JAVA NOTES

> 💡 Imperative programming is ***a software development paradigm where functions are implicitly coded in every step required to solve a problem.*** In imperative programming, every operation is coded and the code itself specifies how the problem is to be solved, which means that pre-coded models are not called on.

> 💡 Functional programming is ***a programming paradigm in which we try to bind everything in pure mathematical functions style***. It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”.

▼ JAVA Datatypes

- 3.14, 7.123, ... are `double` by default.
- $, _ are allowed in variable names.

Primitive Types:

Datatype	size (in bytes) (1 byte = 8 bits)
byte	1
short	2
int	4
long	8
float	4
double	8
boolean	1
char	2

Datatype Conversion:

1. Explicit `int a = ( int ) 5.6;`
2. Implicit `double a = 5.6`

▼ Naming Convention

- Interface : Adjective
  - ex : `Runnable`, `Readable`, `Remote`
- Method : Verb
  - ex : `read()`, `write()`
- Variable : $, _ allowed

- ex : `readBook = true` , `addSum = 10`

- Constant : ALL CAPS
  - ex : `PI = 3.14` , `MONTH = “january”`
- Class : Noun
  - ex : `Student` , `Animal`

▼ Operators

TYPE	PRECEDENCE
Unary	`++` `- -`
Arithmetic	`*` `-` `/` `%` `+`
Logical	`&&` `||`
Relational	`< >` `<=` `>=` `==` `!=`
Assignment	`=` `+=` `*=` `&=` `^=` `|=` `>>=` `<<<=`
Shift	`<<` `>>` `>>>`
Bitwise	`&` `^` `|`
Ternary	`? :`

▼ Jumping Statements & var Args

Jumping Statements

- Break
- Continue

Var Args with For-Each

```
public void fun(int... n) {
        int sum = 0;
        for (int i: n) {
            sum += i;
        }
    }
```

▼ Constructor & Static Block

Contructor

- executes when object creation.
- Used to allocate Memory.
- Used to initialise values.
- `this (1, 2)` calls constructor with two params.

Static Block

- It executes before the Constructor.
- Runs only once no matter how many times object created.
- Used to initialise static variables.

```
static int a;
    static {
        a=20;
    }
```

Static Keyword

- Executes when we load a class.
- References **are stored** in Stack memory.
- Objects **are** stored in Heap memory.
- Static stores in ClassLoader memory
- Static methods/attributes can be accessed without creating an object of a class.
- static methods only accept static variables.

▼ Inner Class

- Inner classes are 3 types.
- We know a class cannot be made private, but if we have the class as a member of other class, then the inner class can be made private.

Member Class

```
class Outer {
    //code
    class Inner {
      void show() {}
```

Static Class

```
class Outer {
    //code
    static class Inner {
      void show() {}
```

Anonymous Class

```
class Outer {
    //code
    Outer outer = new Outer() {
        void show() {//code}
```

```
    }
}
```

```
    }
}
```

```
    };
}
```

▼ String & Character Functions

String	Character
toUpperCase ()	isLetter ()
toLowerCase ()	isDigit ()
charAt ()	isSpaceChar ()
contains ()	isLetterOrDigit ()
indexOf ()	isUpperCase ()
lastIndexOf ()	isLowerCase ()
subString ()	
equalsIgnoreCase ()	
replace ()	
trim ()	
toCharArray ()	
isEmpty ()	
concat ()	

▼ OOPS

Following are the concepts of OOPS (Object Oriented Programming)

- OOPS is a type of programming that is based on objects rather than just functions.
- *Class and Object are also OOPS, but following are the pillars.*

▼ Inheritance

One class Inherits the features (fields & methods) of another class.

- Parent `IS-A`
- Referenced class `HAS-A`
- Multiple inheritance not supported (Ambiguity Problem).

```
class A {
    void show() {}
}

class B {
    void show() {}
}
```

```
class C extends A, B {
    public static void main(String[] args) {
        C c = new C();
        c.show();
    }
}
```

- Use @Override annotation to override methods in sub-class (eliminates logical error while overriding functions).
- We can call super class methods by `super.method();`

```
public void show() {
        super.show()
    }
```

▼ Abstraction

focussing only essential details by hiding the working complexity of the system

- Abstraction can be acheived using either Abstract Classes or Interfaces.
  -

Abstraction with Abstract Class:

Abstraction with Interface:

```
class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.getName();
        cat.getName();
    }
}

abstract class Animal {
    abstract void getName();
}

class Dog extends Animal {
    @Override
    void getName() {
        System.out.println("Dog");
    }
}

class Cat extends Animal {
    @Override
    void getName() {
        System.out.println("Cat");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.getName();
        cat.getName();
    }
}

interface Animal {
    void getName();
}

class Dog implements Animal {
    @Override
    public void getName() {
        System.out.println("Dog");
    }
}

class Cat implements Animal {
    @Override
    public void getName() {
        System.out.println("Cat");
    }
}
```

▼ Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.This allows us to perform a single action in different ways.

Types :

Compile - Time Polymorphism (static binding):

- Also called as static polymorphism.
- Achieved by method overloading.

```
class Main {
    public static void main(String[] args) {
        System.out.println(A.sum(1, 2));
        System.out.println(A.sum(1, 2, 3));
    }
}

class A {
    public static int sum(int a, int b) {
        return a + b;
    }

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

Run - Time Polymorphism (Dyamic Method Dispatch) (dynamic binding):

- Call to an overriden method is resolved in run - time.
- Depends on the type of object, which method to call.
- Calls methods dynamically during run-time is called Dynamic Method Dispatch

```
class Main {
    public static void main(String[] args) {
        A a = new A();
        A b = new B();

        a.show();
        b.show();
    }
}

class A {
    public void show() {
        System.out.println("A");
    }
}

class B extends A {
    public void show() {
        System.out.println("B");
    }
}
```

▼ Encapsulation

Binding data with methods is called Encapsulation.

- Makes data safe with,
  - **Access Specifiers**
  - **Getters**

- **Setters.**

- We can maintain log file easily, since access to variables is through getters & setters (by printing text).

```
class Main {
    public static void main(String[] args) {
        Car mustang = new Car();

        mustang.setBrand("ford");
        mustang.setPrice(5000000);

        System.out.println(mustang.getBrand());       // ford
        System.out.println(mustang.getPrice());       // 5000000
    }
}

class Car {
    private String brand;                          // limiting access to variables by making it private
    private int price;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {          // setting the value of variable through setter
        this.brand = brand;
    }

    public int getPrice() {                       // getting the value of variable through getter
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }
}
```

▼ Wrapper Classes

- According to **OOPS**, everything should be objects.
- int, float,….. are primitive data types. (so, java is not 100% OOPS Language)
- Primitives are faster than wrapper classes.
- why using wrapper —> Some frameworks only supports wrapper classes.

Manual :

- Putting primitive value inside object called **Boxing**

  `int i = 5;`

  `Integer j = new Integer(i);` //boxing (or) wrapping.

- Taking value from object called Unb**oxing**

  `int k= j.intValue();` //unboxing (or) unwrapping.

Auto :

- Assigning primitive values directly to object called Auto **Boxing**

  `Integer j = 7;` //auto boxing

- Assigning object value directly to primitive type called Auto Unb**oxing**

  `int k= j;` //auto unboxing

▼ Abstract Classes & Methods

- We can't create object for abstract class.
- We can declare/define methods in abstract class.
  - i.e ; `abstract void fun();` //method declared not defined yet
- Abstract Methods should be defined in child class (**child class is also known as concrete class**)

```
abstract class Vechicle {
    abstract void brand();
}

class Ford extends Vechicle {
```

```
    @Override
    void brand() {
        System.out.println("Ford");
    }
}
```

- Abstract classes are used to avoid code duplication.
  - i.e ; common fields, methods are placed in abstract class & can be accessed by inheriting abstract class.

```
abstract class Person {
    private String name;              //name, id are common fields for both teacher & student
    private int id;
}

class Teacher extends Person {
        //code
}

class Student extends Person {
        //code
}
```

- If Person isn't abstract, then one can create object for person, which creates confusion (**Student/Teacher, which object should create**).

▼ Interface

▼ Interface Intro

- It is similar to a To - Do list.
- Similar to class, but methods declared are abstract by default.
- Regular methods can't be created.
- Variables declared are public static final by default.
- To inherit use implements.
- Contructor can't be created.
- Used to achieve Generalisation (**converting sub class type to super class**)
- Interface is defined as follows :

```
class Main {
    public static void main(String[] args) {
        Vehicle ford = new Ford();                //instance is of vehicle & object is of ford
    }
}

interface Vehicle {
    void showBrand();
}

class Ford implements Vehicle {

    @Override
    public void showBrand() {
        System.out.println("ford");
    }
}
```

- Static methods are allowed in interface after JAVA 1.8 / JAVA 8

```
class Main {
    public static void main(String[] args) {
        A.fun();             //since, method is static object not needed
    }
}

interface A {
    static void fun() {
```

```
        System.out.println("A");
    }
}
```

- Method defining is possible after JAVA 1.8 / JAVA 8

```
class Main {
    public static void main(String[] args) {
        Vehicle thar = new Thar();
        thar.fun();                             //in Vehicle
    }
}

interface Vehicle {
    default void fun() {                        //method definition in interface using default keyword
        System.out.println("in Vehicle");
    }
}

class Thar implements Vehicle {
    //no methods
}
```

💡 **Object of interface can not be created**

▼ Interface Types

There are 3 types of Interface

1. Normal Interface

- It has more than one method.

```
interface Vehicle {

    void showBrand();

    void showPrice();

    void showSpeed();
}
```

2. Functional Interface

- It has only one method.
- Also known as Single Abstract Method Interface.
- Having both default & a single method in interface is still a functional interface.
- It supports Lambda - Expression.

▼ `Lambda - Expression code.`

```
class Main {
    public static void main(String[] args) {
        Vehicle ford = () -> {
            System.out.println("ford");
        };
        ford.showBrand();
    }
}

interface Vehicle {
    void showBrand();
}
```

▼ `Lambda - Expression with default method code.`

```
class Main {
    public static void main(String[] args) {
        Vehicle ford = () -> {
            System.out.println("ford");
        };
        ford.showBrand();
        ford.fun();
    }
}

interface Vehicle {
    void showBrand();

    default void fun() {
```

3. Marker Interface

- It has no methods.
- Also known as Tagging Interface.
- Used for getting meta-data or for tagging purpose, classify code.

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

    }
}

interface Vehicle {
    //no methods
}
```

```
        System.out.println("in interface");
    }
}

          //output: ford
          //        in interface
```

▼ Anonymous class with Interface

💡 If we need the class/interface for single use then we can create anonymous class/interface.

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

        Vehicle ford = new Vehicle() {                  //anonymous class
            @Override
            public void showBrand() {
                System.out.println("ford");
            }
        };

        ford.showBrand();                               // displays "ford"
    }
}

interface Vehicle {
    void showBrand();
}
```

▼ Multiple Inheritance with Interface

```
interface A {
    default void fun() {
        System.out.println("A");
    }
}
```

```
interface B {
    default void fun() {
        System.out.println("B");
    }
}
```

```
class C implements A, B {

    @Override
    public void fun() {
        A.super.fun();           // o/p: A
        B.super.fun();           // o/p: B
    }
}
```

▼ Interface vs Abstract class

When to use interface / abstract class?

💡 if we are defining characteristics of an object type (specifying what an object is) we should go with abstract classes

if we are defining capabilities that we promise to provide (***what the object can do)*** we should go with interface

💡 **INTERFACE : What the object can do?** **(i.e ; 'A' is capable of [doing] _____)**

💡 **ABSTRACT : What an object is ___?** **(i.e ; 'A' is a 'B')** "Dog is an Animal", "Lamborghini is a Car"

▼ Final keyword

Final keyword can be used with class, methods & variables.

Final with Variables	Final with Class	Final with Methods
Value once assigned can't be changed.	Class can't be extended.	Method won't be overrided in sub classes.
Becomes constant like `PI = 3.14`.		Method overloading is possible.
If final variables, not initialised then it can be done only in constructor.		

▼ Enums

Enums is a class with static members.

- It defines collection of constants.

```
class Main {
    public static void main(String[] args) {
        Days day4 = Days.WEDNESDAY;
        System.out.println(day4);
    }
}

enum Days {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

▼ Packages & Access Modifiers

Package format should be "**com.telusko**"

- To import package ——> **import java.io.*;**
- To create package ——> **package com.telusko;**

Access Modifiers

Modifier	Access
private	specific class
public	any class / package
protected	subsiding class (to extended class)
default	specific package
final	-
abstract	-

Encapsulation is achieved through using **Access Modifiers.**

▼ Exception Handling

To achieve Exception Handling "Throwable" class is used.

Exception Types:

1. **Checked Exception** : When compiler knows about Exception is called Checked Exception, and alerts the programmer to catch the exception.
2. **Unchecked Exception** : When compiler doesn't know about the Exception is called Unchecked Exception. (runtime exceptions)

Try - Catch Block : **Multiple Catch Block :** **Try - Catch - Finally Block :**

- Doubtful code should be put in try block.
- Catch the Exception in Catch block.
- Catch block executes only if there is a exception in try block.

```
try {
   int a = 10 / 0;
}
catch (Exception e){
  System.out.println("fix the code");
}
```

- Multiple Exceptions can be catch using multiple catch block.
- If the Exception has Parent - Child relationship then it must be sorted in catch block.
  - (Exception - ClassNotFoundException)
- It can acheived in 2 ways,

▼ **Syntax 1:**

```
try {
   int a = 10 / 0;
}
catch (ArithmeticException e) {
  System.out.println("can't be divided by zero");
}
catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("check array size");
}
catch (Exception e){
  System.out.println("fix the code");
}
```

▼ **Syntax 2:**

```
try {
  int a = 10 / 0;
}
catch (ArithmeticException |
       ArrayIndexOutOfBoundsException |
       NullPointerException e) {
  System.out.println("can't be divided by zero");
}
```

- It has 3 Blocks,
  - Try
  - Catch
  - Finally
- Whether there is an exception in try block or not, finally block will executes at the end.
- Used to close the resources after using them.

```
try {
   int a = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("can't be divided
} finally {
   System.out.println("finally block");
}
```

- Resources should be closed after using them or else memory will be occupied.

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.*in*));
try{
   //code
}catch(Exception e){
   //code
}finally{
   br.close();
}
```

- All classes from `io` are Resources.
  - `BufferedReader`, `InputStreamReader`, ...
- **Try with Resource: (Automatically closes the resource without mentioning in finally block)**

```
try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {       //Automatically closes the resource 'br
            //code
}
catch(Exception e) {
          //code
}
```

User Defined Exception :

```
class Main {
    public static void main(String[] args) {
        try {
            if (condtition)
                throw new OwnException("user defined exception");
        } catch (OwnException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
class OwnException extends Exception {
    OwnException(String s) {
        super(s);
    }
}
```

▼ Multi Threading

Executing more than one Thread at the same time parallely is known as Multi Threading.

▼ Multi Threading Intro

- Thread is a unit of a process.
- Every Program has atleast one Thread.
  - `i.e; main Thread`
- To use Thread for Normal class, Thread class is used.

```
class B extends Thread{
    //code
}
```

- To use Thread for Base/Child class Runnable interface is used.

```
class A {
    //code
}

class B extends A implements Runnable {
    // implement run()
}
```

Thread in java is **light - weight process**, requires only **few resources to create and use**.

▼ Thread with Normal Class

```
class A extends Thread {
    public void run() {
        for (int i = 1; i <= 10000; i++)
            System.out.println(i);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        A a = new A();
        a.start();
    }
}
```

▼ Thread with Base/Child Class

```
class A implements Runnable {
    public void run() {
        for (int i = 1; i <= 10000; i++)
            System.out.println(i);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Runnable r = new A();
        Thread t = new Thread(r);
        t.start();
    }
}
```

```
class Main {
    public static void main(String[] args) throws Interrupte
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 10000; i++)
                System.out.println(i);
```

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

▼ Thread Priority

- Threads are executed by scheduler based on priority.
  - default → `NORM_PRIORITY = 5`
  - MinP → `MIN_PRIOTIRTY = 1`
  - MaxP → `MAX_PRIOTIRTY = 10`
  - Range → **1 - 10**
- To set Priority use method,
  - `new Thread().setPriority(Thread.MAX_PRIORITY);`

▼ Methods in Thread

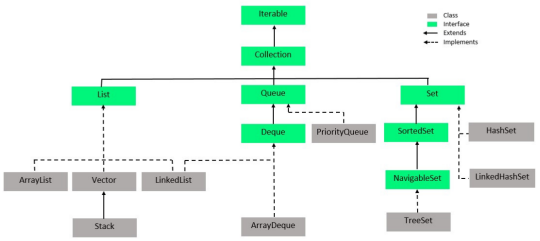
Method	Use
Thread.sleep(ms);	Suspend Thread for some time (1s = 1000 ms).
Thread.currentThread();	It refers to the current Thread.
new Thread().join();	Main Thread waits for this Thread to complete.
new Thread().isAlive();	Checks whether the Thread is alive or not. `Thread t = new Thread() {}; t.isAlive(); ---> false t.start(); t.isAlive(); ---> true t.join(); t.isAlive(); --->` false
new Thread().setName(name);	Sets name for the Thread.
new Thread().getName();	Returns name of the Thread.

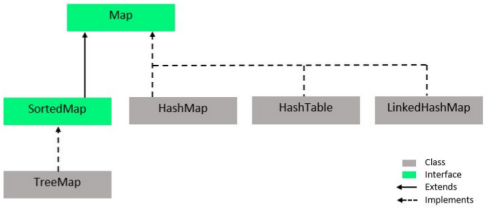
▼ Thread Safety

```
class Main {
    public static void main(String[] args) throws InterruptedException {
        Runnable r1 = new A();
        Runnable r2 = new A();

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(A.i);
    }
}

class A implements Runnable {
    static AtomicInteger i = new AtomicInteger(0);

    public void run() {
        for (int i = 1; i <= 10000; i++)
            count();
    }

    public void count() {
        i.incrementAndGet();
    }
}
```

```
class Main {
    public static void main(String[] args) throws Interrupte
        A a = new A();
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 10000; i++)
                a.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 10000; i++)
                a.increment();
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(A.count);
    }
}

class A extends Thread {
    static int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

▼ Collection & generics





- Colection is an interface.
- Collections is a class.

▼ List

- Works with index numbers.
- Duplicate values allowed.
- List is mutable.

Create List :

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

Add values :

```
list.add(value);
```

Print values :

Values can be printed in 2 ways,

```
for(Integer i : list){
    System.out.println(i);
}
```

```
Iterator i=list.getIterator();
while(i.hasNext()){
    System.out.println(i.next());
}
```

Methods in List :

Method	Use
list.add(obj);	adds object
list.add(pos, obj);	adds object at position
list.remove(obj);	remove object from list
list.get(pos)	returns object at position
list.size()	returns size of the list
list.contains(obj)	checks object present or not
list.clear()	clears the list
list.addAll(list)	adds list to the current list
list.toArray()	converts list into array
list.indexOf(obj)	returns position of the object

Sorting based on Last Digit :

It can be achieved by using comparator

```
List<Integer> i = new ArrayList<>();
        i.add(23);
        i.add(44);
        i.add(35);
        i.add(81);
        Collections.sort(i, (k, j) -> k % 10 > j % 10 ? 1 : -1);
        System.out.println(i);
```

```
List<Integer> i = new ArrayList<>();
        i.add(23);
        i.add(44);
        i.add(35);
        i.add(81);
        Comparator<Integer> c = new Comparator<Inte
            @Override
            public int compare(Integer k, Integer j
                if (k % 10 > j % 20)
```

```
                    return 1;
                else
                    return -1;
            }
        };
        Collections.sort(i, c);
        System.out.println(i);
```

- To sort the Objects (i.e; Student, Dog,……) implement the class with Comparable Interface.

```
class Main {
    public static void main(String[] args) throws InterruptedException {
        List<Student> list = new ArrayList<>();
        list.add(new Student(4));
        list.add(new Student(3));
        list.add(new Student(8));
        list.add(new Student(1));
        Collections.sort(list);
        for (Student s : list)
            System.out.println(s.id);
    }
}

class Student implements Comparable<Student> {
    int id;

    public Student(int id) {
        this.id = id;
    }

    @Override
    public int compareTo(Student s) {
        return id > s.id ? 1 : -1;
    }
}
```

▼ Set

TreeSet

- Elements are sorted while fetching (**Ascending Order**).
- Don’t have duplicate values.
- `Set<Integer> s = new TreeSet<>();`

HashSet

- Fetching elements are random (uses Hash Algorithm).
- Don’t have duplicate values.
- `Set<Integer> s = new HashSet<>();`

Add values :

`s.add(value);`

Print values :

```
Set<Integer> s = new TreeSet<>();
for(Integer i : s){
  System.out.println(i);
}
```

Methods in Set :

Method	Use
list.add(obj);	adds object
list.remove(obj);	remove object from list
list.size()	returns size of the list
list.contains(obj)	checks object present or not
list.clear()	clears the list
list.addAll(list)	adds list to the current list
list.toArray()	converts list into array

▼ Map

HashMap

- Stores data in key - value pair.
- Key must be unique.
- HashMap is not Synchronized.
- `Map<Integer, String> map = new HashMap<>();`

HashTable

- Stores data in key - value pair.
- Key must be unique.
- HashTable is non Synchronized.
- `Map<Integer, String> map = new Hashtable<>();`

Add values :

```
map.put(key, value);
```

Print values :

```
1)    for (Map.Entry<Integer, String> m : map.entrySet())                    // Prints both key & value
          System.out.println(m.getKey() + " : " + m.getValue());

2)    Set<Integer> key = map.keySet();                                       // Prints both key & value
      for (Integer i : key)
          System.out.println(map.get(i));

3)    for (String value : map.values())                                      // Prints only value
          System.out.println(value);

4)    for (Integer key : map.keySet())                                       // Prints both key & value
          System.out.println(map.get(key));
```

Methods in Set :

Method	Use
map.put(k, v)	adds object
map.get(k)	returns value of the key
map.remove(k)	remove object from map
map.size()	returns size of the map
map.entrySet()	returns both value & key as a set.
map.keySet()	returns keys into a Set.
map.values()	returns values as Collection.
map.clear()	clears the map.
map.containsValue(obj)	checks value present or not
map.containsKey(obj)	checks key present or not
map.isEmpty()	checks whether map is empty or not

▼ Collections Methods

Method	Use
Collections.sort(list)	sorts the list.
Collections.reverse(list)	reverse the list.
Collections.rotate(list, distance)	rotates the list by distance.
Collections.max(list)	returns max in the list.
Collections.min(list)	returns min in the list.
Collections.copy(list)	copies the list.
Collections.shuffle(list)	shuffles the elements in the list.
Collections.swap(list, pos1, pos2)	element swaps position with another.

▼ ArrayList vs LinkedList

Method	ArrayList	LinkedList
Get Element	**Fast**	**Slow**
Set Element	**Fast**	**Slow**

Add Element (to list end)	Fast	Fast
Insert Element (at position)	Slow	Fast
Remove Element	Slow	Fast

💡 ArrayList is faster in storing and accessing data. LinkedList is faster in manipulation of data

▼ Date & Time

- Date counting starts from 1-1-1970 in the system.

To get current Date & Time

```
Date date = new Date();
System.out.println(date);
```

Prints Time in ms

```
Date d = new Date();
System.out.println(d.getTime());
```

Date1 passed Date2?

```
Date date1 = new Date();
Date date2 = new Date("10/03
System.out.println(date1.aft
```

Time passed from beginning of the day

```
Date date = new Date();
System.out.println(date.getHours() + " : " +
                   date.getMinutes() + " : " +
                   date.getSeconds());
```

Days passed from beginning of Year

```
Date date1 = new Date();
Date date2 = new Date();

date2.setHours(0);
date2.setMinutes(0);
date2.setSeconds(0);

date2.setMonth(0);
date2.setDate(1);

long daysPassed = date1.getTime() -
                  date2.getTime();

int toDays = 1000 * 60 * 60 * 24;

System.out.println(daysPassed / toDays);
```

Date Formatter

```
SimpleDateFormat df = new Si
            ("MM/dd/yyyy ",
Date date = new Date();
String str = df.format(date)
System.out.println(str);
```

▼ Stream API

💡 Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

💡 filter > sorted > map > - - - - -

Different Operations On Streams-**Intermediate Operations:**

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.
   a. `List number = Arrays.asList(2,3,4,5);List square = number.stream().map(x->x*x).collect(Collectors.toList());`
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.
   a. `List names = Arrays.asList("Reflection","Collection","Stream");`
   b. `List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());`
3. **sorted:** The sorted method is used to sort the stream.
   a. `List names = Arrays.asList("Reflection","Collection","Stream");`
   b. `List result = names.stream().sorted().collect(Collectors.toList());`

Terminal Operations:

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.
    a. `List number = Arrays.asList(2,3,4,5,3);`
    b. `Set square = number.stream().map(x->x*x).collect(Collectors.toSet());`
2. **forEach:** The forEach method is used to iterate through every element of the stream.
    a. `List number = Arrays.asList(2,3,4,5);`
    b. `number.stream().map(x->x*x).forEach(y->System.out.println(y));`
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.The reduce method takes a BinaryOperator as a parameter.

    `List number = Arrays.asList(2,3,4,5);`

    `int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);`

    Here ans variable is assigned 0 as the initial value and i is added to it .

```
//forEach
empList.stream().forEach(emp -> System.out.println(emp));

//map
//collect
List<Employee> list=empList.stream()
    .map(emp ->new Employee(
        emp.getFname(),
        emp.getLname(),
        emp.getSalary()*1.10,
        emp.getProjects()
    )).collect(Collectors.toList());

//filter
empList.stream()
  .filter(emp->emp.getSalary()>30000)
  .map(emp ->new Employee(
       emp.getFname(),
       emp.getLname(),
       emp.getSalary()*1.10,
       emp.getProjects()
)).collect(Collectors.toList());

//filter
//findFirst
empList.stream()
    .filter(emp->emp.getSalary()>30000)
    .findFirst().orElse(null);

// flatMap
String projects=empList.stream()                //Stream<Employee>
    .map(emp -> emp.getProjects())              //Stream<List<String>>
    .flatMap(strings -> strings.stream())       //Stream<String>
    .collect(Collectors.joining(","));
System.out.println(projects);
```

```
//short Circuit operations
empList.stream()
      .skip(1)
      .limit(1)
      .collect(Collectors.toList());

//Finite Data
Stream.generate(Math::random)
      .limit(5)
      .forEach(value -> System.out.println(value));

// sorting
empList.stream()
      .sorted((o1, o2) ->
               o1.getFname().compareToIgnoreCase(o2.getFname()
             );

// min or max
empList.stream()
      .max(Comparator.comparing(Employee::getSalary))
      .orElseThrow(NoSuchElementException::new);

//reduce
empList.stream()
      .map(emp -> emp.getSalary())
      .reduce(0.0,Double::sum);

//grouping by (similar to group by in sql)
Map<String, List<Employee>> emp = empList.stream()
                        .collect(Collectors.groupingBy(Employee:
emp.forEach((k, v) -> System.out.println(k + " " + v));
```