Abstract:- It allows to hide the complex implementation details of a system. It will only expose the necessary details ~~about the~~ to the user.

Abstract Class:- It cannot be instantiated (no obj). It contain only abstract ~~class~~ methods as well as concrete methods.

* Sub classes must provide the implementation to the abstract methods.

final:- Final methods cannot be overriden. Used for specific implementations that should not be changed.

Abstract methods:- * They are declared without any implementation. Subclasses override these methods with the implementations.

Abstract methods only belong to the Abstract class.

- The subclass must implement the abstract methods when it inherits from the abstract class.

- Concrete methods:- It has the implementation inside the method.

- Abstract method:- It cannot have the implementation inside the method.

~~Inher~~

Interface	Abstract class
* No Constructors	* Can have constructors (no implementation)
* All methods are abstract by default	* Can have abstract & concrete ~~clas~~ methods. (partial implementation)
* Only public members (Access Modifiers)	* Can have public, private & protected members.
* A class can implement many interfaces.	* A class can inherit only from one Abstract class
Syntax:- class B implements interface A	Syntax:- class B extends class A.

- We can't create obj of both abstract class & Interface.

class B implements interface A, B → multiple interfaces

* ~~Can have only~~ Implicitly the vatiables are public, static & final	* Can have any variables.

class - class -> extends

class - interface -> implements

interface - interfaces -> extends.

* Subclasses can't inherit private methods& private variables from the super class

super:- Super Keyword is used to access the members (methods & variables) of the superclass. from subclass.

ABSTRACTION

Eg:-

```
public abstract class vehicle{
  public abstract void start Engine();
  public abstract void brake();
}
public class car extends vehicle{
  @Override
  public void start Engine()
   {
     sout(" Car Engine is started");
   }
  @Override
  public void brake(){
    sout (" Use car brakes");
  }
}
```

Here vehicle is an abstract class provides an abstraction interface for vehicles "car". Car implements the methods that it inherit from the abstract class. We can create the objects of the "Car" but not "Vehicles" in this we can achieve abstraction by hiding the implementation details of the Vehicle and provides common interface to interact with them

Encapsulation :-

Eg:

```
public class BankAccount{
  private double BankBalance;
  ~~public double~~ .
  public BankAccount (double initial Balance){
     balance = initialBalance;
  }
  public void deposit (double ~~depos~~ amount)
  {   balance + = amount;
  }
  public void withdraw (double amount){
     if ( balance >=amount){
         balance - = amount;
      }
     else{
         sout ("Insufficient balance");
      }
  }.
  public double getBalance ( ){
  ~~sout ("Bank balance~~
  return balance;
  }
```

In this example BankAccount class encapsulates the balance (details) variable It provides controlled interface for other classes to access the ~~element~~ methods inside the class.

LAMBDA EXPRESSION:-

Lambda expression is a short block of code that takes parameters & returns a value.

eg:- (x, y) —> x+y

↓ Parameters ↓ Arrow operator ↳ expression.

Boiler plate code:- It is a code repeated in multiple places with little or no variation. Like Getters & setters, Error handling, Database connection and Interface implementation.

Anonymous inner Class:-

* Traditionally to override a method a method and give it new implementation we need to create a subclass that extends superclass and then we will give new implementation to that method. We can avoid creating a new class that only perform one task like ~~adding~~ overriding the method instead we can use inner class.

* Anynymous inner class does' not have name.

eg:-
```
interface A{
    public void show(){
        sout( " It is A show");
    }
class Main{
    public static void main (string[] args)
{
@override  A obj = new A{
    public void show(){
    sout (" It is obj show"); }};}
```

In the above example we can't instantiate or create obj of the interface. we create an ~~anyn~~ anonymous inner class to override the method in the interface.

* Anonymous class are used in interface, funtional interface & lambda expressions

* Anonymous inner ~~interface~~ class can implement only one ~~method~~ interface at one time.

functional interface:- It is an interface that has only one method

* We can use Lambda expressions with only functional interface. We instantiate the interface with anonymous inner class. or we can use lambda expression to implement the functional interface's abstract methods without ~~using~~ a need of creating new class

Advantages of Lambda expressions:-

- Concise code
- Improved readability.
- Reduced boiler plate code

Uses of Lambda expression:-

Event handling:- It is used in Eventhand such as button ~~dict~~ clicks & mouse moven

Data Processing:- Lambda expressions can be used to ~~execute~~ process the data, such as filtering & mapping

Concurrency:- Lambda expressions can be used to execute code concurrently such as parallel streams.

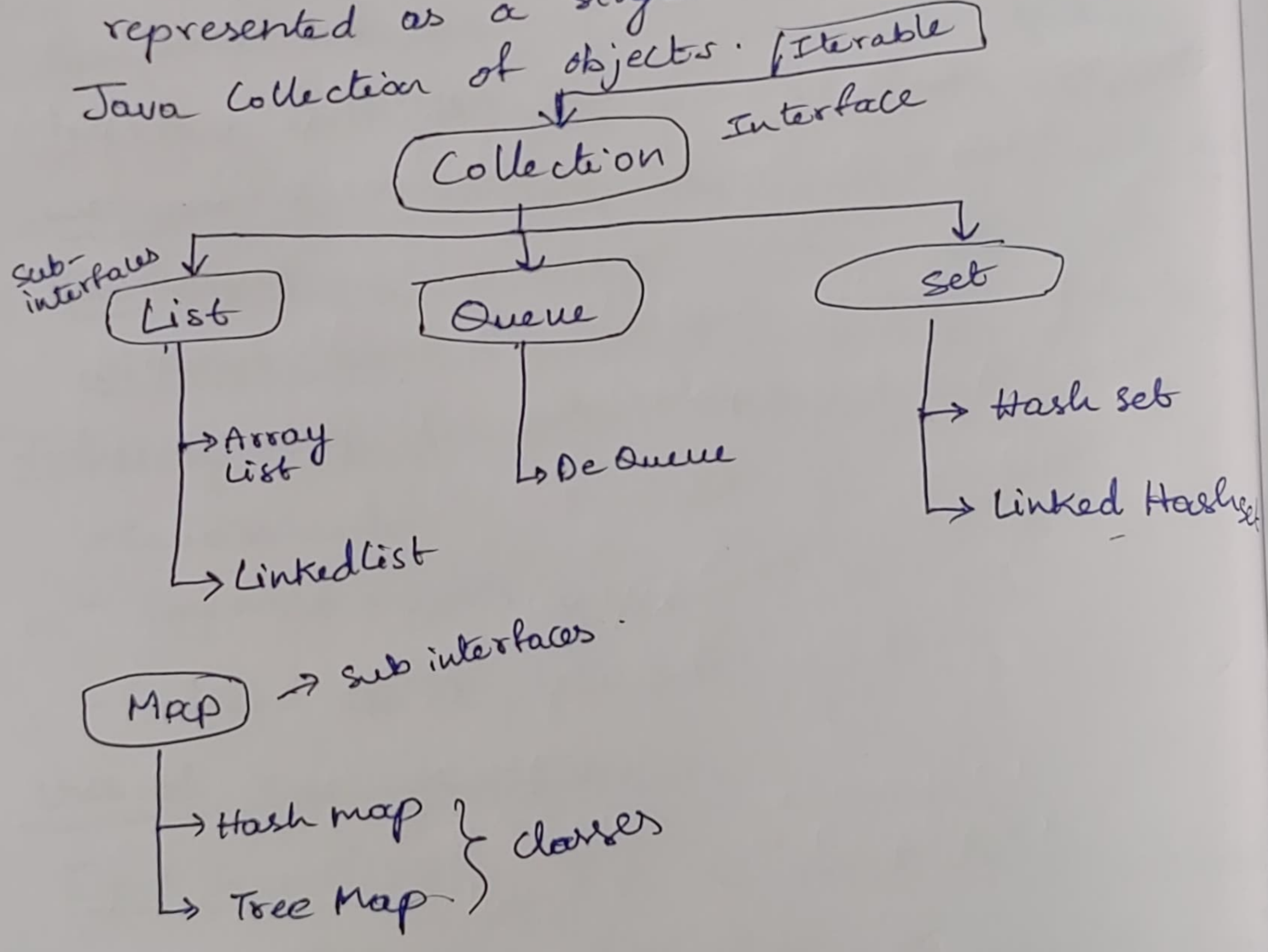
parallel streams:- parallel streams work by dividing the streams into smaller chunks called "splits" and process each split in parallel using multiple threads. The results from each split can be combined to produce final result.

COLLECTIONS

Collection interface:- The collection interface is ~~a~~ part of ~~Ja~~Collections framework (JCF). It is a part of Java.util package. It provides a set of classes & interfaces that can be used store & manipulate collection of objects.

* collection is not implemented by any class. But they are impleted ~~by~~ indirectly via subtypes or subinterfaces like List, Queue & set.

* Collections:- Any group of objects that are represented as a single unit is known as a Java Collection of objects.



$\underbrace{\text{Collection}}_{\text{collection interface}} \underbrace{\text{<Integer>}}_{\text{generics}} \underbrace{\text{nums}}_{\text{collection name}} = \text{new } \underbrace{\text{ArrayList}}_{\text{class}}\text{<Integer>();}$

* Generics ensure that the correct type of objects are added to a collection.

List:- * List is a child interface of collection interface. This interface is dedicated to the data of the list type in which we can store ordered collections of objects.
* Allows duplicates.
* It is implemented using various classes like ArrayList, ~~stb~~, stack, vector, LinkedList, etc

Syntax:-

```
List <Integer> num = new ArrayList<Integer>();
List <Integer> num = new LinkedList<Integer>();
List <Integer> num = new vector<Integer>();
```

Functions of List:-

~~addtel, add~~ add (value) / add (index) = value

~~remove~~ remove (object) - will remove the first occurence object / ~~remove~~(index)

size() - size of the list

contains(Object) - Returns boolean whether the obj is present in the list.

indexOf (obj)

toArray () - Returns an array containing all elements of list

Set:- A set is an unordered collection of objects.

* It does not have index.

* It does not have duplicate values

* Set interface can be impletmented by using sub classes like: Hash set, Tree set, Linked Hash set, etc.,

Syntax:-

```
LinkedHashSet <Integer> num= new HashSet<Integer>();
```

functions of set:-

add()
remove()
contains()
isEmpty()
size()

Map:- Map is not an implementation or does not extend collection. But it is a part of collection.

* Map itself is an interface which supports Key-Value pair. Hash Map & Tree Map are the classes that implement the Map.

* Keys must me unique.

* Values can be repeated.

functions in map:-

put()
get()
~~get Key()~~
~~get Ke~~

KeySet()
Values()
size()

remove()
isEmpty()

Comparator :- Comparator is also an interface that has method called compare

* We can use an interface by implementing ~~anyon~~ anonymous inner class

compare method works on an algorithm where it compare two values and swops them.

Syntax:-

```
comparator com = new Comparator<Integer>()
{ public int compare (~~int~~ Integer i, ~~int~~ Integer j)
    { ~~if (i%j~~
          statements;
    }
}
collection.sort(collectionName, com);
```

↳ sorts the collection acording to the implementation in com.

STREAMS

* Stream is an interface that contain stream () method.

* Stream method returns object of type stre

* stream API process the collections of objects.

Syntax:-

```
Stream <T> stream;
```

T is either a class, object or data dependi upon the declaration.

* Stream don't change the original data structure

* Stream is not a data structure instead it will take inputs from collections, Arrays or I/O channels.

Uses of stream API:-

- Stream API is a way to express and ~~perform~~ operations process collection of objects.

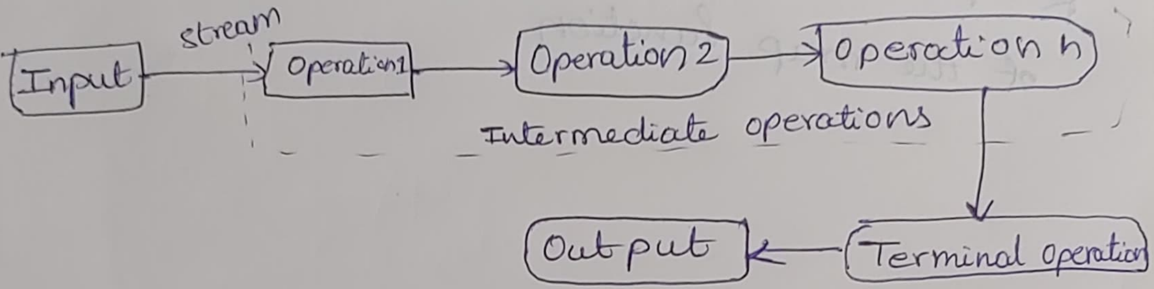
- It enables us to perform operations like filtering, mapping, reducing & sorting.

* Once stream is consumed we can't reuse it again.

* Each intermediate operation is lazily executed and returns a stream as a result. hence various intermediate operations can be pipelined.

* Terminal operations mark the end of the stream & return result

Types of Operations in Streams
- Intermediate Operations
- Terminate Operations.



Intermediate Operation:- Intermediate operations are kind of operations where multiple methods are chained together.

- Intermediate operations transform one stream to another stream.
- It enable filtering ~~, mapping~~ where one method filters data & passes it to another method after processing

Intermediate Operations:-

- map()
- filter()
- sorted()
- flatMap()
- distinct()
- Peak()

map ():- The map method is used to return stream consisting of applying the given function to the elements of this stream.

Syntax:-

<R>Stream<R> map (Function<? super T, ? exten R> mapper)

~~R represents return type of parameter that~~

~~repres~~

R - Parameter that represents return type of the map function.