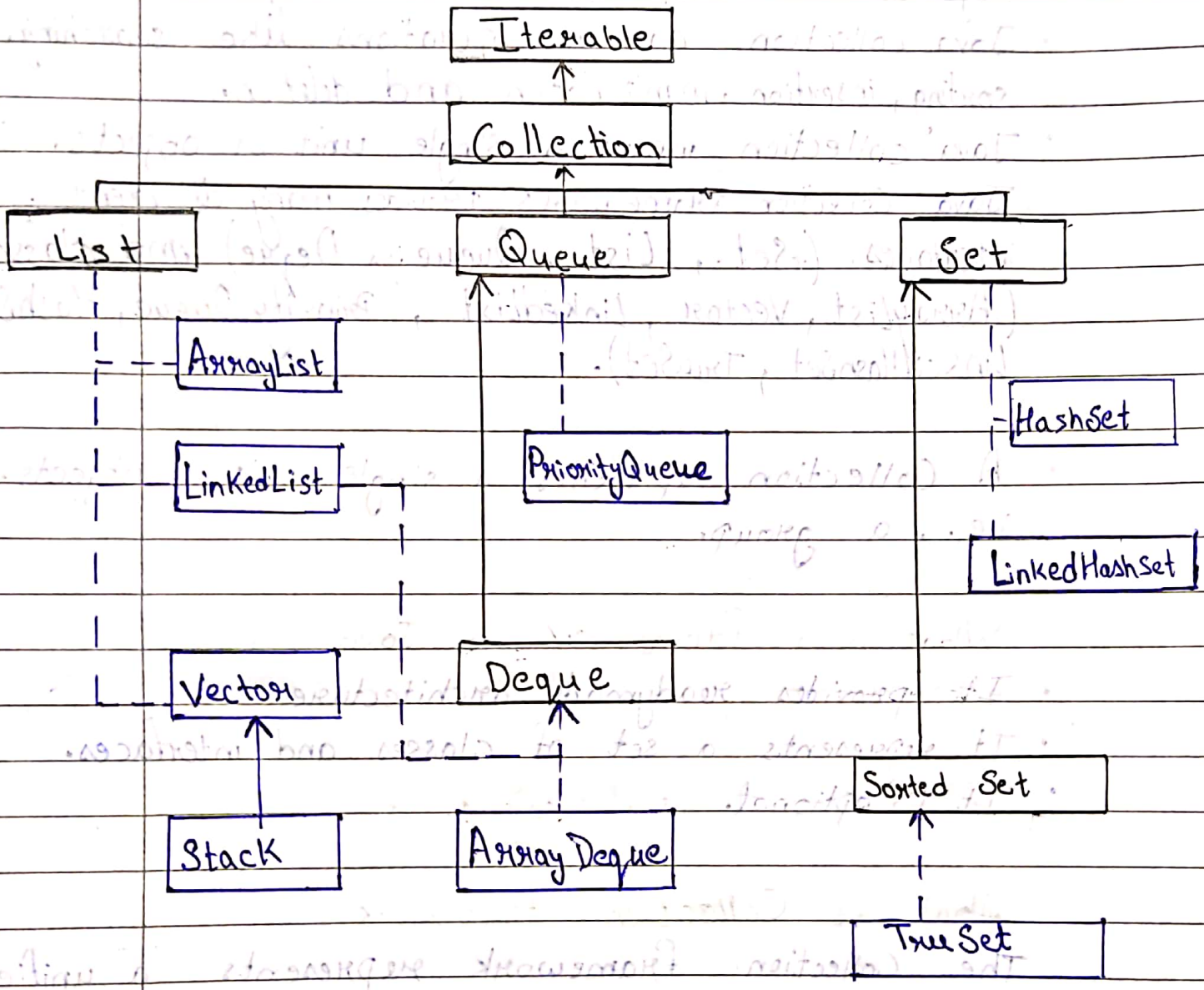
What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Hierarchy of Collection framework

The java.util package contains all the classes and interfaces for the Collection framework.



Methods of Collection framework

There are many methods in Collection interface.

1- public boolean add(E e): used to insert an element.
2- public boolean addAll(Collection <? extends E> c): used to insert specified collection of elements in the invoking collection.
3- public boolean remove(Object element): deletes an element from the collection.
4- public boolean removeAll(Collection <?> c): used to delete all the elements of the specified collection from the invoking collection.
5- default boolean removeIf(Predicate <? super E> filter): used to delete all the elements that specified predicate.

Example-

```
Predicate<Integer> pr = a -> (a%2!=0);
collection.removeIf(pr);
```

Explaination: removes all the elements which do not comes under P's table.

6. public boolean retainAll(Collection <?> c): used to delete all the elements of invoking collection except the specified collection.
7. public int size(): returns total number of elements.
8- public void clear(); removes total number of elements.
9. public boolean contains(Object element): used to search an element..
10- public boolean containsAll(Collection <?> c): used to search the specified collection in the collection.
11- public Iterator iterator(): It returns an iterator.

Example:

```
ArrayList<String> list = new ArrayList<>();
    list.add ("Vishal");
    list.add ("Nikhil");
    list.add ("Priyanshi");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()){
    String element = iterator.next();
    System.out.println(element);
}

}
}
```

12. public Object[] toArray(): Converts Collection to array.
13. public boolean isEmpty(): checks if collection is empty.
14. public <T> T[] toArray(T[] a): converts Collection to array. Here, the runtime type of the returned array is that of the specified array.
15. default Stream<E> parallelStream(): returns a possibly parallel Stream with the collection as its source. It allows to perform operations on the stream's elements concurrently.

Example:

```
List<String> upperCaseList = list.parallelStream().map(
                                String::toUpperCase)
                            .collect(Collectors.toList());
```

16- default Stream <E> stream: returns a sequential Stream with the Collection as its source.

Example-

```
List<String> upperCaseList = list.stream().map(String::toUpperCase).
                             collect(Collectors.toList());
```

17- default Spliterator<E> spilerator(): It returns a Spliterator over the specified elements in the Collection.

Example:

```
Spliterator<String> spliterator = list.Spliterator();
spliterator.forEachRemaining(System.out::println);
Spliterator<String> spliterator2 = spliterator.trySplit();

if (spliterator2 != null){
    spliterator2.forEachRemaining(System.out::println);
}
```

Spliterator is a type of iterator used for parallel processing. It provides additional characteristics over other iterators.

18- Public boolean equals(Object element): matches two collections or object inside collection.

19- public int hashcode: returns hashcode number of the collections.

Iterator Interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Three methods of Iterator interface :

1- public boolean hasNext(): It returns true if iterator has more element otherwise returns false.

2- public Object next(): Returns the element and moves the cursor pointer to the next element.

3- public void remove(): It removes the last elements returned by the iterator. It is less used.

Example:

```
ArrayList<String> list = new ArrayList<>();
list.add("Vishal");
list.add("Nikhil");

Iterator itr = list.iterator();
while (itr.hasNext()){
    Sout(itr.next());
}
```

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method:
Iterator <T> iterator()
It returns the iterator over the elements of type T.

Collection Interface

Collection Interface is the interface which is implimented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some method examples -

1. Boolean add(Object obj)
2. Boolean addAll(Collection c)
3. Void clear()

These methods are implimented by all the subclasses of collection interface.

List Interface

List interface is the child interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List Interface is implemented by classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface:

```
List <data-type> list1 = new ArrayList<>();
List <data-type> list2 = new LinkedList<>();
List <data-type> list3 = new Vector<>();
List <data-type> list4 = new Stack<>();
```

Various methods can be used to insert, delete and access the elements of list.

Queue Interface

Queue interface maintains first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. PriorityQueue, Deque, Arraydeque implements the Queue interface.

Example-

```
Queue <String> q1 = new PriorityQueue();
Queue <String> q2 = new ArrayDeque();
```

PriorityQueue

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities.
- Null values not allowed.

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet and TreeSet.

List Interface & It's Implementations

- The List interface in Java allows you to store elements in an ordered fashion. Also it is redundant which means duplicate elements are allowed.
- Allows its features or classes to keep the insertion order, thus all child classes can use an indexed or positional approach to store, remove or access elements.
- Contains ListIterator method which allows programmers to iterate the list in both directions.

ArrayList

- ArrayList is a list implementation that can dynamically add or remove contained elements.
- The size of an ArrayList can be dynamically increased or decreased, which makes it more flexible.

Properties:

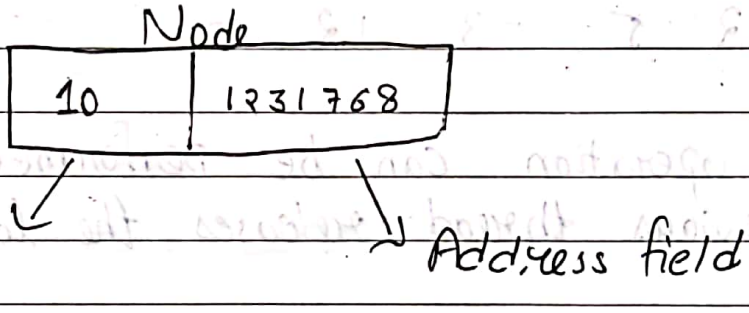
1- duplicates allowed.
2- maintains insertion order.
3- Non-synchronized (not thread safe).
4- Allows random access using index.
5- Manipulation is slower than LinkedList (shifting of elements happens).
6- Primitive datatype not allowed for creating ArrayList.

Wrapper class needs to be used.

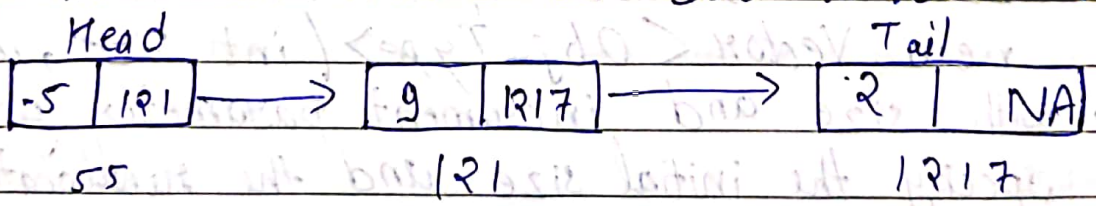
7- ArrayList is initialized with size.

LinkedList

- LinkedList class extends list and collection interface
- LinkedList elements are stored in non-contiguous locations. Each element is separate object with data and address field.



- Due to dynamicity and ease of insertions and deletions, LinkedLists are prefferred over the array.
- Nodes cannot be accessed directly. The traversal from head should be done to reach any node.



Time complexity for traversal = O(n)

- Since this class is created with Doubly LinkedList data structure, backward and forward insertion is possible.

LinkedList <Obj_Type> object = new LinkedList <Obj_Type>();

Vector

- Vector class in java implements array like structure which can grow its size to insert n number of items.
- Vector class supports random access of elements using indices and it is completely synchronized (Thread safe).

12 5 -3 12 ---- -- N

- No operation can be performed until the previous thread releases the lock.

Vector <Obj_Type> obj = new Vector <Obj_Type> ();

- Vector () Constructor will create a vector with the initial capacity of 10 elements.
- Size can also be specified in constructor.
  new Vector <Obj_Type> (int size, int incr);
- with size and increment parameter you can specify the initial size and the reallocation size of your vector.

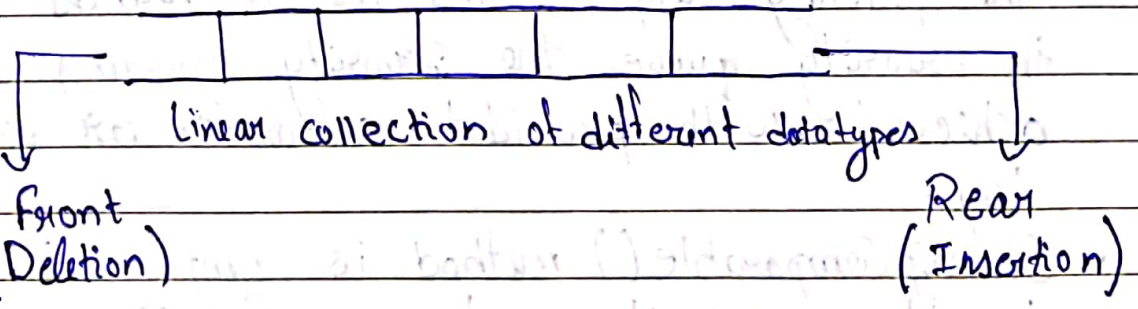
Stack

- Stack class is a child of vector class. And as its name suggests it is implementation of stack data structure.
- The stack follows Last in First out for insertion and removal of elements.

```
void push (Object element)
search (Object element)
boolean empty()
peek()
pop()
```

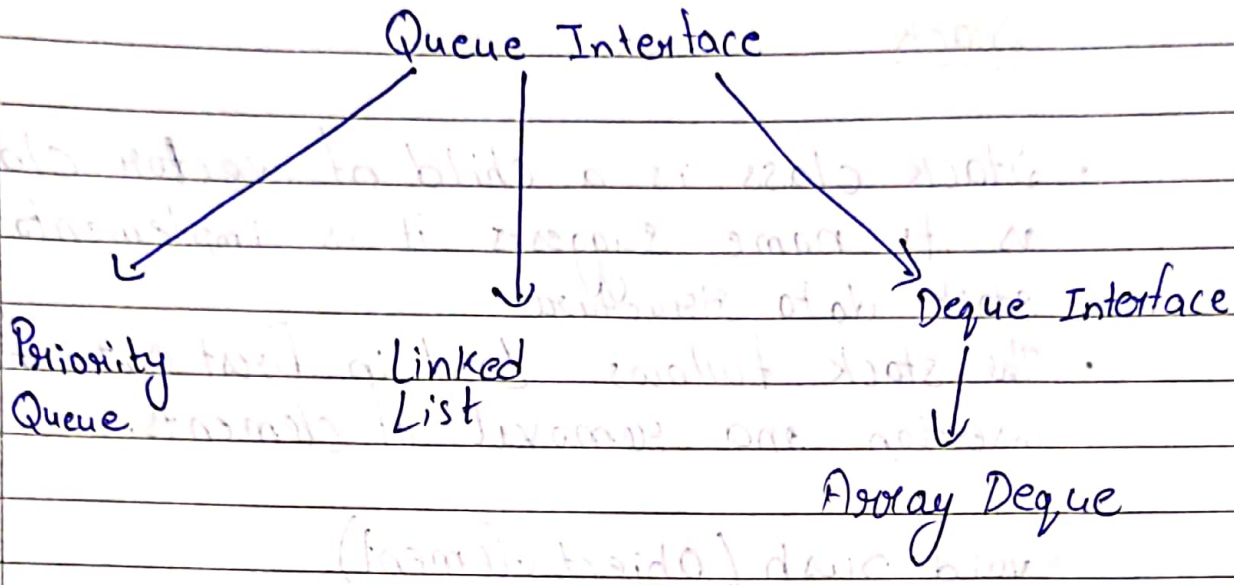
Queue Interface

Queue Interface in java extends the collection and iterable interface and follows the FIFO approach for inserting and removing elements



- First to enter, first to leave

Queue interface cannot be instantiated directly. We require concrete class to utilize its functionalities.

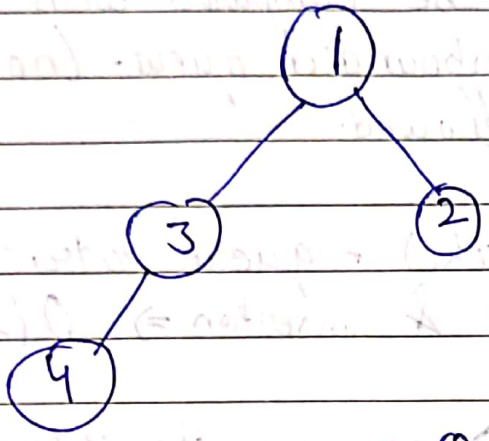


Priority Queue

- When items need to be processed depending on their priority, a PriorityQueue is implemented.
- This type of queue also employes First in first Out approach for inserting and removing the elements.
- Elements of this data container needs to be processed in terms of priority before putting them into the queue.
- The priority heap serves as a foundation for the priority queue. The priority ordering is either achieved with provided comparator or priority heap.
- ShiftUpComparable() method is employed internally to attain priority order in PriorityQueue. The method compares current element with its parent node at every insertion. If the order is not correct, parent element will be swapped with the current element.

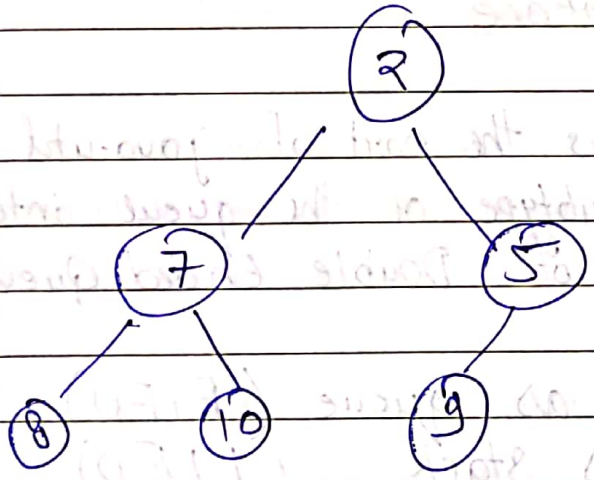
Constructing Priority Heap:

Insertion order : 4, 2, 1, 3

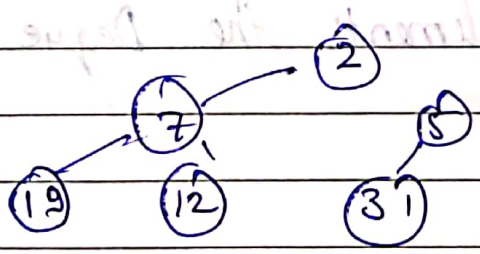


Output ⇒ 1, 2, 4, 3

Insertion order : 5, 8, 2, 7, 10, 9



Insertion order : 12, 19, 31, 7, 2, 5



Properties of Priority Queue:

1- Priority Queue can only contain comparable objects. means values that can be compared with each other.
2- Priority Queue is unbounded queue. (no size limit).
3- Null objects not allowed.
4- Not thread-safe.
5- poll(), remove(), peek() - queue retreival methods.
6- Complexity of removal & insertion ⇒ O(log n)

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(10);
```

Deque Interface

Deque interface is the part of java.util package and also the subtype of the queue interface. It is implementation of Double Ended Queue data structure.

- Can be used as queue (FIFO)
- Can be used as stack (LIFO)

ArrayDeque implements the Deque Interface.

ArrayDeque

ArrayDeque class provides the mechanism to use resizable-array as an Double Ended Queue data structure.

- ArrayDeque is special kind of array that grows and allows users to add or remove elements from both the sides.

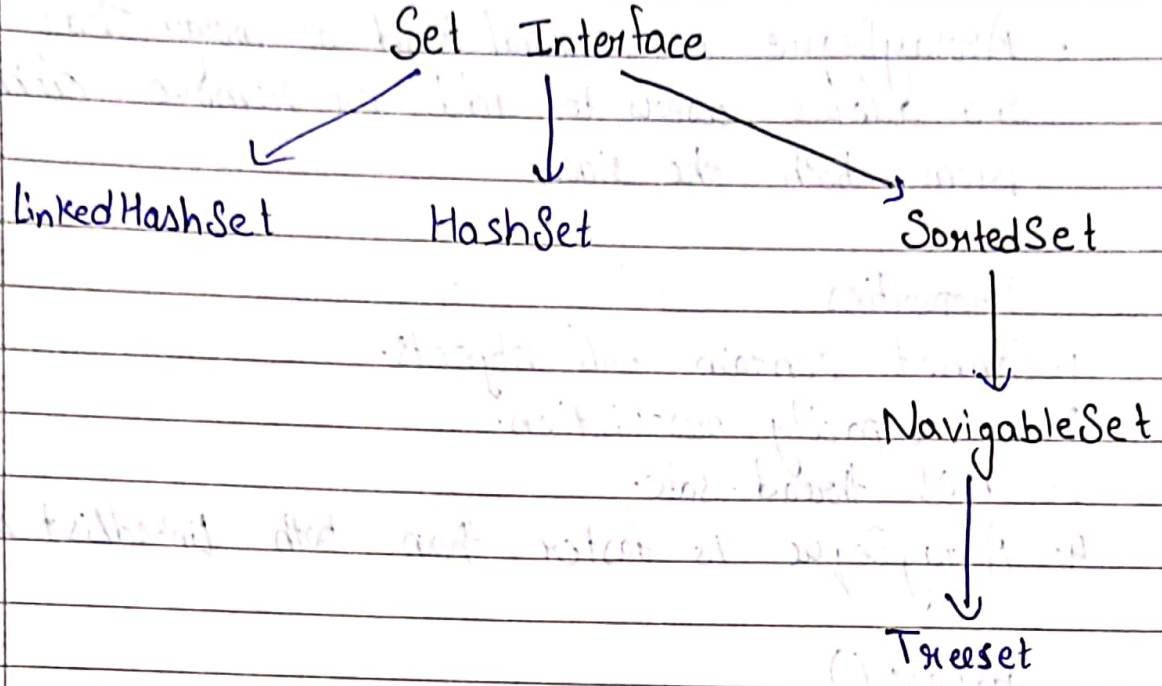
Properties

1- Cannot contain null objects.
2- No capacity restriction.
3- not thread-safe.
4- ArrayDeque is faster than both LinkedList & Stack.

addLast()
addFirst()

Set Interface

- Set interface in java extends the collection interface and it's the programming interpretation of mathematical set.
- Set interface restricts the insertion of duplicate elements by making use of a HashTable data structure.
- Set interface cannot be instantiated directly. Need a collection framework class that implements Set to put it into the use.



HashSet

- HashSet class is implemented using Hash Table data structure.
- HashSet gives constant time performance O(1).
- Ordering of elements in HashSet is not defined.
- Capacity and load factor are important variable for performance.

What is capacity?

The initial capacity is just the number of buckets produced when the HashSet class employs a hashtable internally.

Current size = initial capacity / size

The number of buckets will be doubled

What is load factor?
Load factor specifies how full the HashSet can become before its capacity is automatically raised.

No. of Insertions = Load Factor * Current Capacity

↓

The number of buckets will be doubled.

$$\text{Load Factor} = \frac{\text{No. of objects stored in HashSet}}{\text{Initial capacity of HashSet}}$$

For constant time performance it is adviced to make sure that the capacity is not too high and load too low.

Example-

Initial capacity = 16
Load factor = 0.75

The Rehashing will begin when insertion at index = $16 * 0.75 = 12$ happens.

That means if you insert only 11 elements then Memory Overhead can be avoided. However, the next insertion will take more time because rehashing will occur.

How HashSet works internally?

- The values we provide in hashset works as key of HashMap.
- The value parts of key: value pairs is a constant object: or dummy object.

- Note
HashSet can store collections like ArrayList, LinkedList & Vector.
HashSet makes sure that the collections or object you enter are unique.

Types of initialization in HashSet

1- HashSet<obj_type> object = new Hashset<obj_type>();

HashSet() constructor will create the HashSet with the initial capacity of 16 and load factor 0.75.

2- new HashSet<obj_type>(initial capaity);

This constructor will create Hashset with the custom initial capacity and load factor of 0.75.

3- new HashSet<obj-type>(initial_cap , Load-factor);
This constructor will create HashSet with the custom inital capacity and load factor.

4. new HashSet <obj_type> (Collection c);
This constructor will convert provided Collection object into HashSet.

Built-in Methods

- void add (element)
- addAll (coll. c)
- contains (ele)
- size()
- clear()
- iterator()
- remove()
- toArray (IntFunction <T[]> generator)
- hashCode()
- clone()
- stream()
- equals(Object o)

LinkedHashSet

- As the name implies, the LinkedHashSet is a blend of the Doubly linked list and the HashSet data structure.
- The HashSet maintains unique elements and Linked list maintain traversal order.
- The ordering in LinkedHashSet is achieved according to the insertion hierarchy.

SortedSet Interface

- SortedSet interface is implemented to add a feature that can store all the component of a set in a sorted manner.
- The class named NavigableSet extends the sortedset interface and provides the implementation to navigate through the set.
- Three set ; implements set, SortedSet, NavigableSet .

TreeSet Class

- Treeset uses Tree data structure for storing objects.
- Treeset can be ordered with an explicit comparator or by natural ordering depending on constructor definition.
- Treeset is essentiall a self-balancing binary search tree, similar to Red-Black Tree.
- In TreeSet, the operations such as add, delete, and search requires O(logN) time, which is quite effective considering it sorts the objects.
- It is completely non-synchronized. However, we can synchronize it externally using :

```
SortedSet<Obj-type> obj = Collections.synchronizedSortedSet
                                        (treeSet );
```

TreeSet Initializations

1- new TreeSet<obj-type>();
This generic constructor builds an empty TreeSet object in which elements will get stored in default sorting order.

2- new Treset<obj-type>(Comparator comp);
This constructor builds an empty TreeSet object with in which elements will need an external specification of the sorting order.

3- new TreeSet<obj-type>(Collection c);
This constructor is used when you want to convert the collection object to a TreeSet object.

4- new TreeSet<obj-type>(SortedSet s);
This constructor is used to convert the SortedSet object to the TreeSet object.

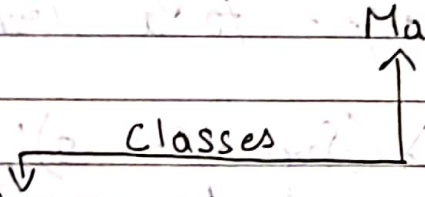
Methods unique to TreeSet:
1- pollFirst()
2- pollLast()
3- SubSet()
4- Floor()
5- last()
6- first()
7- lower()

Map Interface

In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual values. A map cannot contain duplicate keys, and each key is associated with a single value.

Example:

Key		Value
us	→	United States
br	→	Brazil
in	→	India



HashMap
TreeMap (also implements SortedMap)
EnumMap
LinkedHashMap
WeakHashMap

Methods

- put(K, V) → new value replace old value if inserted again.
- putAll() → insert entries of other map in this map.
- putIfAbsent(K, V)
- get(K)
- getOrDefault(K, defaultValue) → if key not found then return value
- ~~contains(K)~~ containsKey(K)

- replace (K, V)
- replace (K, old value, new value)
- remove (K)
- remove (K, V)
- keySet() → Returns set of all the keys
- values() → returns set of all the values
- entrySet() → returns set of all key value mapping present in the map.

HashMap

- HashMap<K, V> is a general-purpose implementation of the Map interface. It stores data in an array of linked lists, also known as a hash table. The hash table uses the hashCode() of keys to determine the index for storing the key-value pair.

Characteristics

- Performance: constant time for get, put, remove
- Order: does not guarantee any order of keys. Order may change after rehashing.
- Null values: one allowed in keys and multiple in values.
- Use HashMap when you need fast access to your data without concern for the order.

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.get("one");
```

LinkedHashMap

- LinkedHashMap <K, V> extends HashMap and maintains a doubly-linked list running through all its entries, which defines the order of iteration either by insertion order or access order (when configured).

Characteristics

- Order: Maintains insertion order by default. It can be configured to maintain access order (the order in which keys are accessed).
- Performance: Slightly slower than HashMap due to the overhead of maintaining the linked list.
- Null values: allow one null key & multiple values
- Use when you need insertion or access order.

TreeMap

TreeMap <K, V> is a Map implementation that uses a Red-Black tree structure. It order the keys based on natural order or by custom 'Comparator' provided.

Characteristics

- Order: maintains sorted key order.
- Performance: O(log n) for get, put, remove.
- Null vales: null keys not allowed. values allowed.
- Submaps: Supports operations like subMap, headMap, tailMap to get portions of Map.

- Use it when you need the keys to be sorted, and you can tolerate slightly slower performance compared to HashMap.

EnumMap

- EnumMap<K extends Enum<K>, V> is a specialized implementation of Map interface for use with enum keys.

Characteristics

- Performance: extremely efficient since it uses an array internally to store values.
- Order: Maintains natural order of enum keys.
- Null values: keys not allowed but values allowed.
- Space-Efficiency: More efficient than HashMap when using enums as keys.
- Use when you need a map with enum keys.

```
enum Day {Monday, TUESDAY, WEDNESDAY}
Map<Day, String> map = new EnumMap<>(Day.class);
 map.put (Day.Monday, "Work");
 map.put (Day.Tuesday, "GYM");
```

WeakHashMap

- It is a Map implementation that stores keys as WeakReference. If a key is no longer in use, the entry can be removed by garbage collector.

Characteristics

- Garbage collection: Entries are automatically removed when the key is no longer referenced outside of the map (i.e. garbage collected).
- Allows null keys and values.
- Use case: useful for caches where you don't want to prevent keys from being garbage collected.
- Use for caching or memory-sensitive data structures where you do not want the presence of a key in the map to prevent it from being garbage collected.

```
Map<Object, String> map = new WeakHashMap<>();
Object key = new Object();
map.put(key, "value");
key = null; // key is now eligible for garbage collection.
```

Summary:

- HashMap: general-purpose map with no ordering.
- LinkedHashMap: Maintains insertion or access order.
- TreeMap: Sorted keys with Comparator support.
- EnumMap: Efficient map for enum keys.
- WeakHashMap: Allows garbage collection for keys.

Comparable and Comparator

In Java Comparable and Comparator are two interfaces used for sorting objects. They define how objects are compared to one another. Though they serve a similar purpose, they have different use cases and implementations. Let's explore each in detail:

Comparable Interface

- The Comparable interface is used to define the natural ordering of objects of a class. It has a single method, compareTo(T o). which must be implemented by the class whose objects are to be ordered.

Syntax:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Characteristics:

- Method compareTo(T o) returns a negative integer, zero, or a positive integer if the object is less than, equal to or greater than the specified object, respectively.
- You can define only one default sorting logic for a class.
- The class itself defines how its objects should be ordered
- Modifying the natural order requires changing the

class implementation.

Example:

```
public class Student implements Comparable<Student>{
 private String name;
 private String rollNo;

public Student (String name, int rollNo){
    this.name = name;
    this.rollNo = rollNo;
}

@Override
Public int compareTo(Student other){
     return this.rollNo. - other.rollNo;
 }
}
```

This example sorts Students by their roll number.

Usage:

```
Collecting.sort(students); // uses Comparable(natural order)
```

Comparator Interface

- The Comparator interface is used to define an external, custom ordering for classes. It allows you to create multiple sorting sequences without modifying the original class.

Syntax:

```
public interface Comparator<T>{
    int compare (To1, To2);
}
```

Characteristics

- Method compare (To1, To2) returns a negative integer, zero, or a positive integer if the first argument is less than, equal to, or greater than the second, respectively.
- You can define multiple sorting strategies for same class.
- The sorting logic is external to the class.
- Allows sorting without altering the class implementation.

Example:

```
public class NameComparator implements Comparator<Student>{
@Override
    public int compare (Student s1, Student s2){
        return s1.getName().compareTo(s2.getName());
    }
}
```

This compareTo method is coming from String class

```
public class RollNoComparator implements Comparator<Student>{
 @Override
    public int compare(Student s1, Student s2){
      return s1.getRollNo()- s2.getRollNo();
  }
}
```

usage:

```
List<Student> students = new ArrayList<>();
students.add(new Student("Vishal", 20));
students.add(new Student("Nikhil",25));

// Sorting by Name

Collection.sort(students, new NameComparator());

// Sorting by Roll No.

Collection.sort(Students, new RollNoComparator());
```

Comparison between Comparable & Comparator

	Feature	Comparable	Comparator
1-	Interface location	java.lang package	java.util package
2-	Method to implement	compareTo(To)	compare(To1, To2)
3-	Single or multiple sorting	Single	Multiple
4-	Where to define sorting logic	Inside the class that implements Comparable.	Outsid the class, often as separate class
5-	Modification requirement	Changing the class for sorting	No need to modify class.
6-	Example use case	Natural order (i.e String or Integer	Custom order i.e Name or rollNo

In short, Comparable is for a single natural ordering defined within the class, while Comparator is for multiple custom orderings that are external to the class.

Sorting using Lambda Expressions

1- Sorting by a Single Attribute using Lambda

```
students.sort ((s1, s2) -> s1.getName().compareTo(s2.getName()));
```

2- Sorting by multiple Attribute

```
students.sort (Comparator.comparing(Student::getName).thenComparing(
                                        Student:getRollNo));
```

3- Reversing order using Lambda

```
students.sort(Comaarator.comparingInt(Student::getRollNo).reversed());
```

String , StringBuffer & StringBuilder

1. String

String is a class in java that represents a sequence of characters. It is immutable, meaning once a String object is created, it cannot be changed. Any operation that appears to modify a String actually creates a new string object.

```
String s = "Hello";
s = "Hello World";
```

- s initially points to "Hello";
- A new string object "Hello World" is created and s points to it.

2. StringBuffer

Stringbuffer is a class in java that is used to create mutable (modifiable) strings. Unlike String, StringBuffer objects can be modified after they are created. StringBuffer is synchronized, making it thread-safe, which means it can be used by multiple threads at the same time without causing inconsistencies.

```
StringBuffer buffer = new StringBuffer("Hello World");
buffer.append(" World");
Sout(buffer); // Hello World
```

StringBuilder

StringBuilder is similar to StringBuffer. It is used to create mutable strings. It is non-synchronized, meaning it is not thread-safe. This makes StringBuilder faster than StringBuffer in situations where thread safety is not a concern.

```
StringBuilder builder = new StringBuilder("Hello");
builder.append("Hello World");

Sout(builder); Hello Hello World
```

Comparision

Feature	String	StringBuffer	StringBuilder
Mutable	No	Yes	Yes
Thread-safe	Yes	Yes	No
Performance	Slow (for multiple modifications)	Slower due to synchronization	Fast (No synchronization overhead)
Use Case	Immutable data or light modification	Multi-threaded environment	Single-threaded environment with frequent modification.

Conclusion

- Use String for immutable text.
- Use StringBuffer for mutable string in a thread-safe manner.
- Use StringBuilder when you need a mutable string but don't need thread safety, for better performance.

Why String Immutable?

Immutability means that once an object is created, it's state cannot be changed. In the String, you cannot alter the sequence of characters it holds. Any operation that seems to modify a String will instead create a new String object and leave the original unchanged.

```
String s1 = "Hello";
String s2 = "Hello";
```

Both s_1 & s_2 will point the same "Hello" object in the String pool.

- String optimize memory usage with this functionality.
- It makes String consistent & predictability easy as we know modifications are not allowed.
- Security: database username & password, network connections, file path uses String due to it's immutability.
- Thread safety

Singleton Class

- A class that allows only one instance of itself to be created and provides a global point of access to that instance.
- It is achieved by making the constructor private and providing a static method that returns the single instance of the class

Example:

```
public class Singleton {
  private static final Singleton instance = new Singleton();
  private Singleton(){}
    public static Singleton getInstance() {
      return instance;
    }
}
```

Usage:

```
public class SingletonDemo {
public static void main(String[] args) {
Singleton singleton1 = Singleton.getInstance();
Singleton singleton2 = Singleton.getInstance();

System.out.println(Singleton1 == Singleton2);

}
```

Output:

true

Types of Interface

1 Normal Interface (Abstract Interface)

- purpose: To define a contract of behaviour for classes to implement.
- Characteristics:
- can have multiple abstract methods.
- Since Java 8 interface can have default and static methods with bodies, providing concrete implementation.
- Can be extended by other interfaces or implemented by class

Example:

```
public interface Animal {
    void eat();
    void sleep();
}
public class Dog implements Animal {
    @Override
    public ~~static~~ void eat(){
        Sout("Dog is eating");
    }

    @Override
    public void sleep() {
        Sout("Sleeping");
}
```

2- Marker Interface
- Contains no methods or fields.
- Provides metadata information to the Java runtime or compiler
- Used to tag a class as having some property or behaviour without defining any methods.

Example: Serializable , Clonable

```
Public class MyClass implements Serializable {
        // No methods to implement from Serializable
}
```

3- Function Interface
- To support functional programming by defining a single abstract method.
- They are the basis of lambda expression.
- Contains exactly one abstract method.
- Can have additional default or static methods.
- Optionally marked with @FunctionalInterface
- Used extensively with lambda expression & method reference

Example:

```
@FunctionalInterface
public interface Greeting {
    void sayHello(String name);
}
```

Usage:

```
Greeting greeting = (name) -> System.out.println("Hello, " + name);
greeting.sayHello("John");
```

__/__/114

Types of Functional Interface

1. Predicate: Represents a condition or test on a single argument, returns boolean value.
   - test() is used to call.

```
Predicate<Integer> isGreaterThan = x -> x > 10;
```

2. Function: Represents a function that takes one argument and produces a result.
   - apply() is used to call.

```
Function<Integer, String> intToStr = x -> Integer.toString(x);
```

3. Consumer: Performs an action on a single argument and does not return any result.
   - accept() is used to call.

```
Consumer<Integer> print = System.out::println;
```

4. Supplier: Supplies a value without taking any input.

```
Supplier<Double> randomValue = Math::random;
```

   - get() is used to call.

5. Runnable: no arguments & no results.

```
Runnable task = () -> sout("Task Running");
```

6. Callable: Similar to Runnable but return result.

```
Callable<Integer> task = () -> 123;
```

More Interface Types

- Single Inheritance Interface
- Nested Interface
- Annotation Interface
- Extending Interface
- Default method in Interface
- Static method in Interface

Types of Classes

1- Concrete Class
- A class that can be instantiated to create objects
- Contains complete implementations of it's methods.
- Can be used directly to create instances.

2- Abstract Class
- A class that cannot be instantiated and may contain abstract methods (methods without an implementation).
- Can contain both abstract & concrete methods.
- Used as a base class for other classes to extend and provide implementations for the abstract methods.

Example:

```
public abstract class Animal{
    abstract void sound();
    void sleep(){
        System.out.Println("Sleeping....");
    }
}

public class Cat extends Animal{
    void sound(){
        Sout("Meow...");
} }
```

3- Interface
- A reference type in java that can only contain abstract methods (before Java 8) and constants.
- A class implements an interface, inheriting the abstract methods.
- Can include default methods and static methods.

4- Enum Class
- A special class that represents a fixed set of constants (enumerated types).
- Provides a way to define a set of related constants.
- Can have, fields, methods & constructors.

Example:

```
public enum Day{
    SUNDAY, MONDAY, TUESDAY
}
Day today = Day.Monday;
```

5- Singleton Class
Already defined 3 pages prior.

6- Nested Class
- A class defined within another class.
- Can access static members of outer class - if it a Static Nested Class.
- Can access both static & instance member of the outer class if its Inner Class.

Example:

```
public class OuterClass{
   private int field = 10;
   class InnerClass {
      void display() {
         Sout("Data" + field);
      }
   }
}
```

```
OuterClass.InnerClass inner =
                    new OuterClass().new InnerClass();

inner.display();
```

7- Local Class

- A class defined within a method.
- Can access local variables and parameters of the method.

Example:

```
public void myMethod(){
      class LocalClass{
         void display(){
          System.out.println("Inside local class");
         }
   }
   LocalClass local = new LocalClass();
   local.display();
}
```

8- Anonymous Class
- A class without a name that is defined and instantiated in a single statement.
- Useful for implementing interface or abstract classes on the fly.

Example:

```
Animal animal = new Animal(){
    void sound(){
        Sout("Roar!");
    }
};

animal.sound();
```

9 Abstract Singleton Class
- Class that combines the features of an abstract and singleton class.
- Cannot be instantiated directly.
- Contains abstract method that muste be implemented by subclasses.

Java 8 Features

1- Lambda Expression

- A concise way to represent a functional interface (an interface with a single abstract method) using an expression.
- Simplifies the syntax of code that uses functional interfaces, especially for evant handling, threading & collection processing.

Example:

Traditional way

```
Runnable r1 = new Runnable () {
    @Override
    public static void run(){
        System.out.Println("Running---");
    }
};
```

Using Lambda Expression

```
Runnable r2 = () -> System.out.println("Running");
```

2- Functional Interface

See page: 113

3- Default Methods

- Methods in Interface that have a body, allowing you to provide a default implementation.
- Allows for backward compatibility with existing interfaces while enabling new functionality without breaking existing

implementations.

Example:

```
public Interface MyInterface {
    default void defaultMethod() {
        System.out.Println("Default");
    }
}
```

4- Method Reference

- A shorthand notation of a lambda expression to call a method.
- Make the code more readable and expressive by reffering to existing methods or constructors without executing them.

Example:

```
List<String> names = Arrays.asList("Vishal", "Nikhil");
names.forEach(System.out::println); // reffering to println method
```

5- Optional Class

- A container object which may or may not contain a non-null value.
- Helps avoiding NullpointerException.

6- New Date & Time API

A new package java.time.

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plusDays(1);
```

7. Type Annotations
- An enhancement that allows annotations to be applied to any use of a type, not just class declarations.
- Enables better compile-time checking and improved tooling support.

Example:
```
@NonNull String str; // using annotation on type
```

8. Stream API

The Stream API in Java was introduced in Java 8 as a powerful way to work with collections (like list or set) in a more readable, flexible, and efficient way. It allows you to process data in a functional programming style by chaining methods together to transform or filter data without having to write a lot of loops and conditionals.

What is Stream?

A Stream is a sequence of data elements that can be processed in bulk. Think of it as a pipeline through which your data flows, allowing you to perform operations on each piece of data in a very concise way.

Example: Imagine a list of numbers. A Stream can be used to filter out only the even numbers or sum them up without using loops.

Key Concepts of Stream API

1- Streams don't store data: A stream is not a data structure like a list or an array. It doesn't store the data itself but operates on the data from existing collections.

2- Streams are Lazy: Operations on streams are not executed until a terminal operation is called. This means the stream can optimize the process of working with data, like only processing the parts that are necessary.

3- Functional Style: The stream API lets you write code in a declarative way, means you can focus on what you want to do, not how to do it. For example, you can tell the stream to filter out odd numbers without worrying about how to loop over the collection.

How Stream Works

Streams consits of three types of operations:

1- Source: This is where the stream gets its data from (like a list, set or array).

Example: Creating a Stream from a list:

```
List<Integer> numbers = Array.asList(1,2,3,4,5);
Stream<Integer> numStream = numbers.stream();
```

2- Intermediate Operations: These are the operations that perform or filter the data. You can chain multiple intermediate operations together, but they are not executed until a terminal operation is called.

Common intermediate operations:

- filter(): Select elements that meets a condition.
- map(): Transform each element into something else.
- sorted(): Sorts the elements.
- distinct(): Removes duplicates.

Example: Filtering even numbers from a list:

```
Stream<Integer> evenNum = numbers.stream().filter(n->n%2 ==0);
```

3- Terminal operations: These are the operations that trigger the execution of the stream. Once a terminal operation is called, the stream is "used up" and cannot be used again.

Common terminal operations:

- forEach(): Applies an action on each element.
- Collect(): Gathers the result into a collection like a set or list.
- reduce(): Reduces the stream to a single result like summing up all the elements.
- Count(): Counts the number of elements.

Example: Collecting filtered numbers into a list.

```
List<Integer> evenList = numbers.stream().filter(n->n%2==0)
                         .collect(Collectors.toList());
```

Stream Example

Let's say you have a list of numbers and you want to:

1- Filter out only the even numbers.
2- Multiply each of those even numbers by 2.
3- Collect the result into a new list.

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);

List<Integer> processedNumbers = numbers.stream()
        .filter(n->n%2==0)
        .map(n->n*2)
        .collect(Collectors.toList());

System.out.println(Processed Numbers); // [4,8,12]
```

Types of Streams

- Sequential Streams: These streams process data one element at a time, in a single thread. This is the default type of stream.
- Parallel Stream: These streams divide the data into chunks and process them in parallel using multiple threads, which can make processing large

datasets faster.

Parallel stream example:

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5);
numbers.parallelStream().forEach(n -> sout(n));
```

Parallel stream is useful for heavy computations but should be used cautiously because they involve multithreading, which can introduce complexity.

Common Stream Operations

1- Filtering - Exclude or include elements based on a condition.

Example:

```
Stream<Integer> evenNum = numbers.stream().filter(n -> n%2==0);
```

2- Mapping - Transform elements, such as converting them to another type.

Example:

```
Stream<String> stringNum = numbers.stream().map(String::valueOf);
```

3- Sorting : Sort elements in natural or custom order.

Example:

```
Stream<Integer> sortedNum = number.stream().sorted();
```

4- Reducing : Combining elements into a single value, such as sum.

Example:

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

↑ method referencing

5. Collecting: Gather the elements into a collection or other data structure.

Example:

```
List<Integer> evenNum = numbers.stream().filter(n -> n%2==0)
                        .collect(Collectors.toList());
```

Difference Between Map & Foreach

Aspect	ForEach()	map()
1- Operation type	Terminal operation (consumes the stream)	Intermediate Stream (transform the Stream)
2- Purpose	Perform an action on each element (e.g., print)	Transform each element (e.g, change type or value)
3- Return type	void (no return value)	Returns a new stream with transformed elements
4- Common use cases	Printing, logging, saving data, side-effects	Converting data types, transforming values, modifying content.
5- Stream Behaviour	Consume the stream, making it unusable afterward	Allows further processing with other stream operations.

Example Combining map() & forcach()

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.Stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

Aggregation, Association and Composition

In Java, aggregation, association and composition are concepts related to object-oriented programming that describe relationships between classes and objects.

1. Association:

Association is a relationship between two separate classes where both classes have their own lifecycle, and no class owns the other.

Example:

A class student can have association with a class teacher. A student is taught by a teacher, but neither the student nor the teacher is dependent on each other for their lifecycle.

```
class Student{
    String name;
}

class Teacher{
    String name;
}

class School{
  public static void main(String[] args){
        Student student = new Student();
        Teacher teacher = new Teacher();
      // Association: Student is taught by teacher
}}
```

2- Aggregation:
Aggregation is a special form of association where one class (whole) contains other class objects (parts). The part can exist independently of the whole.

Example:
A class Department can contain a class Professor, but if the department is removed, the professor can still exists elsewhere.

```
class Professor {
    String name;
}

class Department {
    List<Professor> professors;

    Department (List<Professor> professors) {
        this.professors = professors;
    }
}
class University {
    public static void main(String[] args) {
        Professor p1 = new Professor();
        Professor p2 = new Professor();
    List<Professor> profList = Arrays.asList(p1, p2);

    Department department = new Department(profList);
// Aggregation: Professors belongs to department but can exist independently
}
}
```

3. Composition:

Composition is a strong form of aggregation where the containe objects (parts) cannot exist independently of the containing class (whole). If the whole object is destroyed, the parts are also destroyed.

Example:

A class House contains a class Room. A room cannot exists without a house. If the house is destroyed, the room is also destroyed.

```
class Room {
    String name;
}
class House {
    Private List<Rooms> rooms;

    House () {
    rooms = new ArrayList<>();
    rooms.add (new Room());
    rooms.add (new Room());
    }
}

class City{
    public static void main (String[] args){
        House house = new House();
    // Composition: Rooms cannot exists without the House.
    }
}
```

Summary:

- Association: A loose relationship (a student and a teacher)
- Aggregation: A "has-a" relationship where parts can exists independently (a department has professors).
- Composition: A stronger "has-a" relationship where parts cannot exists independently (a house has rooms)

Loose Coupling and Tight Coupling

1. Loose Coupling

- Loose coupling means classes or components have minimal dependencies on each other.
- Change in one class have little or no impact on other class.
- Easier to maintain and extend.
- Increases code flexibility and reusability.

Example:
Car is loosely coupled with Engine because it interacts with the Engine through an interface. This allows changing the engine type without modifying the Car class.

```
interface Engine {
    void start();
}
```

```
class PetrolEngine implements Engine{
    public void start(){
        System.out.println("Petrol engine started");
    }
}

class DieselEngine ;implements Engine{
    public void start(){
    System.out.println("Diesel engine started");
    }
}
class Car{
    private Engine engine;
    public Car(Engine engine){
        this.engine = engine;
    }
    public void startCar(){
        engine.start();
    }
}

class Main{
    public static void main(String[] args){
        Engine petrolEngine = new PetrolEngine();
        Car car = new Car(petrolEngine); // passing Engine at Runtime

        car.startCar(); // Loosely coupled with Engine
    }
}
```

2- Tight Coupling:

- Tight coupling means that classes or components are highly dependent on each other. One class relies heavily on implementation details of another class.
- Difficult to modify and maintain.
- Change in one class requires change in other tightly coupled classes.
- Reduces flexibility and reusability.

Example:

Car is tightly coupled with PetrolEngine. If we want to use a DieselEngine, we need to modify the Car class.

```
class PetrolEngine {
    public void Start() {
        System.out.println("Petrol Engine Started");
    }
}
class Car {
    private PetrolEngine engine; // Tightly coupled to PetrolEngine

    public Car() {
        this.engine = new PetrolEngine(); // Engine is hardcoded
    }
    public void startCar() {
        engine.start();
    }
}
```

```
class Main{
    public static void main(String[] args){
        Car car = new Car();
        car.startCar(); // Tightly coupled with PetrolEngine
    }
}
```

Relation to Aggregation, Composition and Association

- Loose Coupling: Aggregation and Association promote loose coupling because the classes can exist independently of each other. They allow replacing or changing components without affecting others.
- Tight Coupling: ~~Coupling~~ Composition can sometimes lead to tighter coupling because the lifecycle of one class is directly tied to the other. However, proper use of abstraction (like interfaces) can mitigate this.

Java 12 Updates (March 2019)

1- Switch Expressions (Preview):

- Switch statements can return values, making the code more readable.

Example:

```
int result = switch (day) {
    case Monday, Tuesday, Sunday -> 6;
    case Thursday -> 7;
    default -> 0;
};
```

2- Compact Number Formatting

- Formats number in a shorter, localized way
e.g. "1.5K" instead of 1500

Example:

```
NumberFormat fmt = NumberFormat.getCompactNumberInstance();
Sout(fmt.Format(1500)); // 1.5K
```

3- New Garbage Collector: Shenandoah

- A new low-pause-time garbage collector designed to reduce application latency.
- Useful for applications that need predictable response times.

Java 17 Updates (September 2021) · LTS

1- Pattern matching for instanceof:
- Simplifies the instanceof checks by casting and checking in one step.
- No need for a separate cast after instanceof.

Example:

```
if (obj instanceof String str) {
    Sout (str.toUpperCase());
}
```

2- Sealed Classes
- Limits which classes can extend or implements a class or interface.

Example:

```
public abstract sealed class Shape permits Circle, Square {

}
```

Only circle & Square can extend Shape.

3- Records (finalized):
- A compact way to create data classes with automatic getters, equals(), hashCode(), and toString().

Example:

```
public record Person (String name, int age) { }
```

Java 21 Updates (September 2023)

1- Virtual Threads (Project Loom)

- New way to handle concurrency with lightweight threads.
- Much faster than traditional Java threads.

Example:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()){
executor.submit(()-> Sout("Hello from a virtual thread"));
}
```

2- Sequenced collections:

- A new interface that provides guaranteed order for collections like lists and sets ensuring elements are returned in a predictable order.

Example:

```
SequencedCollection<String> names = new LinkedHashSet<>();
names.add("John");
names.add("Jane");
Sout(names);    // order maintained
```

3 Record Pattern

```
record Point(int x, int y) {}
if (obj instanceof Point(int x, int y)) {
    Sout("x:" + x + ", y:" + y);
}
```

4- String Templates:

- Introduces a way to embed expressions directly into strings, simplifying string concatenation and formatting.

Example:

```
String name = "Alice";
String message = STR."Hello, \{name}!";
```

JAR, WAR and EAR Files

1- JAR (Java ARchive) file

- A JAR file is a package that contains compiled Java classes and other resources (like images or text files). It bundles everything into a one file.
- JAR files are used for standalone, applications or libraries that can be shared with other projects.
- If you write a Java program and compile it, you can package it into a JAR file so its easier to distribute.
- You can run it using: java -jar myapp.jar

2- WAR (Web Application Archive)

- A WAR file packages web applications. It contains things like java class files, HTML pages, JSP files, Javascript and more.
- WAR files is used to deploy web applications on servers like Tomcat or Jetty.
- If you build a website (Java-based), you package

it as a WAR file to upload to a server.
- Used for web applications
- It contains a special folder (WEB-INF) with web-related configurations.

3- EAR (Enterprise Archive) file
- An EAR file is used to package large enterprise applications. It can contain both JAR file (for backend logic) and WAR files (for web components).
- EAR file is used for big applications that need both web features and complex backend systems, like banking or large company software.
- If you create a big application with web pages and server-side processing, you'll use an EAR file to bundle everything together.
- Contains JARs & WARs to handle different parts of the system.

Multithreading

- Multithreading in java allows concurrent execution of two or more threads, enabling better utilization of CPU and making programs more efficient.
- Threads can be created in two main ways: by extending the Thread class or by implementing the Runnable interface.

Key concepts of Multithreading

1- Thread: A thread is a lightweight process and is the smallest unit of a program that can be executed independently. Thread class is used to create and manipulate threads.

2- Runnable: Runnable is an interface that defines a single method, run(). It represents a task that a thread can execute. Classes can implement Runnable interface to create a thread without extending the Thread class.

3- Creating a Thread:

- Extending Thread class:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // starts the thread
    }
}
```

- Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run(){
        System.out.println("Thread is running");
    }
}

public class Test {
    public static void main (String [] args) {
        Thread t = new Thread (new MyRunnable());
        t.start();
    }
}
```

4- What is Multitasking?

- Ability of operating system to perform multiple tasks or processes at same time.
- Two types:

1- Process-based Multitasking

- Independent programs runs simultaneously.
- A browser, text editor, music player running at same time.

2- Thread-based Multitasking

- Single process is divided in to multiple threads, and these threads run concurrently.
- A browser can load web pages in one thread while downloading a file in another.

5 Thread Lifecycle

- New: A thread is created but not yet started.
- Runnable: The thread is ready to run and waiting for CPU allocation.
- Running: The thread is currently executing.
- Blocked/Waiting: The thread is waiting for some condition to be met (e.g., waiting for I/O).
- Terminated: The thread has finished execution.

6. Thread Methods

- start(): Starts the thread and invokes the run() method.
- sleep(long millis): Causes the thread to pause execution for the specified time.
- join(): Waits for the thread to die (finishes execution).
- yield(): Temporarily pauses the currently executing thread and allows other threads to execute.
- interrupt(): Interrupts a thread, commonly used to signal a thread to stop.
- isAlive(): checks if thread is still running.

7. Synchronization

When multiple threads access shared resources, data inconsistency can occur. Synchronization ensures that only one thread accesses a resource at a time.

- Synchronized method: synchronized to prevent simultaneous access

```
public synchronized void synchronizedMethod(){
    // only one thread can execute this method at a time
}
```

- Synchronized block: Synchronizes a specific block of code:

```
synchronized (this) {
    // critical section
}
```

8 What is a Deadlock

- Deadlock is a situation where two or more threads are blocked forever, waiting for each other to release the locks. Proper synchronization strategies can avoid deadlocks.

9 Concurrency Utilities

Java provides a java.util.concurrent package with utilities for advanced thread management.

- Executors: Used to manage a pool of threads:

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(new MyRunnable());
executor.shutdown.
```

- Callable and Future: These allow a thread to return a result or throw an Exception.

- Locks: Provides more control over synchronization compared to synchronized keyword.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
```

10- Thread Communication: Threads can communicate via methods like wait(), notify() and notifyAll(). These are used when threads need to cooperate on the same task.

- wait(): Makes the thread wait until another thread calls notify() or notifyAll().
- notify(): Wakes up a single thread waiting on the object's monitor.
- notifyAll(): Wakes up all threads waiting on the objects monitor.

11- Thread Synchronization Example

```
class BankAccount {
    private int balance = 1000;

    public synchronized void withdraw(int amount){
      if(balance >= amount){
    Sout(Thread.currentThread().getName() + "is withdrawing " +
        amount);
    balance -= amount;
    Sout("New balance: " + balance);
    }else{

      Sout("Insufficient balance");
    }
}}

class AccountHolder implements Runnable{
  private BankAccount account;
  public AccountHolder(BankAccount account){
    this.account = account;
  }
```

```
@Override
public void run(){
    for(int i=0; i<5; i++){
        account.withdraw(300);
    }
  }
}

public Class BankApp {

  public static void main(String[] args){

    BankAccount account = new BankAccount();
    AccountHolder holder = new AccountHolder(account);

   Thread t1 = new Thread(holder);
   Thread t2 = new Thread(holder);

    t1.start();
    t2.start();
  }
}
```

Both threads try to withdraw money from the same account. The synchronized keyword ensures that only one thread can withdraw at a time, preventing race conditions.

Output :
Thread-0 is withdrawing 300
New Balance : 700
Thread-0 is withdrawing 300
New Balance : 400
Thread-0 is withdrawing 300
New Balance : 100
Insufficient balance
Insufficient balance
Insufficient balance
Insufficient balance
Insufficient balance
Insufficient balance
Insufficient balance

Concurrency :
Concurrency in Java is the ability to execute multiple task simultaneously, improving performance and resource utilization. It involves threads, which are lightweight units of execution within a process. Java allows concurrency through multithreading, where multiple threads can run concurrently either on a single core or multiple cores.

Key concepts in Java concurrency.
- Multithreading : Running multiple threads simultaneously.
- Synchronization : Preventing thread interface and ensuring consistent data when multiple threads shared resources.

- Concurrency Challenge: Issues like race conditions, deadlocks and memory consistency errors.
- Java Concurrency API: Provides tools like ExecutorService, locks, atomic variables and thread safe collections such as ConcurrentHashMap to manage concurrency more effectly.
- Parallelism: True simultaneous task execution across multiple cores.

Concurrency can be managed using tools such as the ExecutorService for thread pooling, and synchronization techniques like synchronized blocks and locks to maintain thread safety.