If you have any questions or suggestions regarding the notes, please feel free to reach out to me on


**Sanjay Yadav Phone: 8310206130**

https://www.linkedin.com/in/yadav-sanjay/

https://www.youtube.com/@GeekySanjay

https://github.com/geeky-sanjay/


==========

# Roadmap: Java Basics to Advanced - Part 2

==========

## Different Operators:

==========

- Arithmetic Operator
- Relational Operator
- Short Circuit Operator
- Assignment Operator
- Logical Operator
- Ternary Operator
- Bitwise Operator
- Enums and its advanced usage

==========

## Control Flow Statements:

==========

- If Statement
- If Else Statement
- If Else Ladder
- Switch Statement and when to use
- For Loop
- While Loop
- Do While Loop
- Break Statement
- Continue Statement
- Etc....

==========

## Exception Handling:

==========

- Handling of Compile Time and
- Handling of Run time errors

==========

## Generic Programming in Java:

==========

- Understand how to write generic classes and methods
  in java, and when to use

==========

## Java Collections:

==========

- **List:**
  - o Array List
  - o LinkedList
  - o Stack
- **Queue:**
  - o Priority Queue
  - o Dequeue
- **Set:**
  - o HashSet
  - o Tree Set
  - o LinkedHashSet
- **Map:**
  - o Tree Map
  - o HashMap etc.

==========

## Java 8 Features:

==========

- Functional Interface
- Lambda Expression
- Stream APIs
- Predicates
- ForEach() method
- Default and static method in the interface

==========

## Unit Testing:

==========

- Understand how to do unit testing of Java code.
- Mockito Framework Understand

==========

## Collections Framework - Part1

==========

- **What is Java Collection Framework**
- **Why we need Java Collection Framework**
- **Collections Hierarchy**
- **Iterable Interface**
- **Collection Interface**
- **Collection vs Collections**

**What is Java Collections Framework ->** A collection is a group of elements/objects. The framework is like a toolbox that helps us organise and handle these groups. It gives us tools, like methods, to add, change, remove, and arrange the things in the group. In Java, the Collections framework is a set of classes and tools that let us work with groups of items easily.

## What is Java Collections Framework?

- Added in Java version 1.2
- Collections are nothing but a group of Objects.
- Present in java.util package.
- Framework provides us the architecture to manage these "groups of objects" i.e. add, update, delete, search etc.

## Why do we need Java Collections Framework?

- Prior to JCF, we have Array, Vector, Hash tables.
- But the problem with that is, there is no common interface, so its difficult to remember the methods for each.

1. **List Interface:**

- ArrayList: Implements a dynamic array.
- LinkedList: Implements a doubly-linked list.
- Vector: A legacy class similar to ArrayList but synchronised.

2. **Set Interface:**
   - HashSet: Implements a set using a hash table.
   - LinkedHashSet: Maintains insertion order using a linked list.
   - TreeSet: Implements a set stored in a tree structure.

3. **Queue Interface:**
   - LinkedList: Can be used as a queue (implements both List and Queue interfaces).
   - PriorityQueue: Implements a priority queue.

4. **Map Interface:**
   - HashMap: Implements a map using a hash table.
   - LinkedHashMap: Maintains insertion order using a linked list.
   - TreeMap: Implements a map stored in a tree structure.

5. **Deque Interface:**
   - ArrayDeque: Implements a resizable array-based deque.

6. **Collections Class:**
   - Contains utility methods for working with collections, such as sorting, searching, etc.

7. **Comparator Interface:**
   - Used for custom sorting of objects.

8. **Iterable Interface:**
   - The root interface for all collection classes. It provides an iterator over the collection.
9. **Arrays Class:**
   - Contains various static methods for manipulating arrays.

These collections provide different functionalities and data structures to suit various programming needs. The framework is designed to be flexible and extensible, allowing developers to create their own implementations or extend existing ones.
**Why do we need the Java Collections Framework? ->** The great thing about the Collections Framework is that it gives us a common way to work with groups of things. Before, we only had arrays, vectors, and hashtables, each with its own set of methods.
Now, after the introduction of the Collections Framework, they created a common set of rules (an interface) that all these similar collections follow. So, we don't have to remember different

methods for each type of collection; they all play by the same rules. It's like they're part of the same family now.
- Prior to JCF, we have Array, Vector, Hash tables.
- But the problem with that is, there is no common interface, so it's difficult to remember the methods for each.

```
public class Main {
 public static void main(String[] args) {
   int arr[] = now int[4];
   //inset an element in an array
   arr[0] = 1;
   //get front element
   int val = arr[0]:
   Vector<Integer> vector = new Vector();
   //insert an element in vector
   vector.add(1);
   //get element
   vector.get(0);
 }
}
```

As you can see below, the blue items represent interfaces, and the purple ones are concrete classes. The primary interface is "Iterable," and the collection inherits from it. The collection has three interfaces: queue, list, and set. Because of these interfaces, there are some common methods shared among all collections. Additionally, based on the three different categories, more specific methods have been added for each type of collection.

==========

# Project Creation 101: Step-by-Step Guide for BookMyShow App Development!

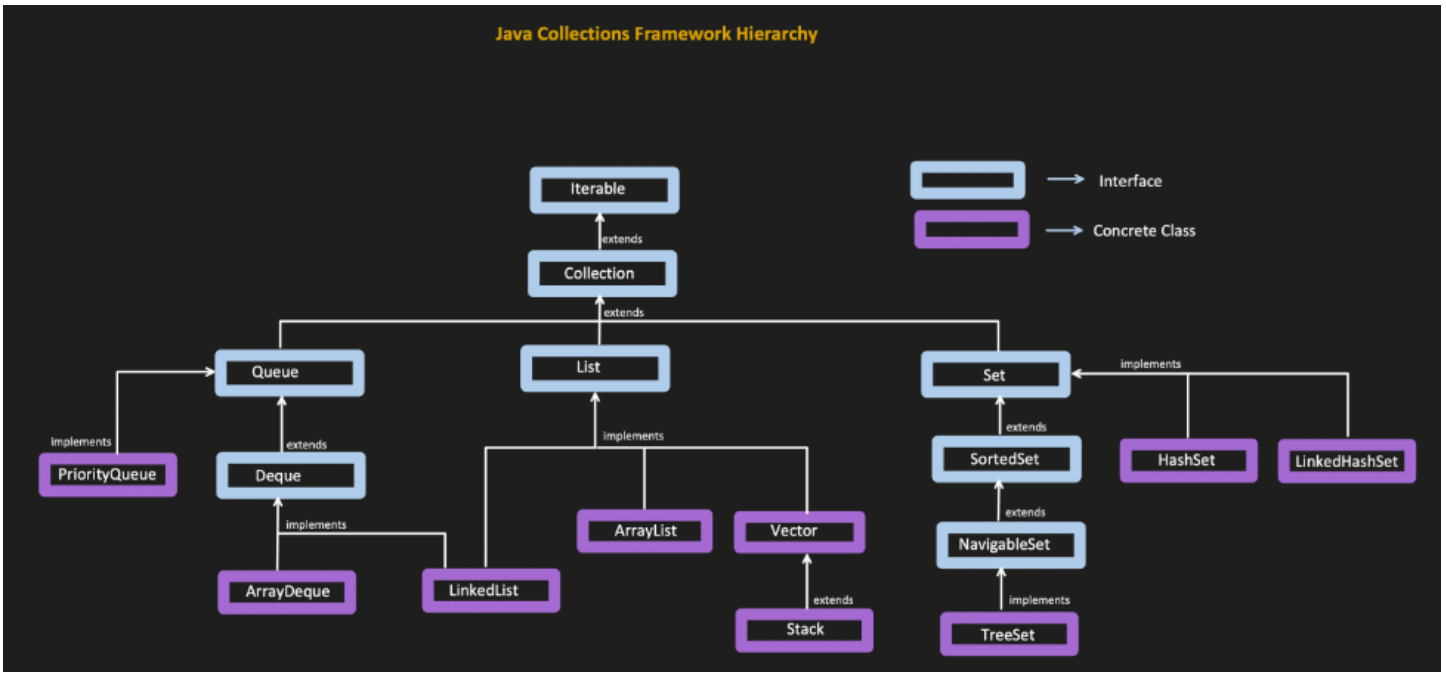
**Visit**

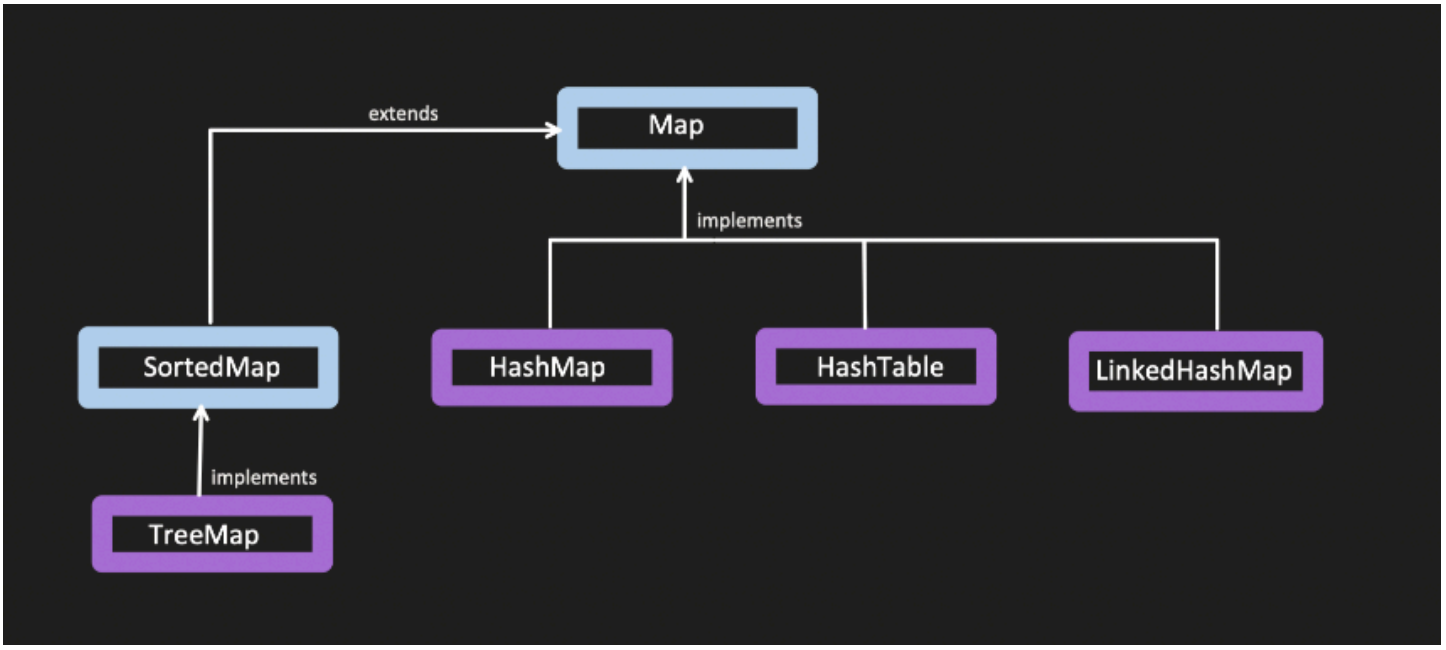
https://youtu.be/et7kLAaOwgk

**Like | Comment | Subscribe**

==========| **@GeekySanjay** |==========

==========




**Iterable:** It is implemented to allow an object to be the target of the "for-each-loop" statement.

**Iterable: Used to TRAVERSE the collection.**

**Below are the methods which are frequently used.....**

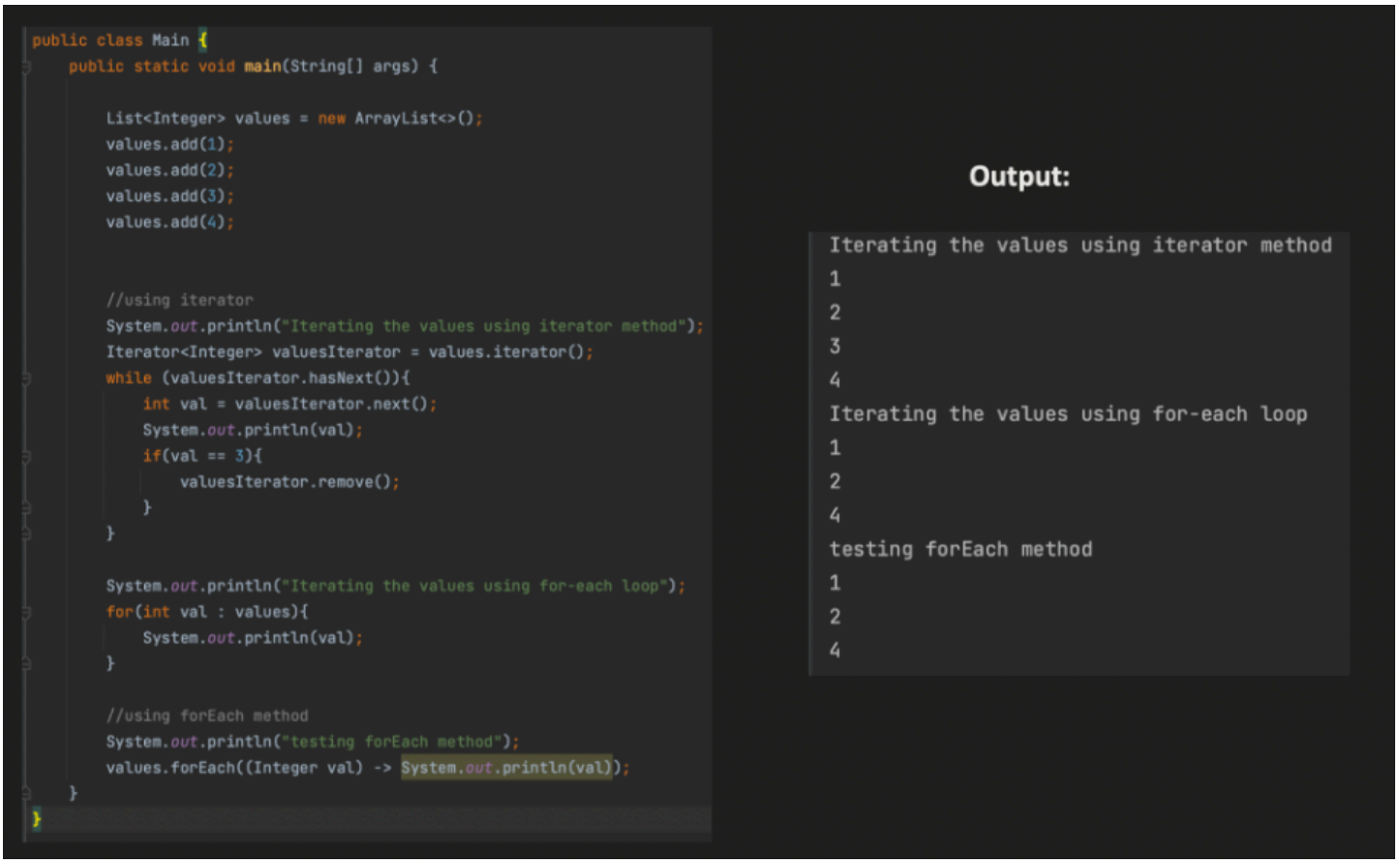
| S.NO | METHODS | Available in | USAGE |
|---|---|---|---|
| 1. | iterator() | Java1.5 | It returns the Iterator object, which provides below methods to iterate the collection. |
| 2. | forEach() | Java1.8 | Iterate Collection using Lambda expression. Lambda expression is called for each element in the collection. |

| Method Name | Usage |
|---|---|
| hasNext() | Returns true, if there are more elements in collection |
| next() | Returns the next element in the iteration |
| remove() | Removes the last element returned by iterator |

**First example**: In the screen below, we're using an ArrayList. We iterate through its values using an Iterable, which is the primary interface. give us an iterator() method which returns the Iterator object and this  Iterator object provides us with three methods: hasNext() checks if there's another element, and next() provides us with the element. We use a while loop with hasNext() to get all values. remove() method to eliminate the last element. This iterator is available for all collections in the same hierarchy.

**Second example**: We loop through the same elements using a forEach loop since Iterable has given us this feature.

**Third example**: We use a lambda expression "() -> statement" with the forEach method at last line, utilising the **lambda operator (->)** to iterate through the values.




==========

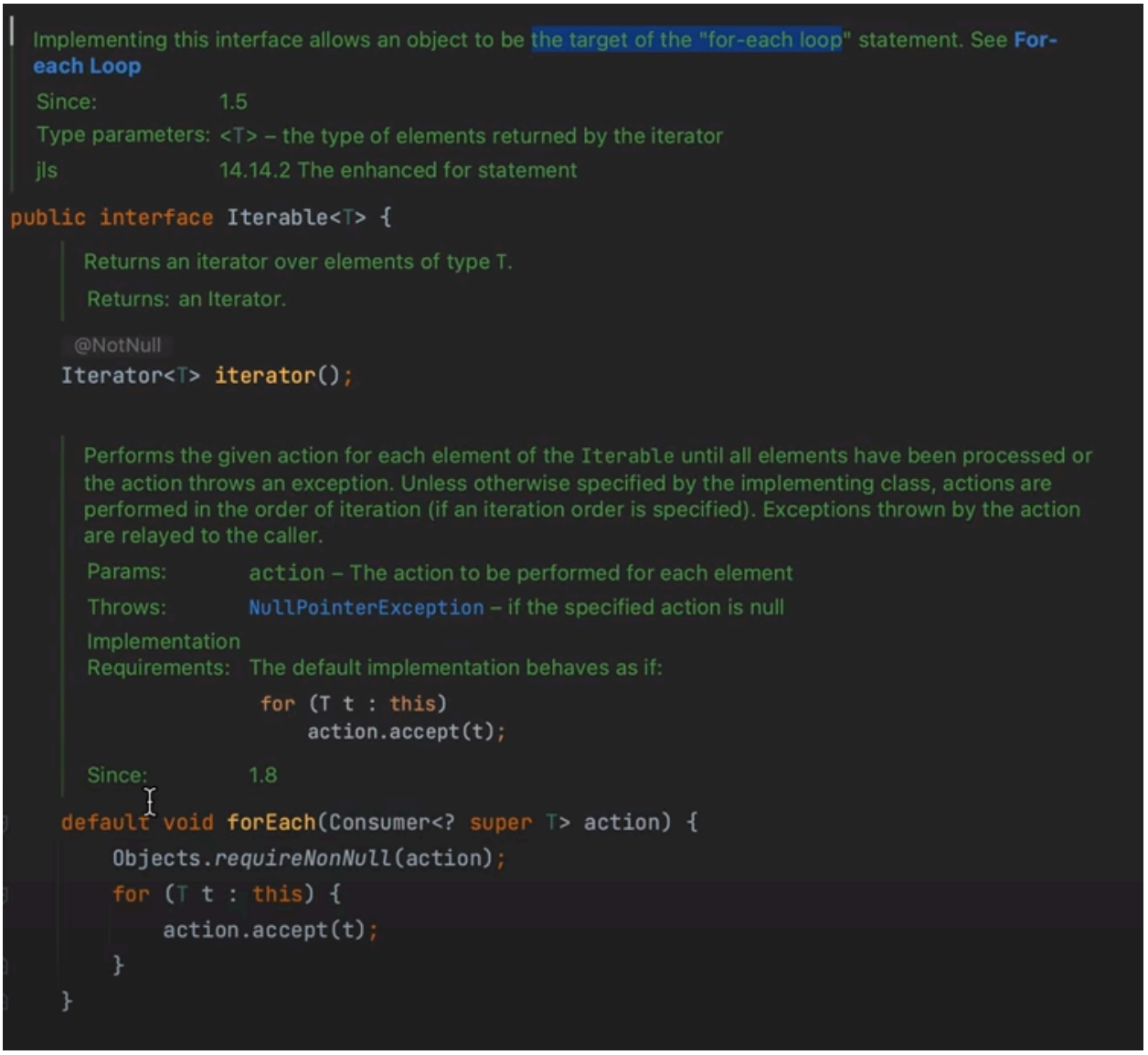
# Revolutionizing Tic-Tac-Toe: A Game with Advanced Strategies and Customization Options in Java!

**Visit**

https://youtu.be/B5zZa9vN4yo

**Like | Comment | Subscribe**

==========| @GeekySanjay |==========




As observed above, the Iterable interface offers a default implementation for the forEach method. In the forEach method, we use a consumer as a parameter, which is a functional interface. This allows us to utilise lambda functions inside the forEach loop, providing a convenient way to work with elements.

==========

# Master Git: A Comprehensive Local Repository Tutorial for Git!

**Visit**

https://youtu.be/0WMbbuORPxk

**Like | Comment | Subscribe**

==========| @GeekySanjay |==========

```java
/**
 * Represents an operation that accepts a single input argument and returns no result. Unlike most other
 * functional interfaces, Consumer is expected to operate via side-effects.
 *
 * This is a functional interface whose functional method is accept(Object).
 *
 * Since:          1.8
 * Type parameters: <T> – the type of the input to the operation
 */
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * Params: t – the input argument
     */
    void accept(T t);

    /**
     * Returns a composed Consumer that performs, in sequence, this operation followed by the after
     * operation. If performing either operation throws an exception, it is relayed to the caller of the
     * composed operation. If performing this operation throws an exception, the after operation will not be
     * performed.
     *
     * Params: after – the operation to perform after this operation
     *
     * Returns: a composed Consumer that performs in sequence this operation followed by the after
     *          operation
     *
     * Throws: NullPointerException – if after is null
     */
    @Contract(pure = true)  @NotNull
    default Consumer<T> andThen( @NotNull Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

**Collection: It represents the group of objects. Its an interface which provides methods to work on group of objects.**

**Below are the most common used methods which are implemented by its child classes like ArrayList, Stack, LinkedList etc.**

| S.NO | METHODS | Available in | USAGE |
|---|---|---|---|
| 1. | size() | Java1.2 | It returns the total number of elements present in the collection. |
| 2. | isEmpty() | Java1.2 | Used to check if collection is empty or has some value. It return true/false. |
| 3. | contains() | Java1.2 | Used to search an element in the collection, returns true/false. |
| 4. | toArray() | Java1.2 | It convert collection into an Array. |
| 5. | add() | Java1.2 | Used to insert an element in the collection. |
| 6. | remove() | Java1.2 | Used to remove an element from the collection. |
| 7. | addAll() | Java1.2 | Used to insert one collection in another collection. |
| 8. | removeAll() | Java1.2 | Remove all the elements from the collections, which are present in the collection passed in the parameter. |
| 9. | clear() | Java1.2 | Remove all the elements from the collection. |
| 10. | equals() | Java1.2 | Used to check if 2 collections are equal or not. |
| 11. | stream() and parallelStream() | Java1.8 | Provide effective way to work with collection like filtering, processing data etc. |
| 12. | iterator() | Java1.2 | As Iterable interface added in java 1.5, so before this, this method was used to iterate the collection and still can be used. |

```java
public class Main {
    public static void main(String[] args) {
        List<Integer> values = new ArrayList<>();
        values.add(2);
        values.add(3);
        values.add(4);
        //size
        System.out.println("size: " + values.size());
        //isEmpty
        System.out.println("isEmpty: " + values.isEmpty());
        //contains
        System.out.println("contains: " + values.contains(5));
        //add
        values.add(5);
        System.out.println("added: " + values.contains(5));
        //remove using index
        values.remove( index: 3);
        System.out.println("removed using index: " + values.contains(5));
        //remove using Object, removes the first occurrence of the value
        values.remove(Integer.valueOf( i: 3));
        System.out.println("removed using object: " + values.contains(3));
        Stack<Integer> stackValues = new Stack<>();
        stackValues.add(6);
        stackValues.add(7);
        stackValues.add(8);
        //addAll
        values.addAll(stackValues);
        System.out.println("addAll test using containsAll: " + values.containsAll(stackValues));
        //containsAll
        values.remove(Integer.valueOf( i: 7));
        System.out.println("containsAll after removing 1 element: " + values.containsAll(stackValues));
        //removeAll
        values.removeAll(stackValues);
        System.out.println("removeAll: " + values.contains(8));
        //clear
        values.clear();
        System.out.println("clear: " + values.isEmpty());
    }
}
```

Output:

```
size: 3
isEmpty: false
contains: false
added: true
removed using index: false
removed using object: false
addAll test using containsAll: true
containsAll after removing 1 element: false
removeAll: false
clear: true
```

## Collection vs Collections

**Collection** is part of Java Collection Framework. And its an interface, which expose various methods which is implemented by various collection classes like ArrayList, Stack, LinkedList etc.

**Collections** is a Utility class and provide static methods, which are used to operate on collections like sorting, swapping, searching , reverse, copy etc.

```java
public class Main {
    public static void main(String[] args) {
        List<Integer> values = new ArrayList<>();
        values.add(1);
        values.add(3);
        values.add(2);
        values.add(4);

        System.out.println("max value:" + Collections.max(values));
        System.out.println("min value:" + Collections.min(values));
        Collections.sort(values);
        System.out.println("sorted");
        values.forEach((Integer val) -> System.out.println(val));
    }
}
```

| Methods |
|---|
| sort |
| binarySearch |
| get |
| reverse |
| shuffle |
| swap |
| copy |
| min |
| max |
| rotate |
| unmodifiableCollection |

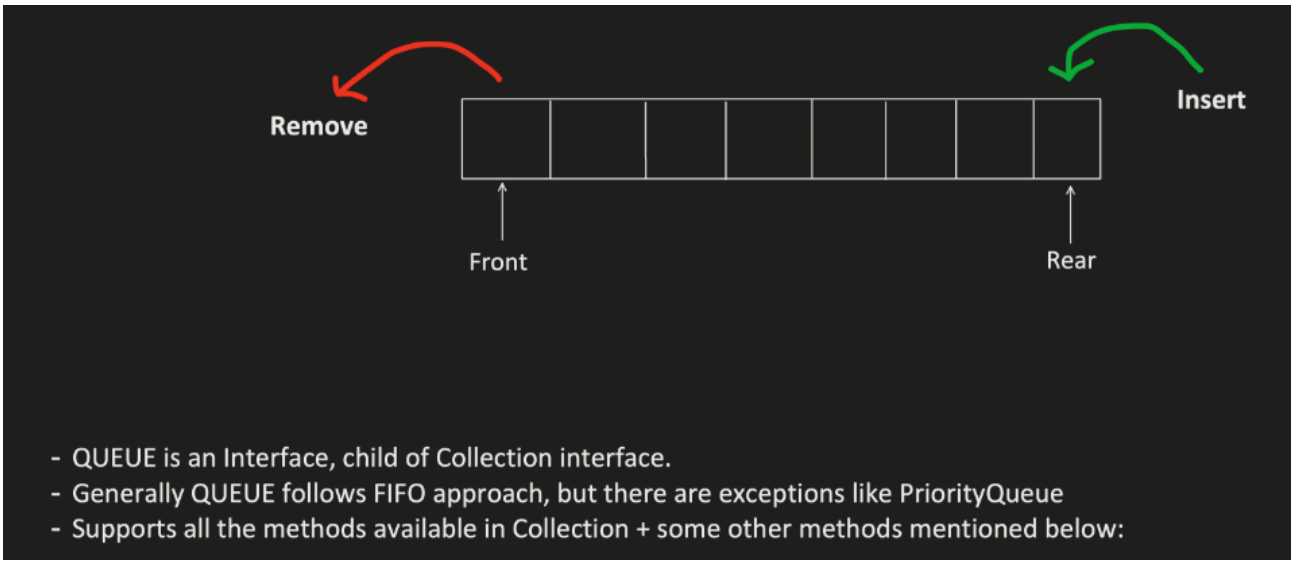
==========

# Collections Framework - Part2- Comparator Vs Comparable | PriorityQueue

==========

- **Queue Interface and its Methods**
- **PriorityQueue**
- **Comparator Vs Comparable**

## Queue Interface and its Methods



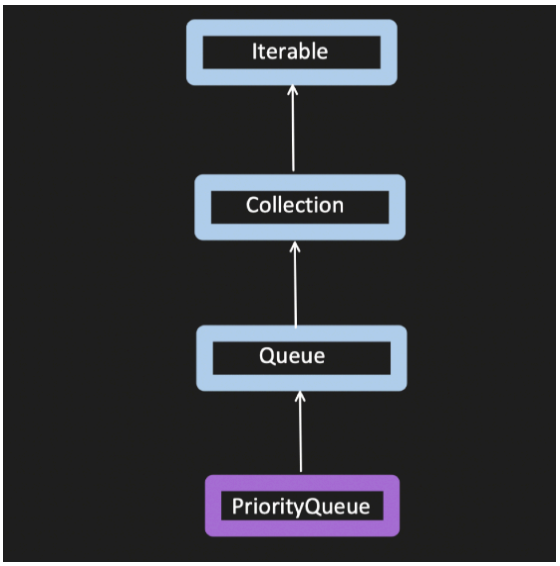

- QUEUE is an Interface, child of Collection interface.
- Generally QUEUE follows FIFO approach, but there are exceptions like PriorityQueue
- Supports all the methods available in Collection + some other methods mentioned below:

| S.No. | Methods Available in Queue Interface | Usage |
|---|---|---|
| 1. | add() | - Insert the element into the queue.<br>- True if Insertion is Successful and ***Exception*** if Insertion fails<br>- Null element insertion is not allowed will throw (**NPE**) |
| 2. | offer() | - Insert the element into the queue.<br>- True if Insertion is Successful and False if Insertion fails<br>- Null element insertion is not allowed will throw (**NPE**) |
| 3. | poll() | Retrieves and Removes the head of the queue.<br>Returns null if Queue is Empty. |
| 4. | remove() | Retrieves and Removes the head of the queue.<br>Returns ***Exception***(**NoSuchElementException**) if Queue is Empty. |
| 5. | peek() | Retrieves the value present at the head of the queue but do not removes it.<br>Returns null if Queue is Empty. |
| 6. | element() | Retrieves the value present at the head of the queue but do not removes it.<br>Returns an ***Exception*** (**NoSuchElementException**) if Queue is Empty. |

**Priority Queue ->** A Priority Queue is a data structure that manages a collection of elements with associated priorities. The elements are stored based on their priority, and the main operations include inserting an element and extracting the element with the highest (or lowest) priority.

The Priority Queue does not follow the typical first-in-first-out (FIFO) order, like a regular queue. Instead, the priority of elements determines the order in which they are processed.

**There are two common types of Priority Queues:**




1. Min Priority Queue (**Default**): In this type, the element with the smallest priority value is given the highest priority. It is often implemented using a min-heap data structure.

2. Max Priority Queue: Conversely, in a max priority queue, the element with the largest priority value is given the highest priority. It can be implemented using a max-heap.

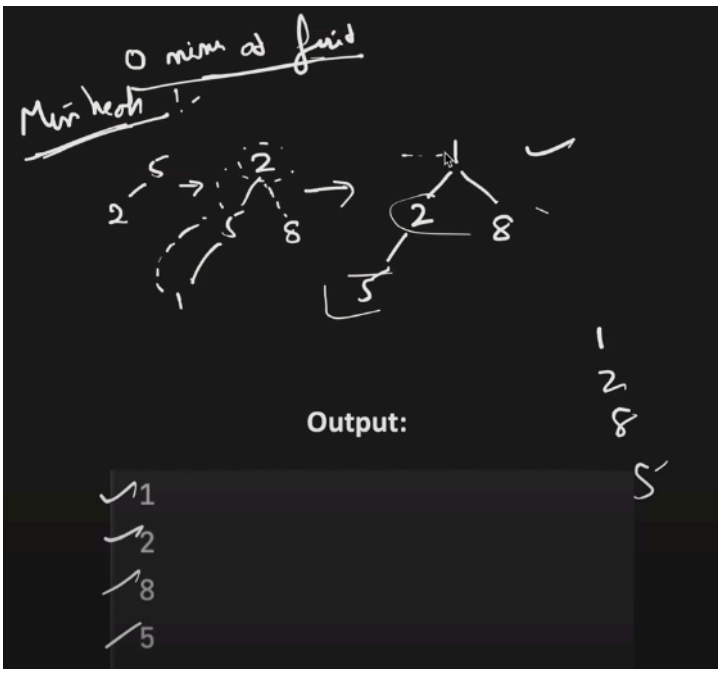
- Its of 2 types, **Minimum Priority Queue** and **Maximum Priority Queue**
- It is based on priority Heap (Min Heap and Max Heap).
- Elements are ordered according to either Natural Ordering (by default) or by ***Comparator*** provided during queue construction time.

**Min Priority Queue:**

```java
public class MinPriorityQueueExample {

    public static void main(String args[]){

        //min priority queue, used to solve problems of min heap.
        PriorityQueue<Integer> minPQ = new PriorityQueue<>();
        minPQ.add(5);
        minPQ.add(2);
        minPQ.add(8);
        minPQ.add(1);

        //lets print all the values
        minPQ.forEach((Integer val) -> System.out.println(val));

        //remove top element from the PQ and print
        while(!minPQ.isEmpty())
        {
            int val = minPQ.poll();
            System.out.println("remove from top:" + val);
        }
    }
}
```

**Output:**

```
1
2
8
5
remove from top:1
remove from top:2
remove from top:5
remove from top:8
```




A priority queue, like the one described earlier, stores elements in a heap structure. When a new element is added with an insert operation, the priority queue automatically adjusts and rearranges the elements to maintain its order.

In the context of sorting a Priority Queue, the default is a Min Heap, and it follows a level order traversal. As observed in the example, the output is 1, 2, 8, 5.

**Max Priority Queue** -> A Max Priority Queue is a type of priority queue where the highest priority element has the maximum value. In contrast to a Min Priority Queue, which uses a min-heap and prioritises smaller elements, a Max Priority Queue uses a max-heap and prioritises larger elements.

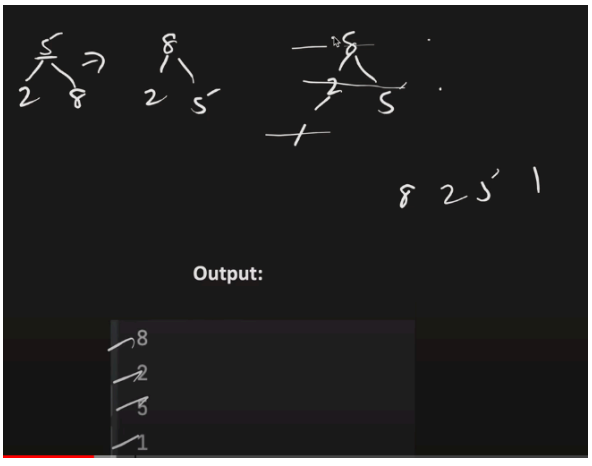
In Java, a Max Heap and a Min Heap are both implementations of the Heap data structure, a binary heap data structure. A binary heap is a complete binary tree where the value of each node is greater than or equal to (for a max heap) or less than or equal to (for a min heap) the values of its children.

We are utilising a lambda expression instead of a complete comparator function to define the Max Priority Queue in this context.

```java
public class MaxPriorityQueue {

    public static void main(String args[]){

        //max priority queue, used to solve problems of max heap
        PriorityQueue<Integer> maxPQ = new PriorityQueue<>((Integer a, Integer b) -> b-a);
        maxPQ.add(5);
        maxPQ.add(2);
        maxPQ.add(8);
        maxPQ.add(1);

        //lets print all the values
        maxPQ.forEach((Integer val) -> System.out.println(val));

        //remove top element from the PQ and print
        while(!maxPQ.isEmpty()){
            int val = maxPQ.poll();
            System.out.println("remove from top:" + val);
        }
    }
}
```

**Output:**

```
8
2
5
1
remove from top:8
remove from top:5
remove from top:2
remove from top:1
```

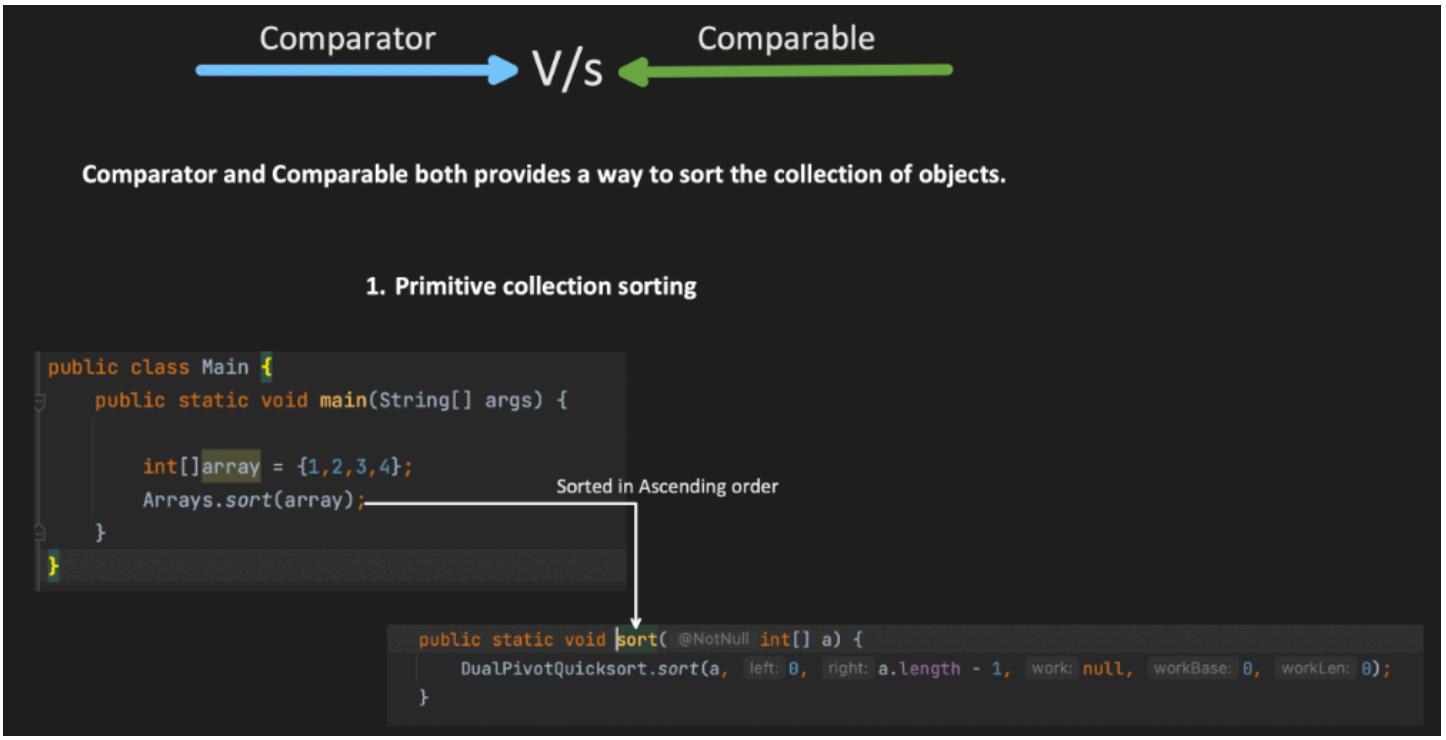



Time complexity:

Add and Offer : O(logn)
Peak : O(1)
Poll and Remove head element: O(logn)
Remove arbitrary element : O(n)

A priority queue, like the one described earlier, stores elements in a heap structure. When a new element is added with an insert operation, the priority queue automatically adjusts and rearranges the elements to maintain its order. like here max element on top.

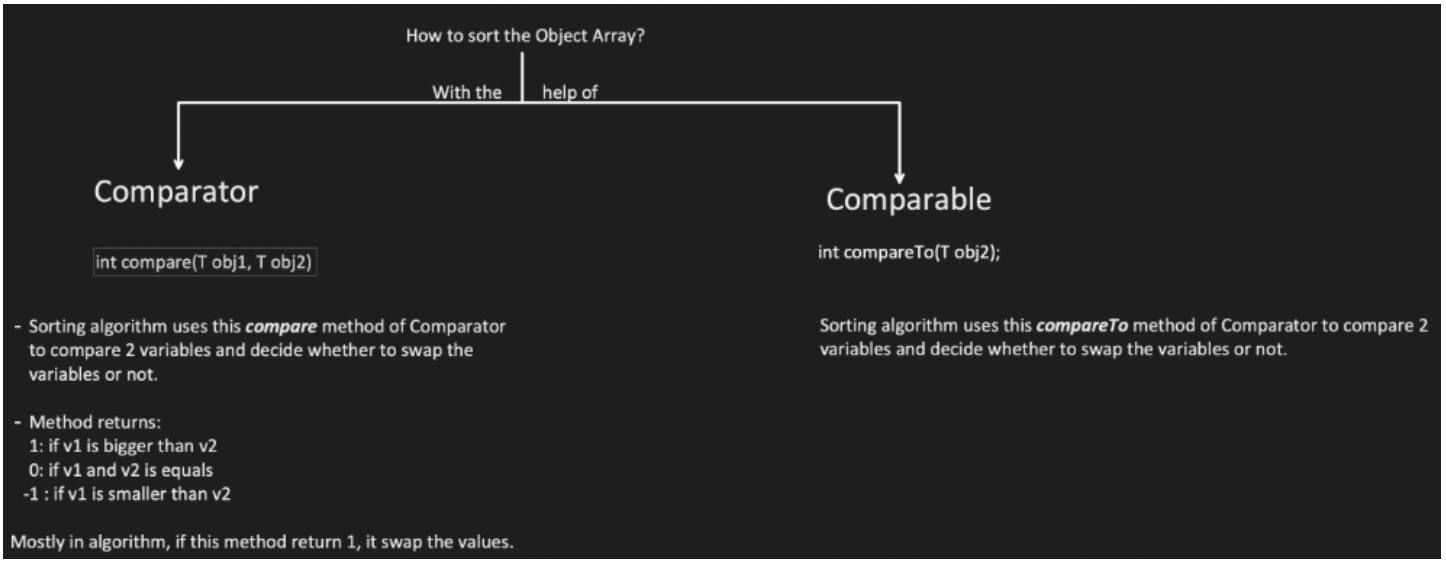
**Comparator** ---------->




Whenever the sorting operation is performed, the Quicksort, mergesort, timesort  algorithm will be utilized to arrange the integers.


We use "Comparable" or "Comparator" when we want to change how things are naturally sorted. By default, Java sorts numbers in ascending order. If we want descending order or need to sort objects, we use "Comparator" or "Comparable". This is because, when sorting objects, the system looks for a "compare" method in them to figure out the sorting logic.

A Comparator is a functional interface with one abstract method called compare, taking two objects (obj1 and obj2) and returning an integer.




```java
public class Main {
 public static void main(String[] args) {
   Integer a[] = {6,4,1,9,2,11};
   Arrays.sort(a, (Integer val1, Integer val2) -> val1-val2);

   for(int v : a){
     System.out.println(v);
   }
 }
}
```

We provided the array and a comparator function. This function normally does natural sorting, but if we pass it, the sorting will follow our specific logic. The sorting algorithm is already written; we just need to inform it about the type of sorting we want by passing the comparator function.

```java
public class Main {
 public static void main(String[] args) {
   Integer a[] = {6,4, 1,9, 2, 11};
   Arrays.sort(a, (Integer val1, Integer val2) -> val2-val1);

   for(int v : a){
     System.out.println(v);
   }
 }
}
```

This will sort in descending order

```
public class Car {
 String carName;
 String carType;

 Car(String name, String type) {
   this.carName = name;
   this.carType = type;
 }
}
```

```
public class Main {
 public static void main(String[] args) {
   Car[] carArray = new Car[3];
   carArray[0] = new Car( name: "SUV", type: "petrol");
   carArray[1] = new Car( name: "Sedan", type: "diesel");
   carArray [2] = new Car( name: "HatchBack", type: "cng");

   Arrays.sort(carArray, (Car obj1, Car obj2) ->
   obj2.carType.compareTo(obj1.carType));

   for(Car car: carArray) {
     System.out.println(car.carName + "..." + car.carType);
   }
 }
}
```

**Output:**

```
SUV..petrol
Sedan..diesel
HatchBack..cng
```

Here, we pass the `compareTo` function to compare two objects using a comparator. This will sort them in descending lexicographical order.

Another way to sort implement comparator in class and use it with sort

```
public class Car {

    String carName;
    String carType;

    Car(String name, String type){
        this.carName = name;
        this.carType = type;
    }
}
```

```
public class Car {
 String carName;
 String carType;

 Car(String name, String type){
   this.carName = name;
   this.carType = type;
 }
}
```

```
public class CarNameConparator implements Comparator<Car> {
 @Override
 public int compare(Car 01, Car 02) {
   return 02.carName.compareTo(01.carNane);
 }
}
```

```
public class Main {
 public static void main(String[] args) {
   List<Car> cars = new ArrayList<>();
   cars.add (new Car( name "suv", type: "petrol")):
   cars.add (new Car( name "sedan", type: "diesel")):
   cars.add (new Car( name: "hatchback", type: "cng")):
   Collections.sort(cars, new CarliameConparator()) ;
   cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." +
carObj.carType));
 }
}
```

Here, we make a new class by implementing a comparator and the compare method. However, this alone doesn't perform sorting. To actually sort, we need to pass this class into a sorting function. In this case, since we're using a List, we'll use the Collections.sort method instead of a simple sort method. Both methods use the same algorithm, but the Collections.sort method internally utilises a list for sorting.

Here, we're employing a dedicated class, but we can also use an anonymous class directly within the sorting function or, for a more concise approach, a lambda expression.

```
public class Car implements Comparator<Car> {
 String carName;
 String cartype;
 Car(){

 }
```

```
  Car(String name, String type) {
    this.carName = name;
    this.carType = type;
  }

  @Override
  public int compare(Car 01, Car 02) {
    return 01.carName.compareTo(02.carName);
  }
}
```

As seen above, we've assigned a default sorting behaviour to the Car object. Now, anyone who wishes to sort can simply pass this class object, and it will sort without requiring a comparator function as below.

```
public class Main {
  public static void main(String[] args) {
    List<Car> cars = new ArrayList<>();
    cars.add (new Car( name "suv", type: "petrol")):
    cars.add (new Car( name "sedan", type: "diesel")):
    cars.add (new Car( name: "hatchback", type: "cng")):
    Collections.sort(cars, new Car()) ;
    cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." +
carObj.carType));
  }
}
```

**Comparable**

Comparator provides various ways to sort by accepting two objects in the compare method. On the other hand, Comparable offers only one way to sort. To use Comparable, we have to implement it directly in the class witch provide sorting label on object label, including the compareTo method and passing just one object. Sorting internally utilizes the compareTo method. If the injected object already implements compareTo, it will override the default method.

==========

# Mastering MySQL: Step-by-Step Installation Guide

## Visit

https://youtu.be/XjvQ7RcMaVI

**Like | Comment | Subscribe**

==========| @GeekySanjay |==========

==========

## Collections Framework - Part7 - Streams

==========

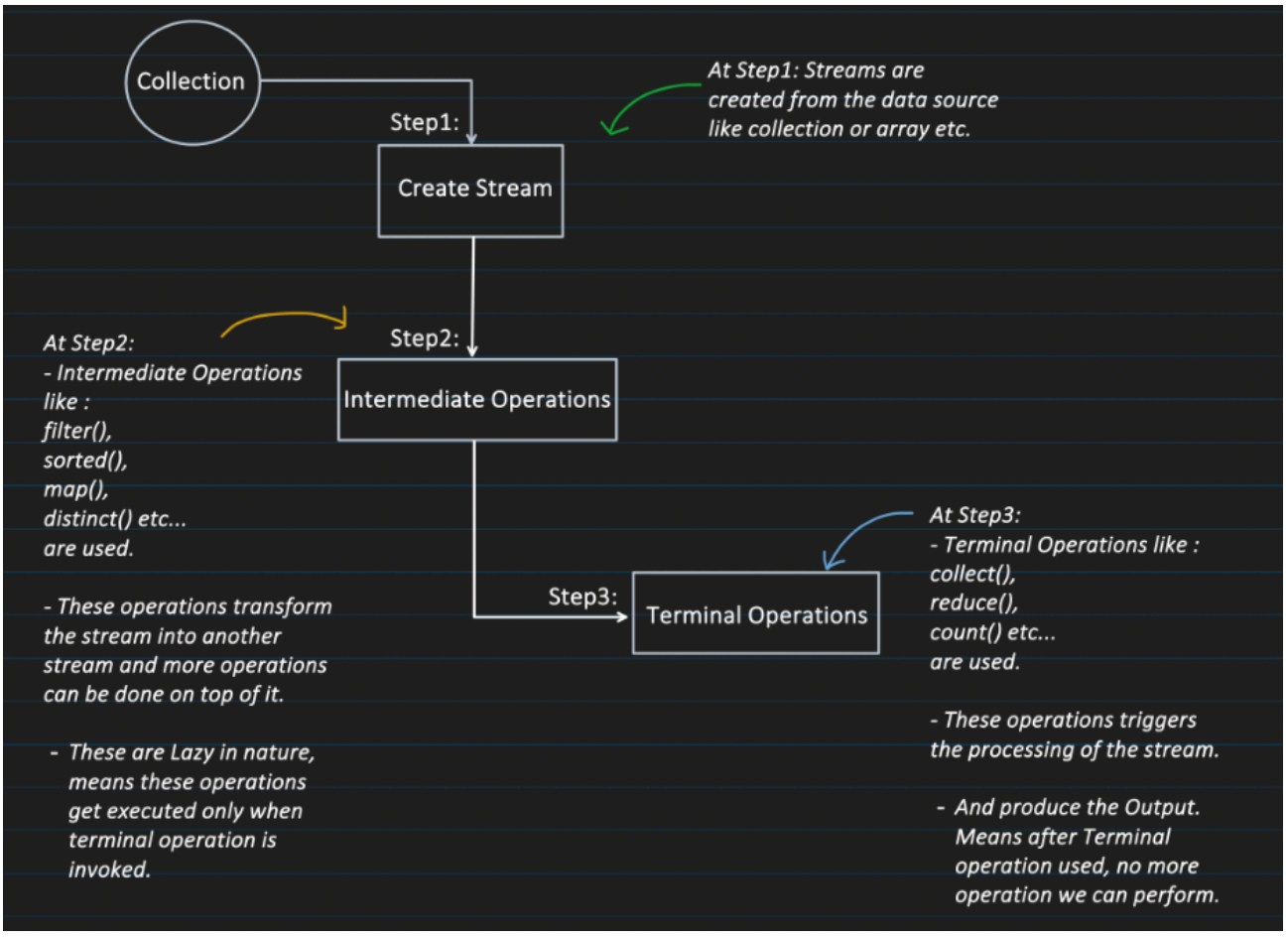
**Streams ->** a stream is a sequence of elements that you can process sequentially. It's like a flow of data that you can manipulate and operate on without having to store everything in memory at once. Java streams provide a flexible and efficient way to work with collections and perform various operations, such as filtering, mapping, and reducing, on the elements of the stream. They are a powerful feature introduced in Java to simplify and enhance the manipulation of data in a concise and expressive manner.
here is the link which you can take i screen

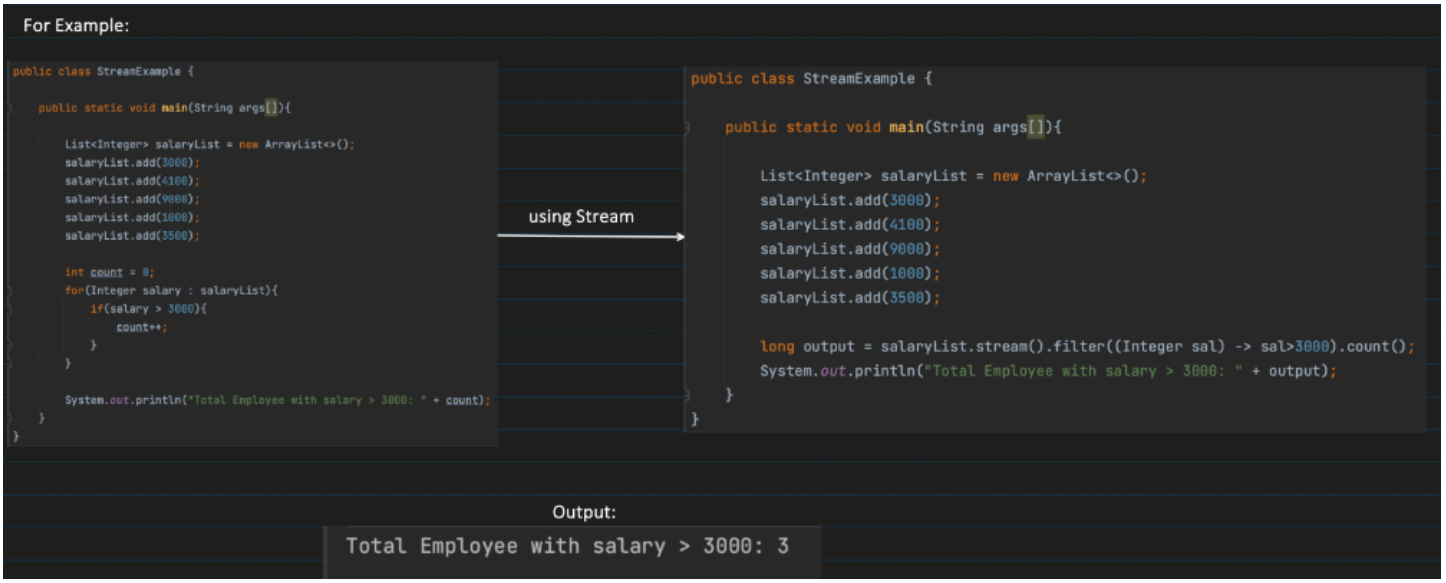
### What is Stream?

- We can consider Stream as a pipeline, through which our collection elements pass through.
- While elements passes through pipelines, it perform various operations like sorting, filtering etc.
- Useful when dealing with bulk processing. (can do parallel processing)

### How it operates: ->

As illustrated below, we can utilise it on any collection using the stream() method, creating a pipeline where we can filter and modify the data. The use of 'map' within the stream is possible because it returns a new stream without altering the original data; instead, it offers a new stream each time. To obtain the final result, we need to employ terminal operation methods like collect, count, or reduce, which provide the ultimate output for use.






In the example above, we are counting the salaries that exceed 3000. Initially, we use a loop and a condition for this purpose. In the second approach, we leverage streams. We initiate a stream() and apply a filter using a lambda function. This function runs on each element, evaluating whether the salary is greater than 3000. If true, the element is included in a new stream; otherwise, it is skipped. After completing the stream, we count the items in the resulting stream and return the final count as the output.

Streams are like a "wait and see" (lazy in nature) approach. They don't do anything until you specifically ask for a result. So, when you're using streams, the actual work or operations won't happen until you say, "Okay, now give me the final answer" by using a terminal function.

**Different ways to create a stream ->**

As we observe, there are different ways to create a stream. In the **first approach**, we directly use it on a collection list. In the **second scenario**, where we have an array, we can't apply it directly. Instead, we use Arrays.stream and pass the array to create a stream. The stream method isn't directly available in arrays like it is in the collection interface. It's implemented in the Arrays class, which converts the array to a stream. In **the third method**, we can create a stream directly using the static method "of" from the Stream class. This allows us to use it statically, passing the values to it, and getting a stream in return.

==========

# Downloading a Scaler Assignment Locally: Step-by-Step Guide

## Visit

https://youtu.be/TN36hpp4he8

**Like | Comment | Subscribe**

==========| **@GeekySanjay** |==========

==========

1. From Collection:

```
List<Integer> salaryList = Arrays.asList(3000, 4100, 9000, 1000, 3500);
Stream<Integer> streamFromIntegerList = salaryList.stream();
```

2. From Array:

```
Integer[] salaryArray = {3000, 4100, 9000, 1000, 3500};
Stream<Integer> streamFromIntegerArray = Arrays.stream(salaryArray);
```

3. From Static Method:

```
Stream<Integer> streamFromStaticMethod= Stream.of( ...values: 1000, 3500, 4000, 9000);
```

In this example, we're employing a stream using the Builder static method from the Stream class. This method returns a StreamBuilder, allowing us to add multiple values using the add method. Finally, we create the stream using the Build method, similar to the builder pattern.

In the fifth example, we use the iterate static method. Here, we start with an initial value and use a lambda function (like n + 5000) for the logic. It continuously generates new values based on the given logic. To prevent an infinite loop, we use the limit method, specifying the desired count of occurrences.

4. From Stream Builder:

```
Stream.Builder<Integer> streamBuilder = Stream.builder();
streamBuilder.add(1000).add(9000).add(3500);

Stream<Integer> streamFromStreamBuilder =  streamBuilder.build();
```

5. From Stream Iterate:

```
Stream<Integer> streamFromIterate = Stream.iterate( seed: 1000, (Integer n) -> n + 5000).limit( maxSize: 5);
```

## Different Intermediate Operation—---------- ->

So far, we've explored various ways to create streams. Now, let's delve into intermediate operations for modifying and filtering data. We can link multiple intermediate operations in a chain to conduct more intricate processing before applying a terminal operation to generate the final result.

1. **Filter** -> **It is used to filter element**. In the filter operation, a lambda function is used because the filter internally accepts a predicate (a functional component that takes one parameter and returns a boolean). This predicate is invoked on each item in the stream, determining whether the item meets the specified condition. After completing this process, the collect terminal method is employed. It converts the stream data to a list using the toList method and returns the result.

filter(Predicate<T> predicate)

```
Stream<String> nameStream = Stream.of( ...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING");
Stream<String> filteredStream = nameStream.filter((String name) -> name.length() <= 3);

List<String> filteredNameList = filteredStream.collect(Collectors.toList());

//OUTPUT: HOW, ARE, YOU
```

2. **Map** -> The 'map' operation is employed to alter data. It requires a method, which we can provide as a lambda expression. Similar to 'filter,' it runs in each iteration, modifying the value and adding it to the stream. Basically, it is used to transform each element.

map(Function<T, R> mapper)

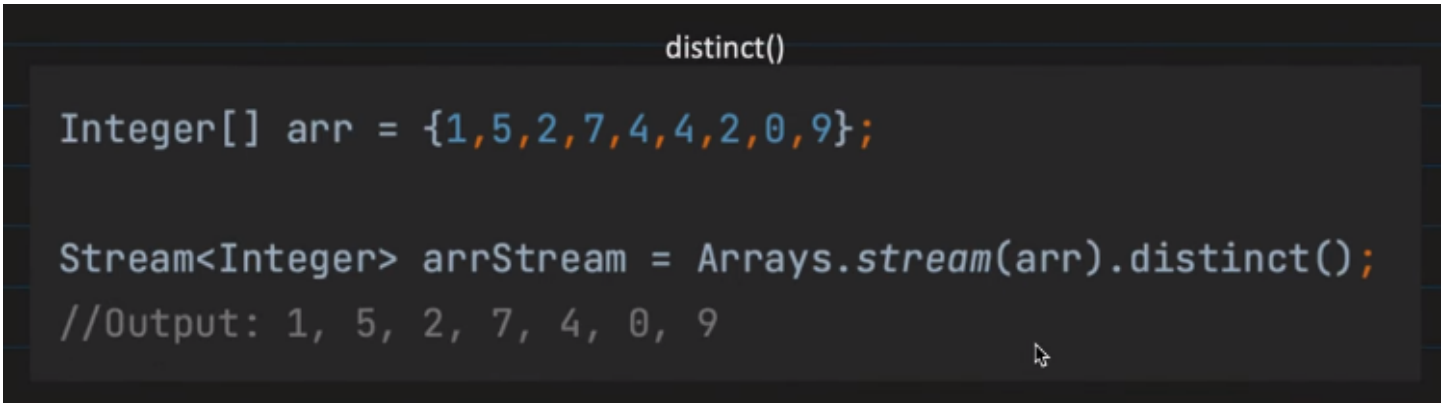
```
Stream<String> nameStream = Stream.of( ...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING");
Stream<String> filteredNames = nameStream.map((String name) -> name.toLowerCase());
//OUTPUT: hello, everybody, how, are, you, doing
```

3. flatMap -> flatMap is a method used in streams to transform each element of the stream into another stream and then flatten the streams into a single stream. It's particularly useful for dealing with nested collections or mapping operations that may result in multiple values per element. like we have a list of lists and we want to make it one stream then it is a very useful method.

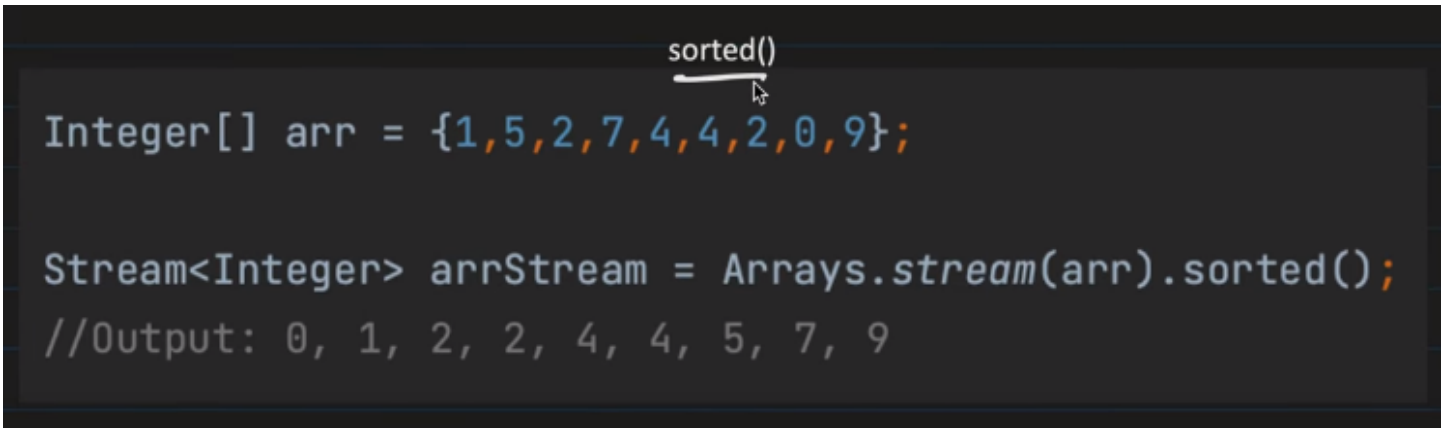
flatMap(Function<T, Stream<R>> mapper)

```
List<List<String>>  sentenceList = Arrays.asList(
        Arrays.asList("I", "LOVE", "JAVA"),
        Arrays.asList("CONCEPTS", "ARE", "CLEAR"),
        Arrays.asList("ITS", "VERY", "EASY")
);

Stream<String> wordsStream1 = sentenceList.stream()
        .flatMap((List<String> sentence)-> sentence.stream());
//Output: I, LOVE, JAVA, CONCEPTS, ARE, CLEAR, ITS, VERY, EASY

Stream<String> wordsStream2 = sentenceList.stream()
        .flatMap((List<String> sentence)-> sentence.stream().map((String value) -> value.toLowerCase()));
//Output: i, love, java, concepts, are, clear, its, very, easy
```

**distinct** -> distinct() is a method used on streams to remove duplicate elements, ensuring that only unique elements remain in the stream




**sorted** -> sorted() is a method used on streams to sort the elements of the stream in a specified order, either natural ordering or using a custom comparator.




```
Integer[] arr = {1,5,2,7,4,4,2,0,9};

Stream<Integer> arrStream = Arrays.stream(arr).sorted((Integer val1, Integer val2) -> val2-val1);
//Output: 9, 7, 5, 4, 4, 2, 2, 1, 0
```

**peek** -> peek() is a method used in streams to perform an action on each element of the stream without modifying the elements. It's often used for debugging or logging purposes to inspect the elements of the stream as they pass through the pipeline.
The peek() method in Java accepts a Consumer functional interface, which takes a parameter but doesn't return anything. It allows performing operations on each element of the stream, such as printing them out.

```
peek(Consumer<T> action)

  List<Integer> numbers = Arrays.asList(2,1,3,4,6);
Stream<Integer> numberStream = numbers.stream()
        .filter((Integer val)-> val>2)
        .peek((Integer val) -> System.out.println(val)) //it will print 3, 4, 6
        .map((Integer val) -> -1*val);
List<Integer> numberList = numberStream.collect(Collectors.toList());
```

**limit** -> limit() is a method used on streams to limit the number of elements in the stream to a specified maximum size. It's commonly used to reduce the size of a stream or to improve performance by processing only a portion of the stream

```
limit(long maxSize)

  List<Integer> numbers = Arrays.asList(2,1,3,4,6);
Stream<Integer> numberStream = numbers.stream().limit( maxSize: 3);

List<Integer> numberList = numberStream.collect(Collectors.toList());
//Output: 2, 1, 3
```

**skip** -> skip() is a method used on streams to skip a specified number of elements from the beginning of the stream. It allows you to bypass a certain number of elements before processing the remaining elements in the stream.

```
skip(long n)

  List<Integer> numbers = Arrays.asList(2,1,3,4,6);
Stream<Integer> numberStream = numbers.stream().skip( n: 3);

List<Integer> numberList = numberStream.collect(Collectors.toList());
//Output: 4, 6
```

**mapToInt** -> mapToInt() is a method used in streams to transform each element of the stream into an integer value. It's typically used when you want to convert elements to primitive integer types (int, short, byte), facilitating mathematical operations or reducing memory overhead compared to objects

As we can see in below example we are passing it premitive interger array and it will return IntStream means we can work directly in premitive int like if we will apply filter to it then we

can define lamda expration like (int item) -> {} but normaly in othor streams we can use premitive int we have to use like (Integer item) -> {}.

if you want to operate directly on primitive integer values within a stream. This is particularly useful for improving performance and reducing memory overhead when working with large datasets of primitive integers. Using mapToInt allows you to avoid the auto-boxing overhead associated with using the standard map operation with objects

mapToInt(ToIntFunction<T> mapper)

```
  List<String> numbers = Arrays.asList("2", "1", "4", "7");
IntStream numberStream = numbers.stream().mapToInt((String val) -> Integer.parseInt(val));

int[] numberArray = numberStream.toArray();
//Output: 2, 1, 4, 7

int[] numbersArray = {2, 1, 4, 7};
IntStream numbersStream = Arrays.stream(numbersArray);
numbersStream.filter((int val) -> val > 2);
int[] filteredArray = numbersStream.toArray();
//Output: 4, 7
```

**mapToLong** -> mapToLong() is a method used in streams to transform each element of the stream into a long value. Similar to mapToInt(), it's used when you want to work directly with primitive long types within a stream, reducing memory overhead and improving performance compared to using objects.

mapToLong(ToLongFunction<T> mapper)
(pls try it out....)

**mapToDouble** -> mapToDouble() is a method used in streams to transform each element of the stream into a double value. It's useful when you want to operate directly on primitive double types within a stream, offering performance benefits and reducing memory overhead compared to using objects.

mapToDouble(ToDoubleFunction<T> mapper)
(pls try it out....)

**Why we call Intermediate operation "Lazy"**

In Java Streams, intermediate operations are often referred to as "lazy" because they don't execute immediately when they're called. Instead, they are typically deferred until a terminal operation is invoked on the stream.

```java
public class StreamExample {

    public static void main(String args[]){

        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);
      Stream<Integer> numbersStream = numbers.stream().filter((Integer val) -> val >=3).peek((Integer val) -> System.out.println(val));
    }
}
```

**Output Nothing would be printed in the Output because there is no terminal method is there.**

```java
public class StreamExample {

    public static void main(String args[]){

        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);
      Stream<Integer> numbersStream = numbers.stream().filter((Integer val) -> val >=3).peek((Integer val) -> System.out.println(val));

      numbersStream.count(); //count is one of the terminal operation
    }
}
```

**Output: 4,7,10**

### Sequence of Stream Operations

In Java Streams, stream operations can be categorized into two main types: intermediate operations and terminal operations. The sequence of these operations in a stream is crucial for understanding how data flows through the stream pipeline.

1. Intermediate Operations:
    - Intermediate operations transform the elements of the stream and return a new stream as a result. They are usually chained together to form a pipeline.
    - Intermediate operations are lazy, meaning they do not execute immediately; instead, they are deferred until a terminal operation is invoked.
    - Common intermediate operations include filter, map, flatMap, sorted, distinct, limit, and skip, among others.

```
stream.forEach(...)
   .collect(...)
   .reduce(...)
   .count()
```

2. Terminal Operations:
    - Terminal operations consume the elements of the stream and produce a final result or side effect. They trigger the execution of the stream pipeline.
    - Terminal operations are eager, meaning they execute immediately when invoked and can't be further chained.
    - Common terminal operations include forEach, collect, reduce, count, min, max, anyMatch, allMatch, noneMatch, findFirst, and findAny, among others.

**Here's an example of a typical sequence of stream operations:**

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

int sum = numbers.stream()          // Create a stream from the list
        .filter(n -> n % 2 == 0)    // Keep only even numbers
        .mapToInt(n -> n * 2)       // Double each number
        .sum();                     // Calculate the sum
```

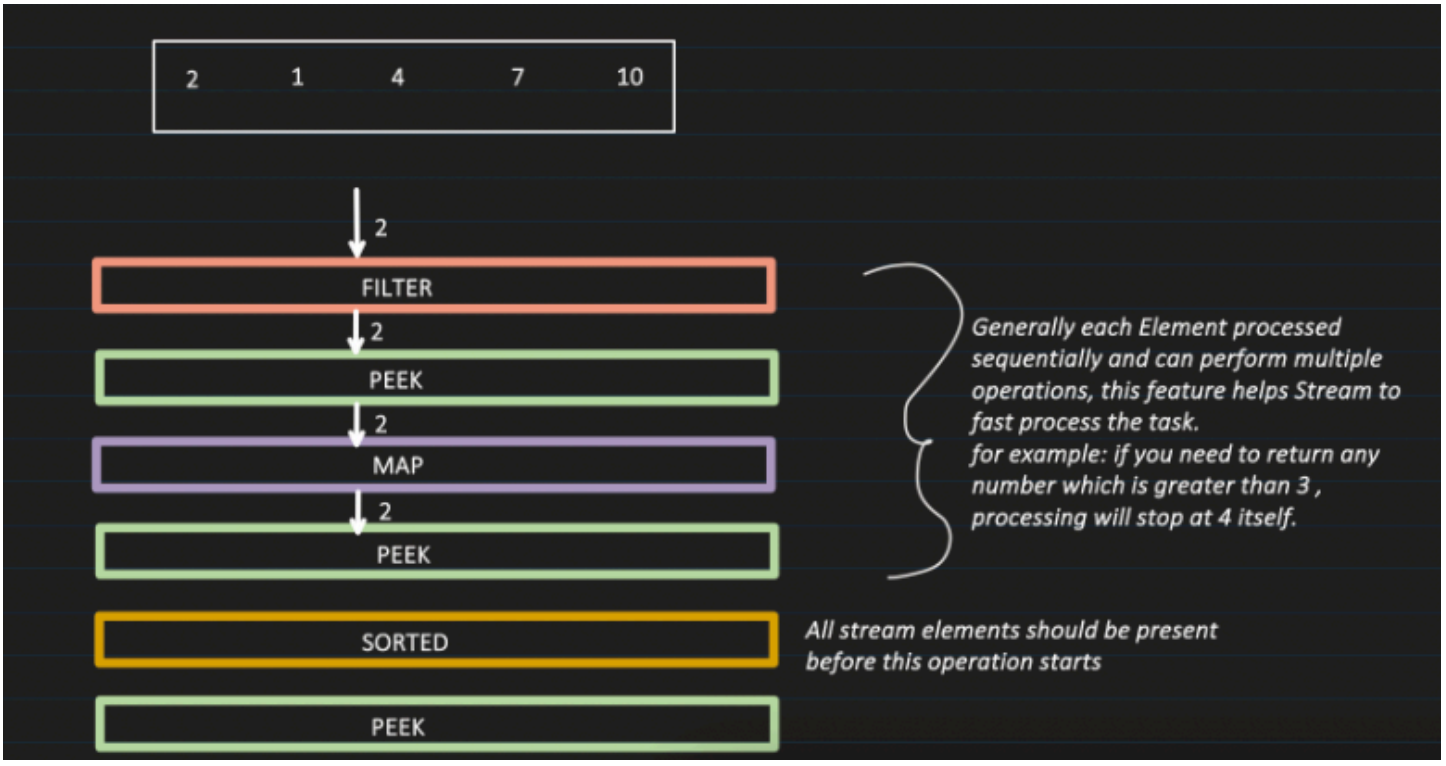
```
public class StreamExample {

    public static void main(String args[]){

        List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);
      Stream<Integer> numbersStream = numbers.stream()
              .filter((Integer val) -> val >=3)
              .peek((Integer val)-> System.out.println("after filter:" + val))
              .map((Integer val) -> (val * -1))
              .peek((Integer val)-> System.out.println("after negating:" + val))
              .sorted()
              .peek((Integer val)-> System.out.println("after Sorted:" + val));

      List<Integer> filteredNumberStream = numbersStream.collect(Collectors.toList());
    }
}
```

Expected Output:

```
after filter: 4
after filter: 7
after filter: 10

after negating: -4
after negating: -7
after negating: -10

after Sorted: -10
after Sorted: -7
after Sorted: -4
```

Actual Output:

```
after filter:4
after negating:-4
after filter:7
after negating:-7
after filter:10
after negating:-10
after Sorted:-10
after Sorted:-7
after Sorted:-4
```




**Different Terminal Operations**

Terminal operations are the once that produces the result. It triggers the processing of the stream

**forEach** -> It performs action on each element of the Stream and **DO NOT** Return any value.

forEach(Consumer<T> action)

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);
numbers.stream()
      .filter((Integer val) -> val >=3)
      .forEach((Integer val)-> System.out.println(val));

//OUTPUT: 4, 7, 10
```

**toArray** -> The toArray() method gathers all the items from the stream and puts them into an array. By default, it gives back an Object[] array. However, if we need a different type of array, we can use a lambda function to change it. In the second example, the lambda function takes the size and produces a new array of integers with that size.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Object[] filteredNumberArrType1 = numbers.stream()
      .filter((Integer val) -> val >=3)
      .toArray();

Integer[] filteredNumberArrType2 = numbers.stream()
        .filter((Integer val) -> val >=3)
        .toArray((int size) -> new Integer[size]);
```

**reduce** -> The reduce method performs a reduction on the elements of the stream. It applies associative aggregation functions like sum, average, max, min, or count.
**reduce(BinaryOperator<T> accumulator)**

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);


Optional<Integer> reducedValue = numbers.stream()
        .reduce((Integer val1, Integer val2) -> val1+val2);

System.out.println(reducedValue.get());
//output: 24
```

**collect** - In a stream, the collect method is used to accumulate elements into a collection, like a list, set, or map

**collect(Collector<T,A,R> collector)**

```
 List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

List<Integer> filteredNumber = numbers.stream()
         .filter((Integer val) -> val >=3)
         .collect(Collectors.toList());
```

**min - max** -> Finds the minimum or maximum element from the stream based on the comparator provided but it is wrong to say min and max because in min it will provide first number whatever it is min or max as you can see in below screen same for max it will return you last value.

**min(Comparator<T> comparator)**
**max(Comparator<T> comparator)**

```
  List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Optional<Integer> minimumValueType1 = numbers.stream()
        .filter((Integer val) -> val>=3)
          .min((Integer val1, Integer val2) -> val1-val2);

System.out.println(minimumValueType1.get());
//output: 4

  Optional<Integer> minimumValueType2 = numbers.stream()
          .filter((Integer val) -> val>=3)
          .min((Integer val1, Integer val2) -> val2-val1);

  System.out.println(minimumValueType2.get());
  //output: 10
```

**count** -> It returns the count of elements present in the stream.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);


long noOfValuesPresent = numbers.stream()
        .filter((Integer val1) -> val1>=3)
        .count();

System.out.println(noOfValuesPresent);
//output: 3
```

**anyMatch** -> Check if any value in the stream matches the given predicate and returns the boolean.
**anyMatch(Predicate<T> predicate)**

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

boolean hasValueGreaterThanThree = numbers.stream()
        .anyMatch((Integer val) -> val>3);

System.out.println(hasValueGreaterThanThree);
//output: true
```

**allMatch** -> Checks if all value in the stream match the given predicate and return the boolean.
allMatch(Predicate<T> predicate)

**Can you pls try it out...**

**noneMatch** -> Checks if no value in the stream match the given predicate and return the boolean.
noneMatch(Predicate<T›> predicate)

**Can you pls try it out..**

**findFirst** -> finds the first element of the stream.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Optional<Integer> firstValue = numbers.stream()
        .filter((Integer val) -> val >= 3)
        .findFirst();

System.out.println(firstValue.get());
//output: 4
```

**findAny**() -> finds any random element of the stream.

**Can you pls try it out..**

**How many times we can use a single stream:**
One Terminal Operation is used on a Stream, it is closed/consumed and can not be used again for another terminal operation.

As shown below, we utilized the stream with the **forEach terminal** operation. However, attempting to use **filterNumbers.collect** afterwards will result in an error because the stream is already closed.

```
List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);

Stream<Integer> filteredNumbers = numbers.stream()
        .filter((Integer val) -> val >= 3);

filteredNumbers.forEach((Integer val) -> System.out.println(val)); // consumed the filteredNumbers stream

//trying to use the closed stream again
List<Integer> listFromStream = filteredNumbers.collect(Collectors.toList());
```

Output:

```
4
7
10
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : stream has already been operated upon or closed
    at StreamExample.main(StreamExample.java:19)
```

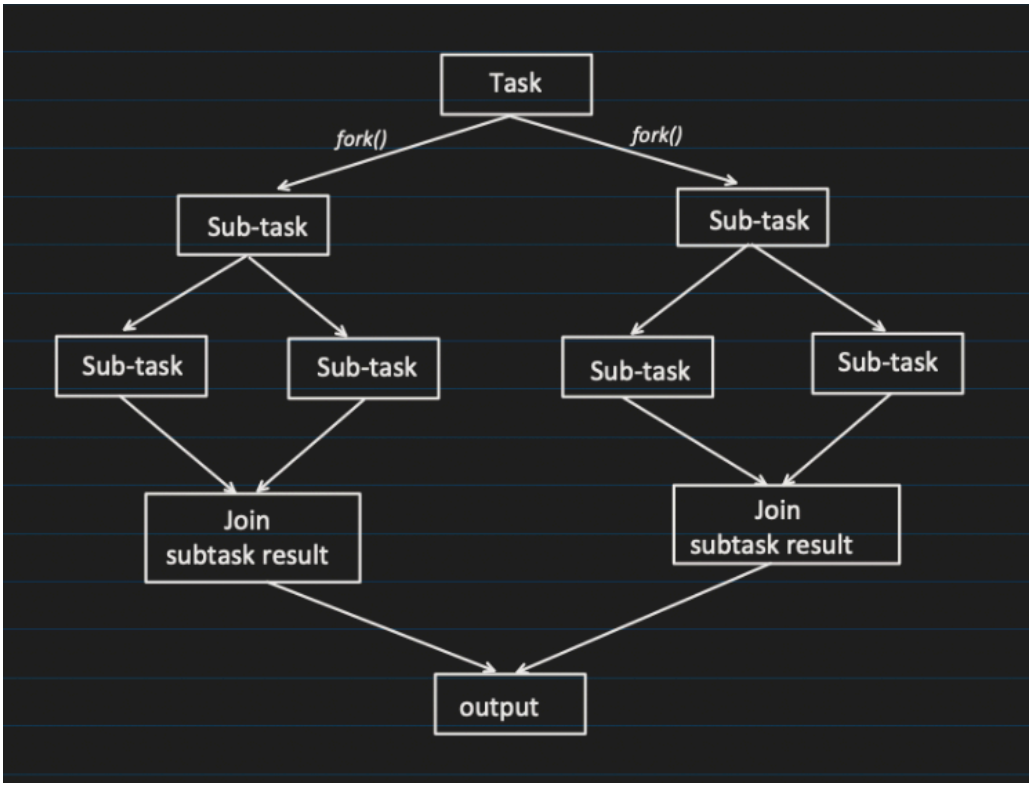
Parallel Stream: Helps to perform operation on stream concurrently, taking advantage of multi core CPU. ParallelStream() method is used instead of regular stream method.
Internally it does:
- Task splitting: it uses "spliterator" function to split the data into multiple chunks.
- Task submission and parallel processing: Uses Fork-Join pool technique.

```java
public class StreamExample {

    public static void main(String args[]) {

        List<Integer> numbers = Arrays.asList(11, 22, 33, 44, 55, 66, 77, 88, 99, 110);

        // Sequential processing
        long sequentialProcessingStartTime = System.currentTimeMillis();
        numbers.stream()
                .map((Integer val) -> val * val)
                .forEach((Integer val) -> System.out.println(val));
        System.out.println("Sequential processing Time Taken: " + (System.currentTimeMillis() - sequentialProcessingStartTime) + " millisecond");

        // Parallel processing
        long parallelProcessingStartTime = System.currentTimeMillis();
        numbers.parallelStream()
                .map((Integer val) -> val * val)
                .forEach((Integer val) -> System.out.println(val));
        System.out.println("Parallel processing Time Taken: " + (System.currentTimeMillis() - parallelProcessingStartTime) + " millisecond");
    }
}
```

```
121
484
1089
1936
3025
4356
5929
7744
9801
12100
Sequential processing Time Taken: 64 millisecond
```

```
7744
121
5929
4356
1936
484
12100
1089
9801
3025
Parallel processing Time Taken: 5 millisecond
```

As observed in the screenshot below, we delve into the internal workings of a parallel stream, specifically its use of fork-join mechanisms. The stream divides tasks into multiple subtasks based on the available CPU cores. Each subtask is then assigned to a CPU core for execution. Once all subtasks within the same tree are completed, the results are combined.



==========

# Multithreading and Concurrency in Java: Part1 | Threads, Process and their Memory Model in depth

==========

- **Introduction of Multithreading:**
  * Definition of Multithreading
  * Benefits and Challenges of Multithreading
  * Processes v/s Threads
  * Multithreading in Java

- Java Memory Model of Process and thread

- Basics of Threads - Part1:
  * Creating Threads
    * Extending the Thread Class
    * Implementing the Runnable Interface
  * Thread Lifecycle
    * New
    * Runnable
    * Blocked
    * Waiting
    * Timed Waiting
    * Terminated

- Basics of Thread - Part2 : Inter Thread Communication and Synchronization
  * Synchronization and Thread Safety
    * Synchronized Methods
      * Synchronized Blocks
  * Inter-Thread Communication
    * wait(), notify(), and notifyAll() methods
    * Producer-Consumer Problem - Assingment

- Basics of Threads - Part3
  * Producer-Consumer Problem - Solution discuss
  * Stop, Resume, Suspended method is deprecated, understand why and its solution
  * Thread Joining
  * Volatile Keyword
  * Thread Priority and Daemon Threads

- Some Advanced Topics
  * Thread Pools
    * Executor Framework
    * ThreadPoolExecutor
  * Callable and Future

    * Fork/Join Framework
    * ThreadLocal in Multithreading

- Concurrency Utilities
    * java.util.concurrent Package
    * Executors and ExecutorService
    * Callable and Future
    * CompletableFuture
    * ScheduledExecutorService
    * CountDownLatch, CyclicBarrier, Phaser, and Exchanger

- Concurrent Collections (already discussed during Collections topic, will provide working example for this)
    * ConcurrentHashMap
    * ConcurrentLinkedQueue and ConcurrentLinkedDeque
    * CopyOnWriteArrayList
    * BlockingQueue Interface
        * ArrayBlockingQueue
        * LinkedBlockingQueue
        * PriorityBlockingQueue

- Atomic Variables
    * AtomicInteger, AtomicLong, and AtomicBoolean
    * AtomicReference and AtomicReferenceArray
    * Compare-and-Swap Operations

- Locks and Semaphores
    * ReentrantLock
    * ReadWriteLock
    * StampedLock
    * Semaphores
    * Lock and Condition Interface

- Parallel Streams (already discussed during Stream topic, will provide working example for this)

- Best Practices and Patterns
    * Thread Safety Best Practices
    * Immutable Objects
    * ThreadLocal Usage
    * Double-Checked Locking and its Issues
    * Concurrency Design Patterns

- Common Concurrency Issues and Solutions
    * Deadlocks

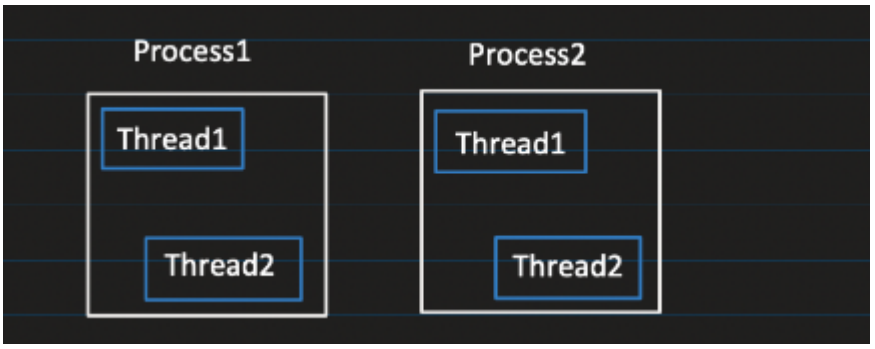
    * Starvation
    * Livelocks
    * Race Conditions
    * Strategies for Avoiding Concurrency Issues

- Java 9+ Features
    * Reactive Programming with Flow API
    * CompletableFuture Enhancements
    * Process API Updates

- Java 11+ Features
    * Local-Variable Type Inference (var keyword)
    * Enhancements in Optional class
    * New Methods in the String class relevant to concurrency

## Introduction of Multithreading

Before we understand what is Multithreading, lets first understand Thread and Process




**Process:** A process is an instance of a program that is getting executed. It possesses its own resources such as memory and threads, which are allocated by the operating system upon its creation.

**Compilation (javac Test.java):** This process generates bytecode that can be interpreted and executed by the JVM.

**Execution (java Test):** This stage initiates a new process by the JVM. In this context, "Test" refers to the class containing the "public static void main(String args[])" method.

**How much memory does the process get?**

When initiating the process with the "java MainClassName" command, a new JVM instance is created, allowing us to specify the amount of heap memory to be allocated.

**Thread —----------------------------------------------------------------------------------------------->**

- Thread is know as ligtweight process

OR

Smallest sequence of instructions that are executed by CPU independently.

- And 1 process can have multiple threads.
- When a Process is created, it start with 1 thread and that initial thread know as 'main thread' and from that we can create multiple threads to perform task concurrently.

**MultiThread.java**

```
public class MultithreadingLearning {
  public static void main(String argsll){
      System.out-println("Thread Name:" + Thread.currentThread() - getName()) ;
  }
```

Output: Thread Name: main

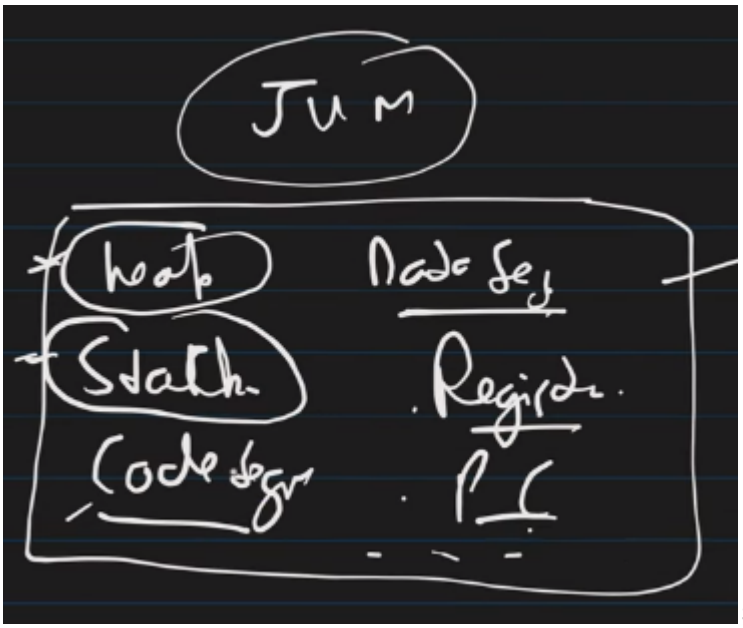
After running above code When I write javac MultiThread.java, it compiles the main file into bytecode and creates a file called MultiThread.class. Then, when I execute java MultiThread (using the .class file), the JVM initiates a new process. This process has its own heap memory, and automatically spawns a thread known as the main thread. We can create additional threads from this main thread to perform tasks concurrently. The bytecode needs to be converted into machine code, which is then executed by the CPU.

Java program with multiple threads. Here's a breakdown:

1. **Compilation**: The javac command compiles the Java source code (MultiThread.java) into bytecode. Bytecode is a low-level representation of the program that is platform-independent.
2. **Execution**: When the java command is used with the name of the compiled class (MultiThread), the Java Virtual Machine (JVM) is invoked. The JVM loads the bytecode (from MultiThread.class) and starts executing it.
3. **JVM and Process**: The JVM creates a new process for executing the Java program. This process has its own memory space, including a heap for dynamic memory allocation.
4. **Main Thread**: When the JVM starts executing the Java program, it automatically creates a main thread. This main thread begins executing the main() method of the MultiThread class.
5. **Additional Threads**: From within the main thread or any other thread created by the program, additional threads can be created using the Thread class or implementing the Runnable interface. These threads allow tasks to be executed concurrently.
6. **Bytecode to Machine Code**: The JVM's Just-In-Time (JIT) compiler converts bytecode into machine code specific to the underlying hardware architecture. This machine code is executed directly by the CPU.

So, the process described accurately outlines the steps involved in compiling and executing a Java program with multiple threads.

When we execute java Main, it triggers the creation of a new process along with a fresh instance of the Java Virtual Machine (JVM). This JVM instance is assigned various resources including heap memory, stack memory, code memory, registers, and more.



s

**How much heap memory is allocated to each process out of the total heap available?**

To specify the amount of memory a process should receive, we use the following command format:

```
java -Xms256m -Xmx2g MainClassName
```

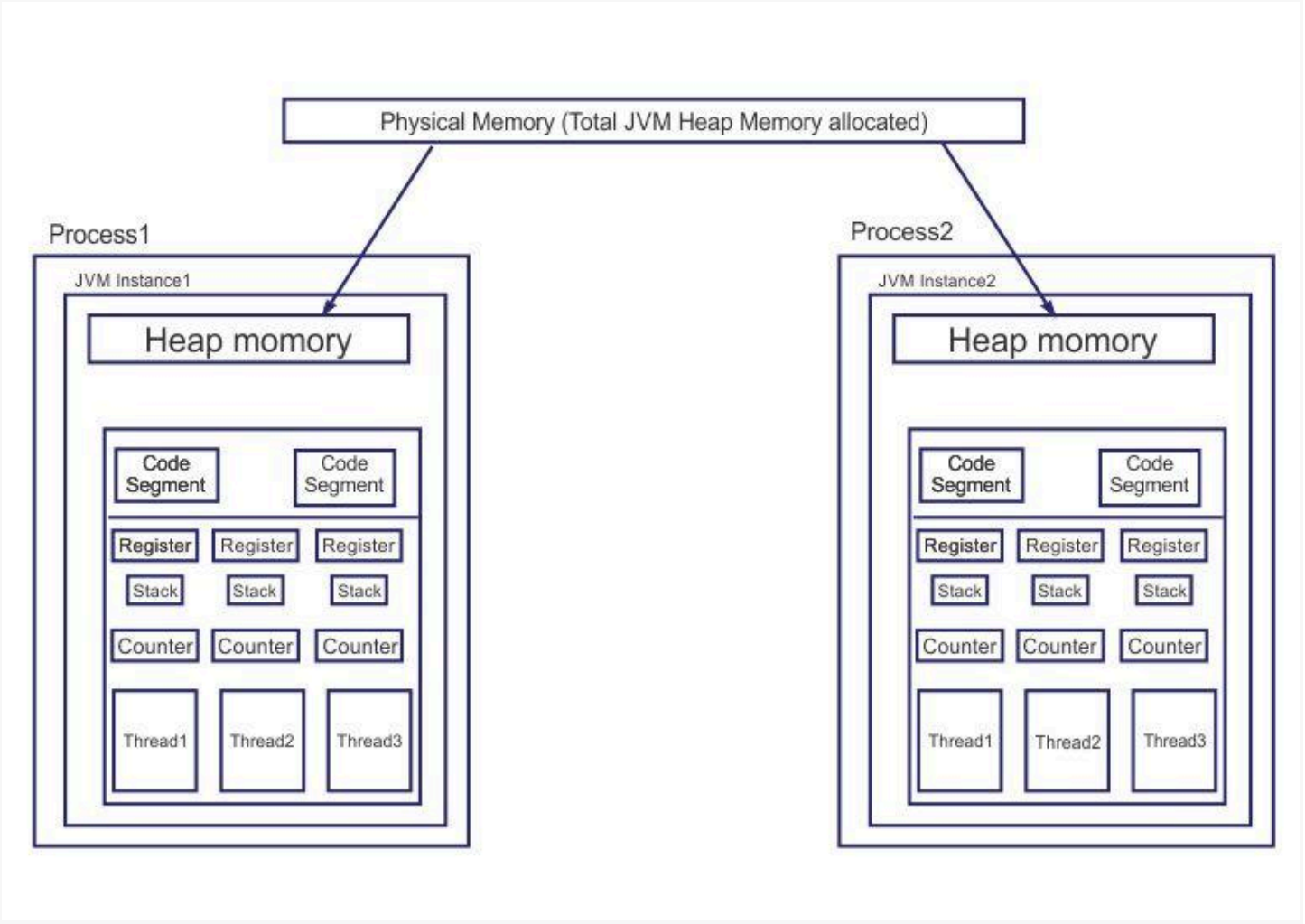
Here's what each parameter means:

- **-Xms<size>**: This sets the initial heap size. In the example provided, 256MB of heap memory is allocated initially.
- **-Xmx<size>**: This sets the maximum heap size that the process can utilize. In the example, 2GB of heap memory is allocated as the maximum. If the process attempts to allocate more memory than this maximum limit, it will encounter an "OutOfMemoryError" exception.

Therefore, by setting these parameters, we can control how much heap memory is assigned to a Java process when it starts up.

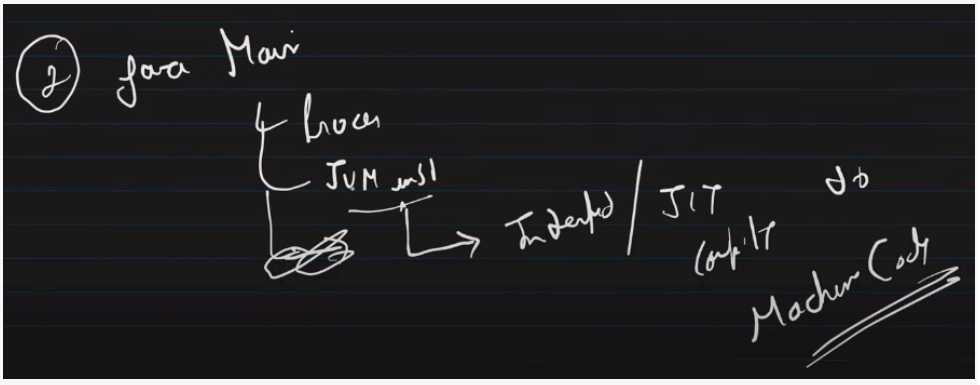
**Let's understand a little bit more about Process and Threads:**

We see three threads made by two data parts. Each thread has its own Register, Stack, and Counter. These are not shared with other threads and stay private to each thread. But, the Code Segments in the process are shared among these three threads, along with the heap memory. So, while Registers, Stack, and Counters are separate for each thread, the Code Segments and heap memory are shared among them.




**Code Segment:**

- This part holds the compiled BYTECODE (like machine code) of the Java Program.
- It's only for reading, can't be modified.
- All threads in the same process share this code segment.



When we execute the **Java Main class** file, it initiates a **process**, and a **JVM instance** is created. The **JVM interprets** the **bytecode** and employs the **JIT Compiler** to convert this bytecode into machine code.

After this process, the machine code segment contains instructions that the CPU can understand. These instructions are stored in our **Code Segment** memory.

**Data Segment:**

- Data Segment holds GLOBAL and STATIC variables.
- It's shared by all threads within the process.
- Threads can both read and change this data.
- Synchronization is needed when multiple threads are involved to prevent conflicts.

**Heap:**

- Objects created at runtime with the "new" keyword are stored in the heap.
- The heap is shared by all threads within the same process but not across different processes.
- Threads can both read from and modify the data in the heap.
- Synchronization is required when multiple threads access the heap simultaneously.

When threads run and create objects, those objects are stored and processed in the heap memory. This memory is accessible across all threads within the same process. However, if there are two separate processes, their heap memories do not share data because each heap points to different references in the JVM heap memory.

**Stack:**

- Every thread possesses its individual stack.
- The stack manages method calls and local variables.

**Register:**

- The JIT (Just-in-time) compiler optimizes machine code by converting bytecode, utilizing registers.
- Registers also assist in context switching between threads.
- Each thread maintains its individual set of registers.

**Counter:**

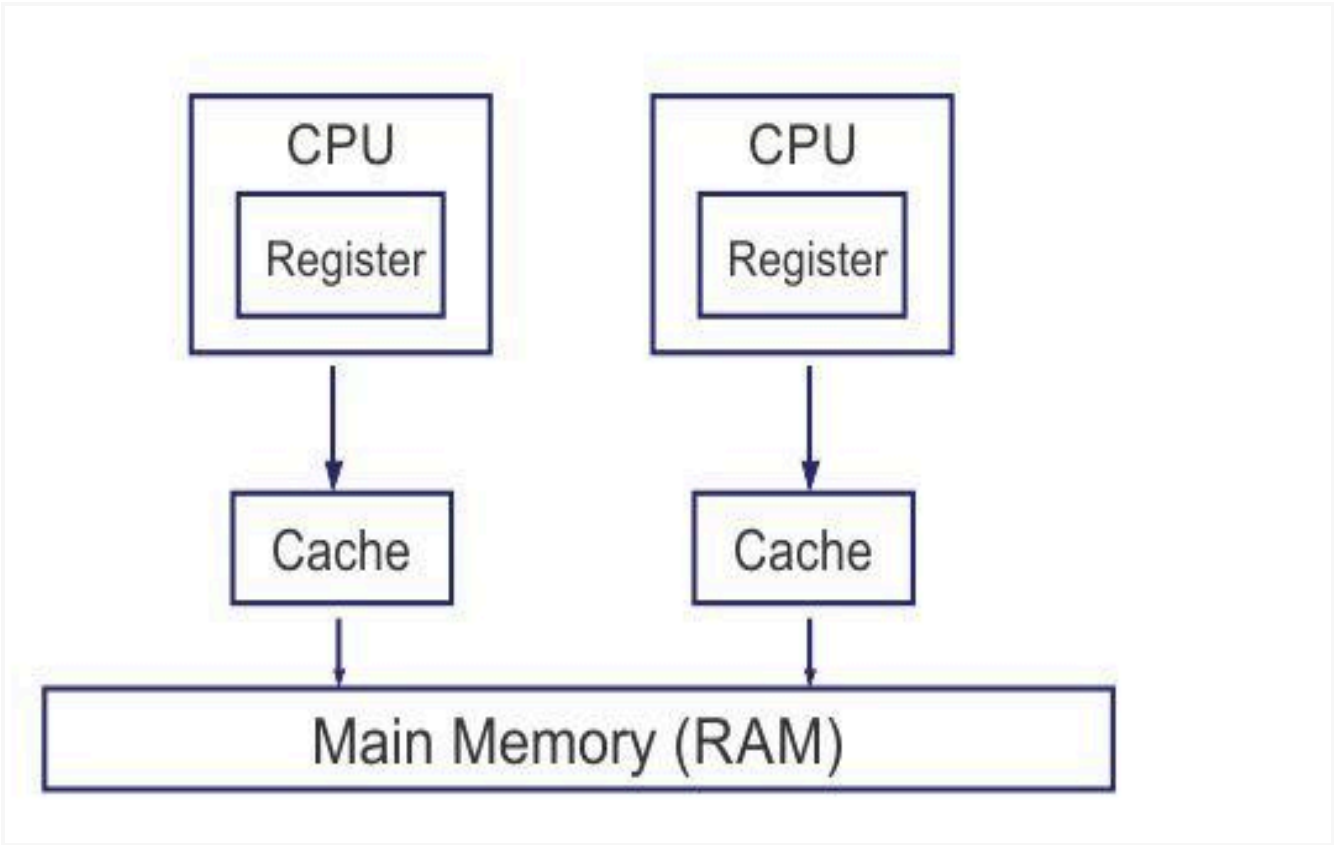
- Known as the Program Counter, it points to the instruction being executed.
- Incremented after successfully executing each instruction.
- All of these functions are overseen by the JVM.

The counter keeps track of the instructions executed by each thread from the code segment. If the execution is successful, it increments the count. It holds a reference to the machine code of those instructions executed by the JVM from the code segment.
Now, to execute the program, the CPU takes charge, not the JVM. Initially, the main thread begins by loading the machine code into its own register, which then gets assigned to the CPU register. The CPU register functions similarly to the thread register. The operating system scheduler manages thread assignment to the CPU, with its own JVM scheduler handling this task.

As the CPU runs the thread, it stores intermediate code in its register. When the thread begins, the counter observes the sequence of machine code assigned to it, helping to save the reference to the machine code in its register. As the CPU executes and the thread progresses, context switching occurs. This involves running the main thread until an instruction is processed and a certain percentage is achieved, then another thread is assigned by the OS scheduler.

Progress data is stored in the thread register before starting work on the new thread. This cycle continues until the program ends. The data is retrieved from the thread register for execution and stored again in the same register; this register acts as a form of local memory for the thread.

If multiple CPU cores are available, threads can run in parallel, following the same process as described above.



**Definition of Multithreading: by using the above point**

Multithreading is a programming technique that allows a program to execute multiple tasks concurrently. It enables multiple threads to share the same resources, such as memory space, while still being able to perform tasks independently.

## Advantages and Disadvantages of Multithreading:

**Advantages:**

- Enhanced performance through task parallelism
- Improved responsiveness
- Efficient resource sharing

**Challenges:**

- Concurrency problems like deadlock and data inconsistency
- Overhead from synchronization
- Testing and debugging complexities

**Multitasking vs Multithreading**

Many interviewers often ask about the difference between multitasking and multithreading. Multitasking is when we run multiple programs together, while multithreading is when one program can do multiple things at once. In multithreading, each task within the program has its own threads that can share resources like memory, but tasks don't share resources with each other.

**Analogy of Multitasking vs Multithreading**

When we talk about multitasking, it's like doing many things at once. For example, if you're reading a book while listening to music, that's multitasking.

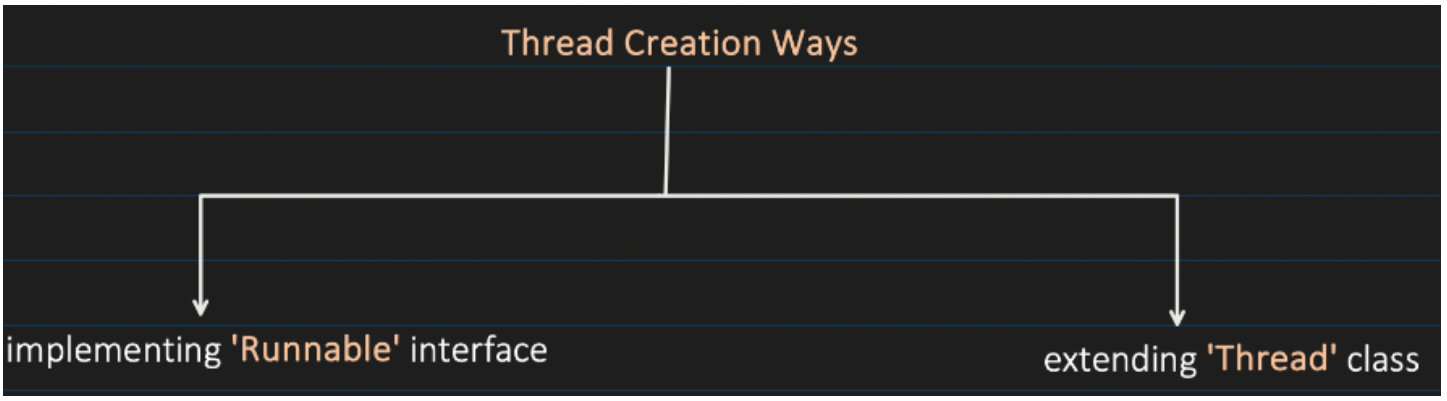
On the other hand, multithreading is like having different parts of a task handled by different people in a group. Each person in the group is like a thread, and they work together to get things done faster.

In computer terms, multitasking means running many programs at the same time, while multithreading means having different parts of a program run at the same time to speed things up.

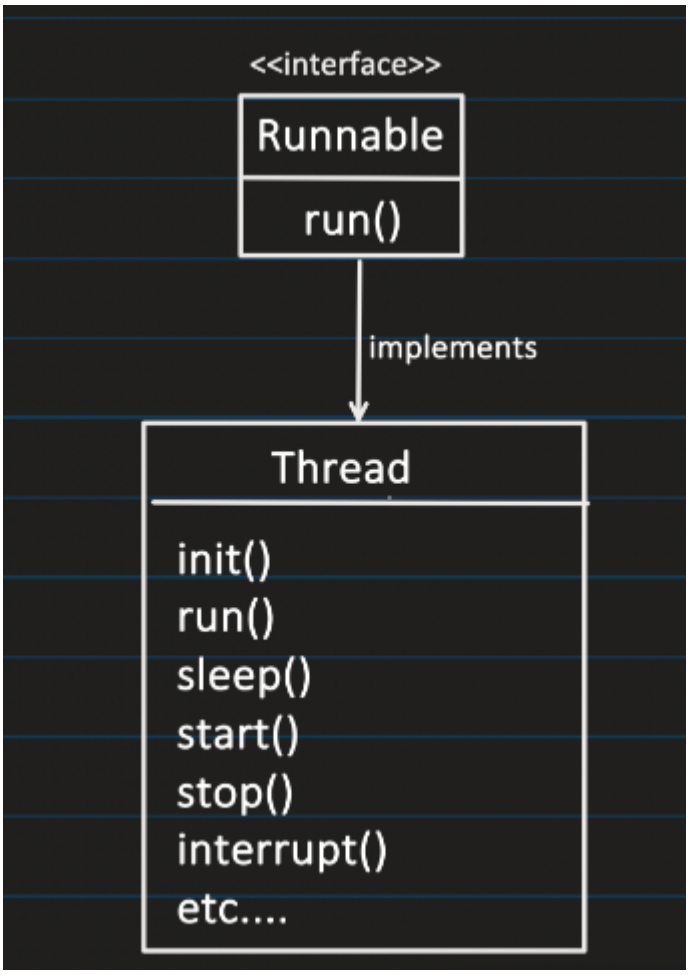
==========

**Thread Creation, Thread Lifecycle and Inter-Thread Communication**

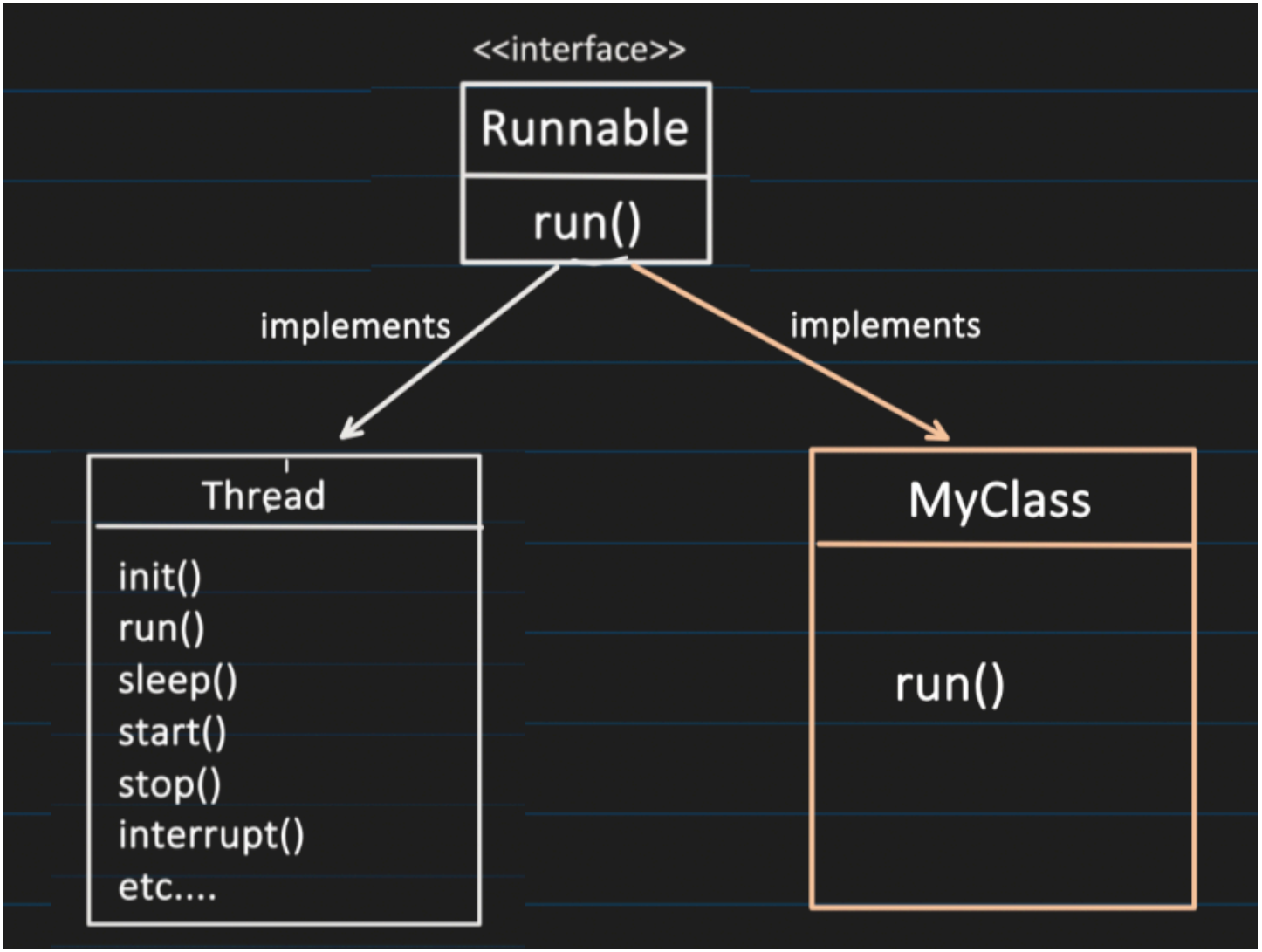
==========




You can make a thread in two ways: by using the Runnable way or by making a new class that's a thread.

| | |
|---|---|
|  | Runnable is a functional interface. It only has one method called run, and it is not have any method to related to threads. On the other hand, the Thread class implements the Runnable interface.Thread has all the methods like init(), run(), sleep(), start(), stop(), interrupt(), and more. |

## Implementing 'Runnable' interface:




Step1: Create a Runnable Object

- Create a class that implements 'Runnable' interface.
- Implement the 'run()' method to tell the task which thread has to do.

```
public class MultithreadingLearning implements Runnable{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Step2: Start the thread

- Create an instance of class that implement 'Runnable'.
- Pass the Runnable object to the Thread Constructor.
- Start the thread.

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

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultithreadingLearning runnableObj = new MultithreadingLearning();
        Thread thread = new Thread(runnableObj);
        thread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```

**Process to create and run thread - Creating and Running a Thread:**

**Step 1: Define Task**

- Create a class called ThreadTask.
- Implement the Runnable interface, overriding the run method.
- Write the logic or code inside the run method. This defines what the thread will execute, without actually creating the thread.

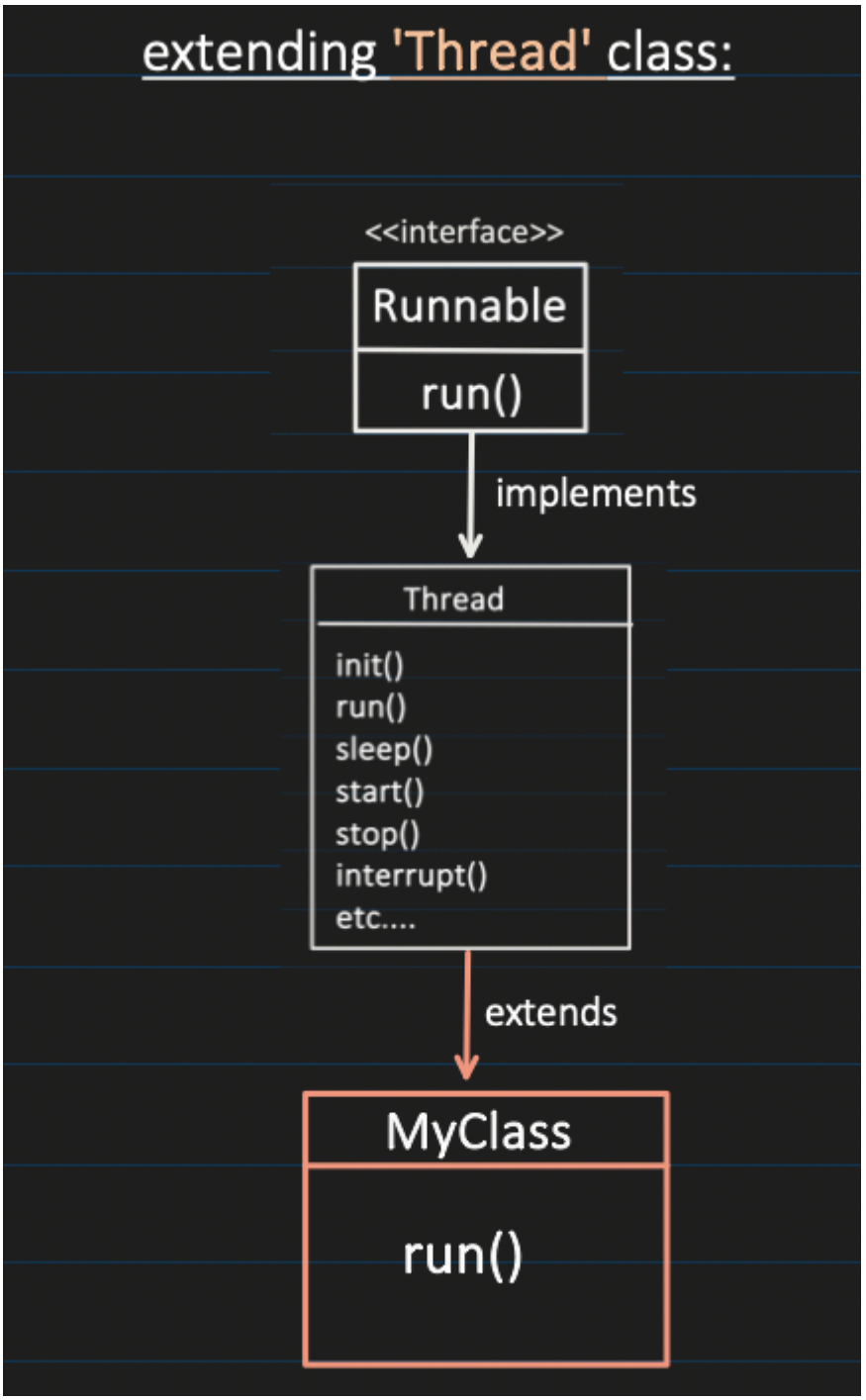
**Step 2: Create Thread**

- Wherever a thread is needed, instantiate ThreadTask using the new keyword.
- Then create an instance of the Thread class and pass the ThreadTask object.
- Since all thread-related methods are in the Thread class, this step is necessary.

**Step 3: Run Thread**

- After creating the Thread instance, call the start method to execute the thread.
- Internally, the start method invokes the run method of the ThreadTask class, executing the defined logic.

**Process Summary** :First, instantiate the ThreadTask class to define the task logic. Next, create an instance of the Thread class and pass the task object to it. The Thread class stores this task object in its target instance, using a constructor that takes a Runnable type. When the start method is invoked, the thread executes the task by calling the run method of the target instance




Step1: Create a Thread Subclass

- Create a class that extends 'Thread' class.
- Override the 'run()' method to tell the task which thread has to do.

```java
public class MultithreadingLearning extends Thread{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Step2: Initiate and Start the thread

- Create an instance of the subclass.
- Call the start() method to begin the execution.

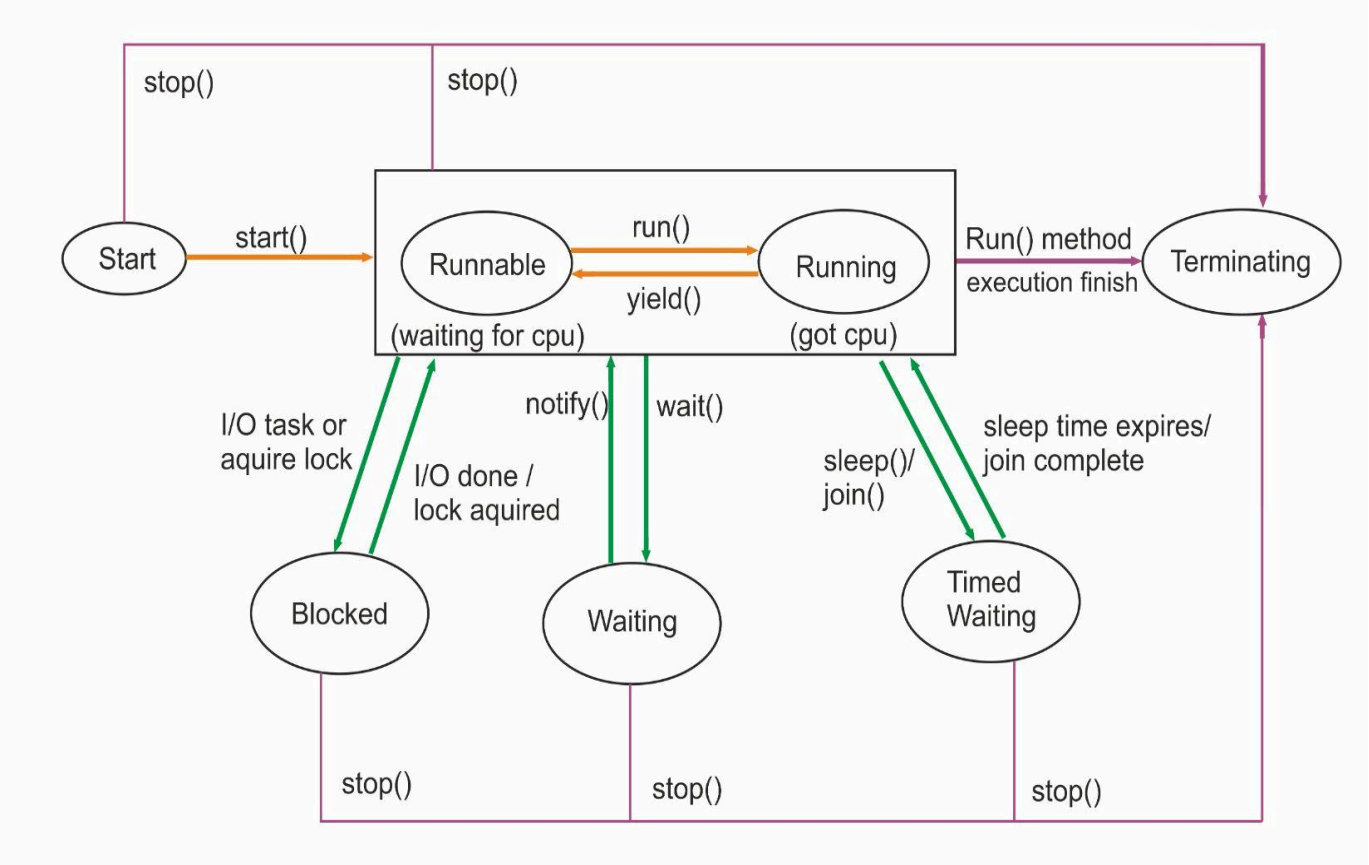
```
public class Main {
    public static void main(String args[]){

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultithreadingLearning myThread = new MultithreadingLearning();
        myThread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```

When we create a subclass of the Thread class, we inherit all the methods of the Thread class. In the subclass, we typically override the run method and define our task logic there. Now, when we want to create a thread in the main class, we don't need to explicitly create an instance of the Thread class and pass the task object. Instead, we can directly create an object of the subclass and call the start method on it. This will automatically invoke the run method implemented in the subclass, effectively starting the thread. So, we simply call subClassObject.start() to begin the thread execution.

**Thread LifeCycle**




**Thread Lifecycle:**

- First, we create a new thread, which is in the "New" state.
- When we start the thread, it becomes "Runnable" and waits for CPU time.
- Once it gets CPU time, it enters the "Running" state.
- After finishing its task or when a context switch occurs, it goes back to the queue and waits to run again.
- When it finishes its task completely, it's "Terminated" and can't start again.

**Blocked State:**

- After starting, a thread can enter a "Runnable" state or be "Blocked".
- It might get blocked due to I/O tasks like fetching data from a server.
- It can also get blocked if someone else acquires a lock on a resource it needs.
- Once the I/O operation is done or the lock is released, it goes back to the "Runnable" state.

**Waiting State:**

- After starting, if a thread directly calls the "wait" method, it enters the "Waiting" state.
- To bring it back to "Runnable", we need to call the "notify" method after it finishes its task.

**Timed Waiting State:**

- After starting, if a thread calls the "sleep" method and sets a specific time, it enters the "Timed Waiting" state.
- After the sleep time finishes, it automatically goes back to the "Runnable" state.

**Terminating a Thread:**

- We can terminate a thread at any point using the "stop()" method, whether it's running, blocked, or waiting. Even if it's just created, we can still stop it.

| Lifecycle State | Description |
|---|---|
| New | • Thread has been created but not started<br>• Its just an Object in memory |
| Runnable | • Thread is ready to run.<br>• Waiting for CPU time. |
| Running | • When thread start executing its code. |
| Blocked | Different scenarios where runnable thread goes into the Blocking state:<br>- I/O : like reading from a file or database.<br>- Lock aquired: if thread want to lock on a resource which is locked by other thread, it has to wait.<br>• Releases all the MONITOR LOCKS |
| Waiting | • Thread goes into this state when we call the wait() method, makes it non runnable.<br>• Its goes back to runnable, once we call notify() or notifyAll() method.<br>• Releases all the MONITOR LOCKS |
| Timed Waiting | • Thread waits for specific period of time and comes back to runnable state, after specific conditions met.<br>like sleep(), join()<br>• Do not Releases any MONITOR LOCKS |
| Terminated | • Life of thread is completed, it can not be started back again. |

**MONITOR LOCK:**

It helps to make sure that only 1 thread goes inside the particular section of code (a synchronized block or method)

```
public class MonitorLockExample {
  public synchronized void task1() {
//do something
      try {
          System.out.println("inside task1");
          Thread.sleep( 10000);
      } catch (Exception e) {
//exception handling here
      }
  }
  public void task2() {
      System.out.println("task2, but before synchronized");
      synchronized (this) {
          System.out.println("task2, inside synchronized");
      }
  }
  public void task3() {
      System.out.println("task3");
  }
}
```

```
public static void main(String args[]) {
  MonitorLockExample obj = new MonitorLockExample();
  Thread t1 = new Thread(() -> {obj.task1(); });
  Thread t2 = new Thread(() -> {obj.task2(); });
  Thread t3 = new Thread(() -> {obj.task3(); });
  t1.start();
  t2.start();
  t3.start();
}
```

==========

## Java Reflection - Coming soon

==========

**- What is Reflection**
**- What is "Class" class which JVM creates at runtime**
**- How to Reflect Classes and access its metadata**
**- How to Reflect Methods and access its metadata**
**- How to Invoke Methods using Reflection**
**- How to Reflect Fields and access its metadata**
**- How to access and change the value of Public field**
**- How to access and change the value of Private fields.**
**- How to Reflect Constructor and access their metadata.**
**- How to access and invoke private constructor using reflection.**
**- How Reflection breaks Singleton design pattern and how to resolve.**

## A. What is Reflection?

This is used to examine the Classes, Methods, Fields, Interfaces at runtime and also possible to change the behaviour of the Class too.

**For example:**

- What all methods present in the class.~
- What all fields present in the class.
- What is the return type of the method?
- What is the Modifier of the Class
- What all interfaces class has implemented
- Change the value of the public and private fields of the Class etc.....

==========

# Java Annotations - Coming soon

==========

**- What is Annotation**

**- Pre-defined Annotation:**

- Deprecated
- Override
- SupressWarnings
- FunctionalInterface
- SafeVarargs and Heap Pollution issue
- Target (Meta-annotation)
- Retention (Meta-annotation)
- Documented (Meta-annotation)
- Inherited  (Meta-annotation)
- Repeatable (Meta-annotation)

**- Custom/UserDefined Annotation**