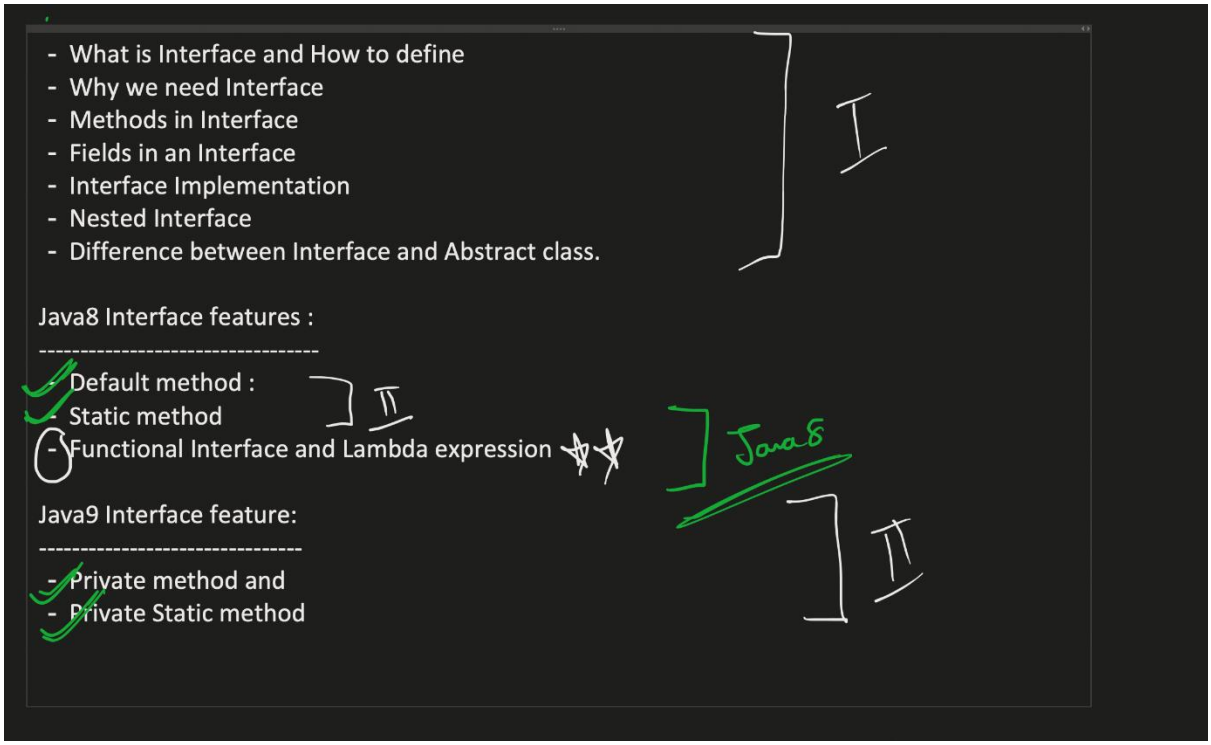
# Table of Contents

# Java: Interface in Depth




**What is Interface?**

Interface is something which helps 2 system to interact with each other, without one system has to know the details of other.

Or in simple term I can say, it helps to achieve ABSTRACTION.

**How to define the interface?**

Interface declaration consist of
- Modifiers
- "interface" keyword
- Interface Name
- Comma separated list of parent interfaces
- Body

Only ***Public*** and ***Default*** Modifiers are allowed (***Protected*** and ***private*** are not allowed)

```
public interface Bird {

    no usages
    public void fly();
}
```

```
interface Bird {

    no usages
    public void fly();
}
```

Comma separated list of parent interfaces (it can extend from ***Class***) Example:

```
public interface NonFlyingBirds extends Bird, LivingThings{

    no usages
    public void canRun();
}
```

**Why we need Interface?**

---

1. **Abstraction**:

---

Using interface, we can achieve full Abstraction means, we can define WHAT class must do, but not HOW it will do

```
public interface Bird {

    no usages
    public void fly();
}
```

```
public class Eagle implements Bird{
    no usages
    @Override
    public void fly() {
        //the complex process of flying take place here
    }
}
```

2. Polymorphism:

- Interface can be used as a Data Type.

- we can not create the object of an interface, but it can hold the reference of all the classes which implements it. And at runtime, it decide which method need to be invoked.

```java
public interface Bird {

    2 usages  2 implementations
    public void fly();
}
```

```java
public class Main {

    public static void main(String args[]){

        Bird birdObject1 = new Eagle();
        Bird birdObject2 = new Hen();

        birdObject1.fly();
        birdObject2.fly();
    }
}
```

```java
public class Eagle implements Bird{

    2 usages
    @Override
    public void fly() {
        System.out.println("Eagle Fly Implementation");
    }
}
```

```java
public class Hen implements Bird{

    2 usages
    @Override
    public void fly() {
        System.out.println("Hen Fly Implementation");
    }
}
```

3. Multiple Inheritance:

In Java Multiple inheritance is possible only through Interface only.

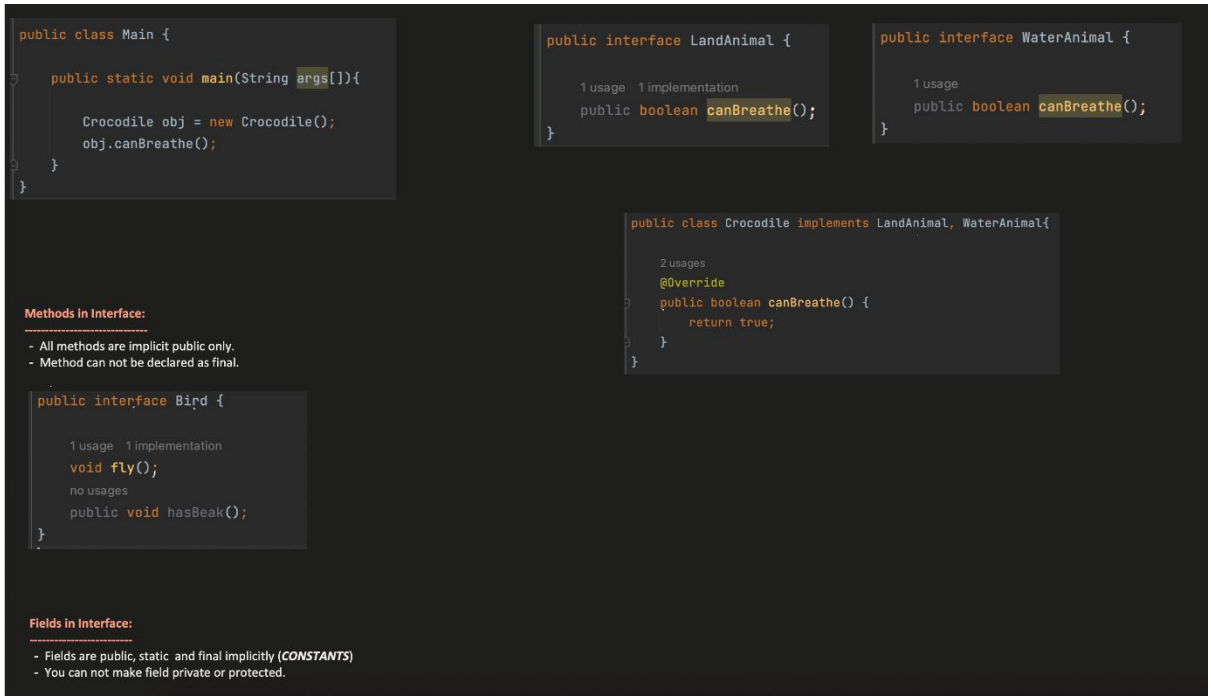
Diamond problem:

```java
public class Main {

    public static void main(String args[]){

        Crocodile obj = new Crocodile();
        obj.canBreathe();
    }
}
```

```java
public class WaterAnimal {

    no usages
    public boolean canBreathe(){
        return true;
    }
}
```

```java
public class LandAnimal {

    1 usage
    public boolean canBreathe(){
        return true;
    }
}
```

```java
public class Crocodile extends LandAnimal, WaterAnimal{
}
```


public class Main {
public static void main(String args[]){
Crocodile obj = new Crocodile();
obj.canBreathe();
}
}
public interface LandAnimal {
1 usage 1 implementation
public boolean canBreathe();
}
public interface WaterAnimal {
1 usage
public boolean canBreathe();
}
public class Crocodile implements LandAnimal, WaterAnimal{
2 usages
@Override
public boolean canBreathe() {
return true;
}
}
Methods in Interface:
- All methods are implicit public only.
- Method can not be declared as final.
public interface Bird {
1 usage 1 implementation
void fly();
no usages
public void hasBeak();
}
Fields in Interface:
- Fields are public, static and final implicitly (CONSTANTS)
- You can not make field private or protected.

```java
public interface Bird {
    no usages
    int MAX_HEIGHT_IN_FEET = 2000;
}
```

==

```java
public interface Bird {
    no usages
    public static final int MAX_HEIGHT_IN_FEET = 2000;
}
```

**Interface Implementation:**

---

- Overriding method can not have more restrict access specifiers.
- Concrete class must override all the methods declared in the interface.
- Abstract classes are not forced to override all the methods.
- A class can implement from multiple interfaces.

```java
public interface Bird {

    2 usages   2 implementations
    public void fly();
}
```

```java
public class Eagle implements Bird{

    2 usages
    @Override
    public void fly() {
        System.out.println("Eagle Fly Implementation");
    }
}
```

```java
public class Eagle implements Bird{
    1 usage
    @Override
    protected void fly() {

        //do something
    }
}
```

**Example of Abstract class implementation of Interface:**

```java
public interface Bird {
    no usages  1 implementation
    public void canFly();
    no usages  1 implementation
    public void noOfLegs();
}
```

```java
public abstract class Eagle implements Bird{
    no usages
    @Override
    public void canFly() {
        //Implementation goes here
    }

    no usages
    public abstract void beakLength();
}
```

```java
public class WhiteEagle extends Eagle{

    no usages
    @Override
    public void noOfLegs() {
        //implement interface method
    }

    no usages
    @Override
    public void beakLength() {
        //implementing abstract class method
    }
}
```

## Nested Interface:

- Nested Interface declared withing another Interface.
- Nested Interface declared within a Class.

Generally its is used to group, logical related interfaced. And Nested interface

Rules:

- A nested interface declared withing an interface must be public.
- A nested interface declared within a class can have any access modifier.
- When you implement outer interface, inner interface implementation is not required and vice versa.

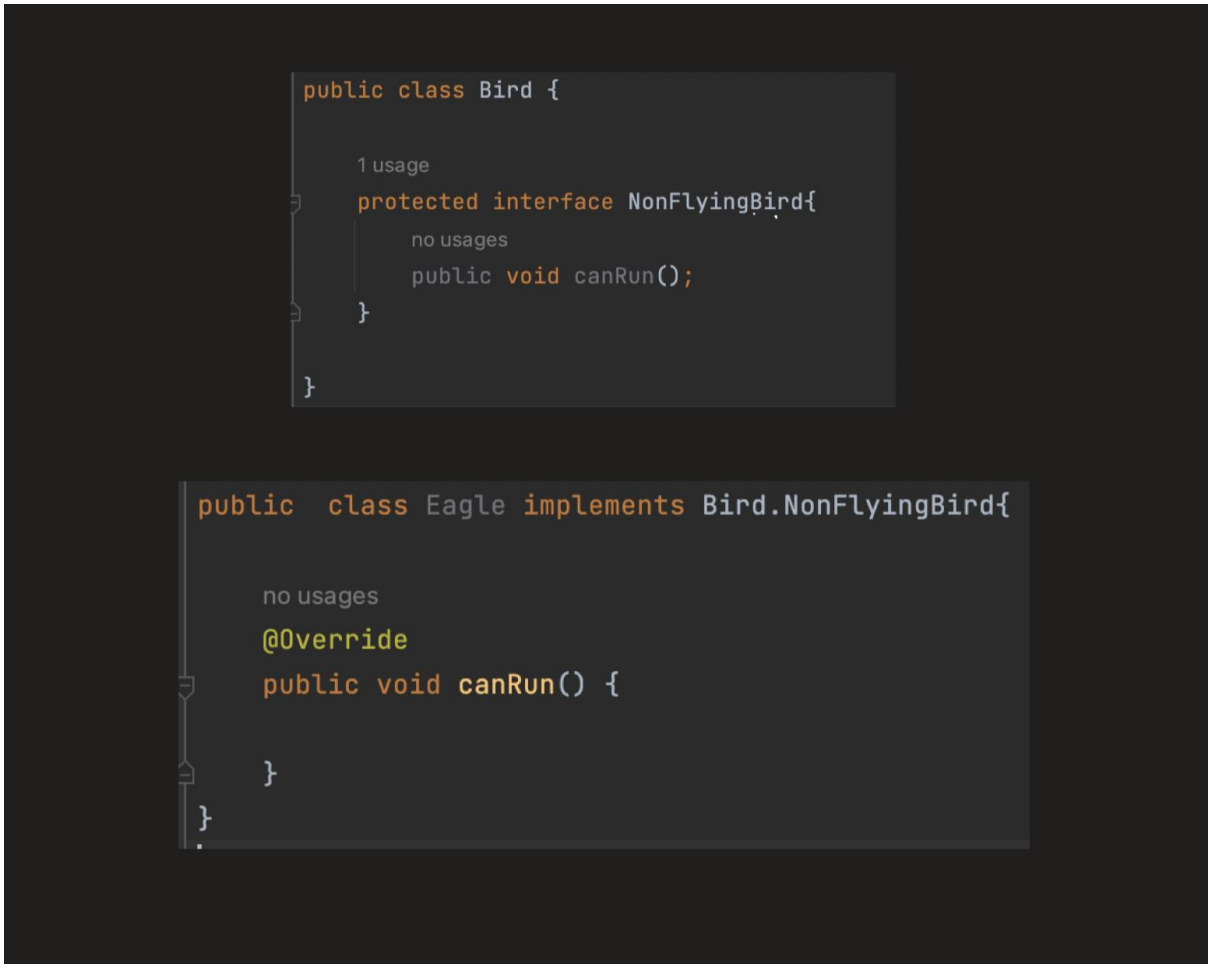
```
public interface Bird {

    no usages   1 implementation
    public void canFly();

    no usages
    public interface NonFlyingBird{

        no usages
        public void canRun();
    }
}
```

```
public  class Eagle implements Bird{
    no usages
    @Override
    public void canFly() {
        //Implementation goes here
    }
}
```

```
public  class Eagle implements Bird.NonFlyingBird{

    no usages
    @Override
    public void canRun() {

    }
}
```

```
public  class Eagle implements Bird, Bird.NonFlyingBird{

    1 usage
    @Override
    public void canRun() {

    }

    no usages
    @Override
    public void canFly() {

    }
}
```

```
public class Main {

    public static void main(String args[]){

        Bird.NonFlyingBird obj = new Eagle();
        obj.canRun();
    }
}
```



```java
public class Bird {

    1 usage
    protected interface NonFlyingBird{
        no usages
        public void canRun();
    }

}
```

```java
public  class Eagle implements Bird.NonFlyingBird{

    no usages
    @Override
    public void canRun() {

    }
}
```

## Interface Vs Abstract Class:

| S.No. | Abstract Class | Interface |
|---|---|---|
| 1. | Keyword used here is "abstract" | Keyword used here is "interface" |
| 2. | Child Classes need to use keyword "extends" | Child Classes need to use keyword "implements" |
| 3. | It can have both abstract and non abstract method | It can have only Abstract method (from Java8 onwards it can have default ,static and private method too, where we can provide implementation) |
| 4. | It can Extend from Another class and multiple interfaces | It can only extends from other interfaces |
| 5. | Variables can be static, non static, final , non final etc. | Variable are by default CONSTANTS |
| 6. | Variables and Methods can be private, protected, public, default. | Variable and Methods are by default public (in Java9 , private method is supported) |
| 7. | Multiple Inheritance is not Supported | Multiple Inheritance supported with this in Java |
| 8. | It can provide the implementation of the interface | It can not provide implementation of any other interface or abstract class. |
| 9. | It can have Constructor | It can not have Constructor |
| 10. | To declare the method abstract, we have to use "abstract" keyword and it can be protected, public, default. | No need for any keyword to make method abstract. And be default its public. |

**JAVA 8 and 9 FEATURES:**

1. **Default Method(Java8):**

- Before Java8, interface can have only Abstract method. And all child classes has to provide abstract method implementation.

```
public interface Bird {

    public void canFly();
}
```

Any new method added in interface, means need to change in all its implementation

```
public interface Bird {

    public void canFly();
    public int getMinimumFlyHeight();
}
```

```
public class Eagle implements Bird{

    @Override
    public void canFly() {
        //eagle fly implementation
    }
}
```

```
public class Sparrow implements Bird{

    @Override
    public void canFly() {
        //sparrow fly logic
    }
}
```

```
public class Eagle implements Bird{

    @Override
    public void canFly() {
        //eagle fly implementation
    }

    @Override
    public int getMinimumFlyHeight() {
        return 100;
    }
}
```

```
public class Sparrow implements Bird{

    @Override
    public void canFly() {
        //sparrow fly logic
    }

    @Override
    public int getMinimumFlyHeight() {
        return 100;
    }
}
```

**Using Default Method:**

```
public class Main {

    public static void main(String args[]){

        Eagle eagleObj = new Eagle();
        eagleObj.getMinimumFlyHeight();
    }
}
```

```
public interface Bird {

    public void canFly();

    default int getMinimumFlyHeight(){
        return 100;
    }
}
```

```
public class Eagle implements Bird{

    @Override
    public void canFly() {
        //eagle fly implementation
    }
}
```

```
public class Sparrow implements Bird{

    @Override
    public void canFly() {
        //sparrow fly logic
    }
}
```


**Why Default method was introduced:**

- To add functionality in existing Legacy Interface we need to use Default method. Example *steam()* method in Collection.

**Default and Multiple Inheritance, how to handle:**

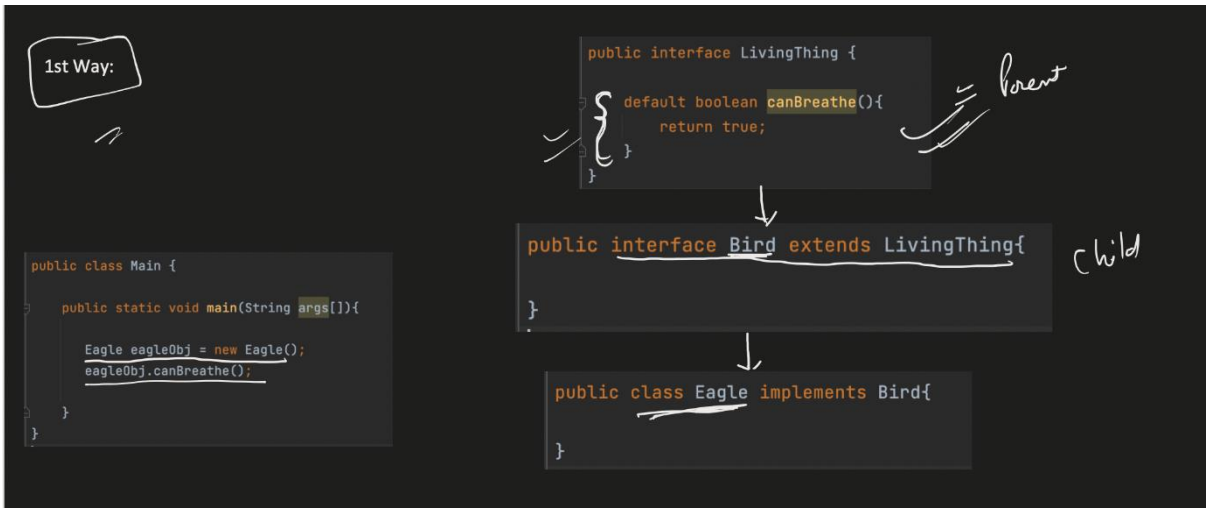
```
public interface Bird {

    default boolean canBreathe(){
        return true;
    }
}
```

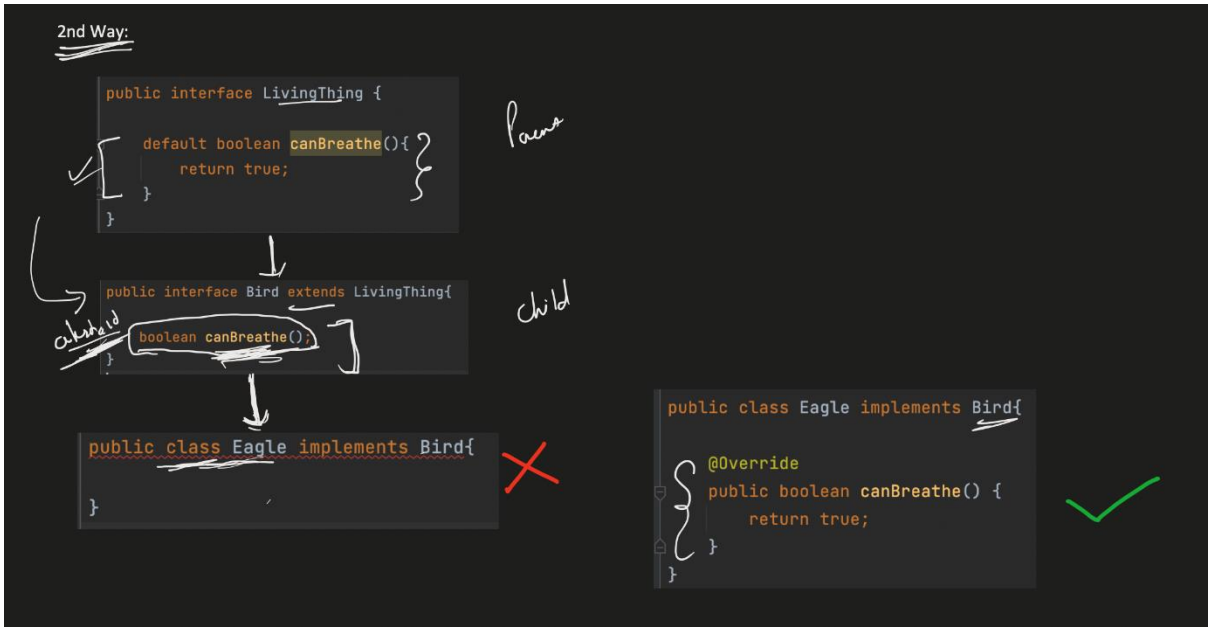
```
public interface LivingThing {

    default boolean canBreathe(){
        return true;
    }
}
```

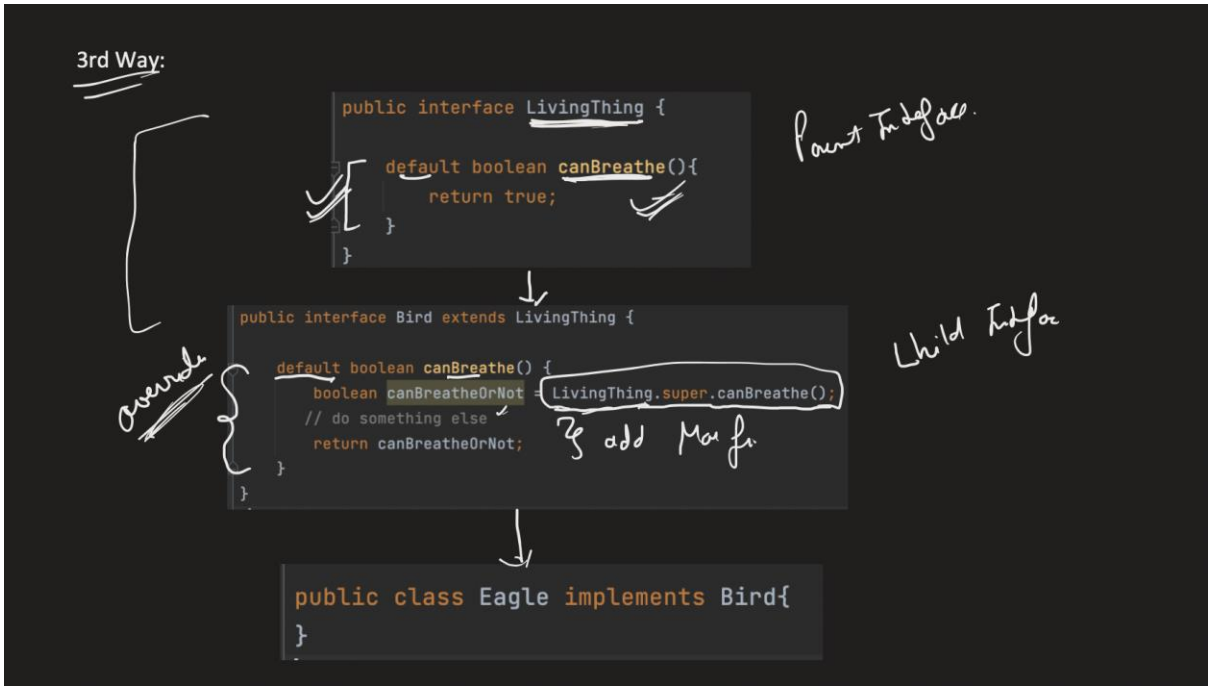
```
public class Eagle implements Bird, LivingThing{

}
```

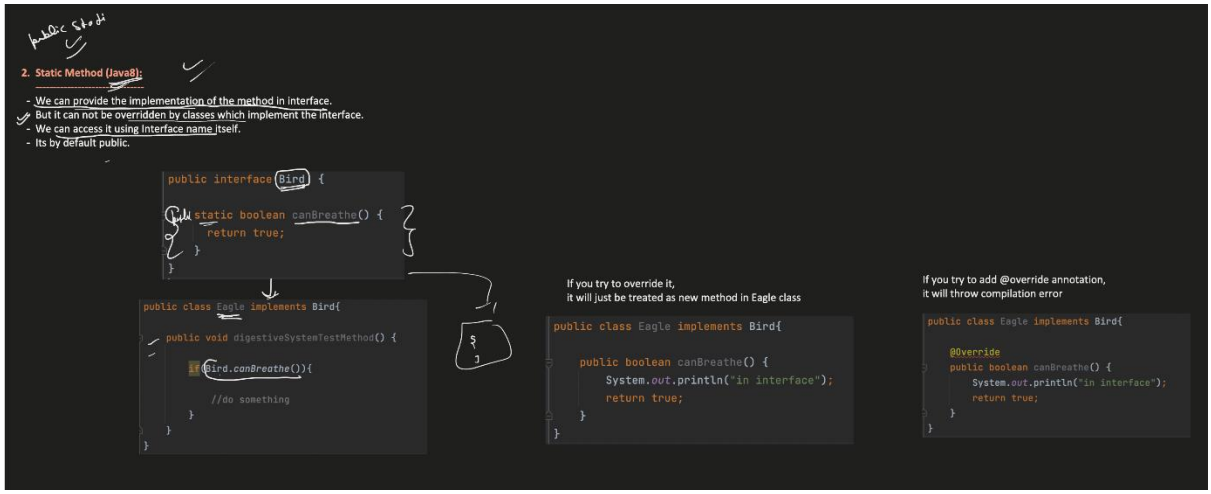
```
public class Eagle implements Bird, LivingThing{

    public boolean canBreathe(){
        return true;
    }
}
```

**How to extend interface, that contains Default Method:**


1st Way:
public interface LivingThing {
default boolean canBreathe(){
return true;
}
}
Parent
public interface Bird extends LivingThing{
}
Child
public class Main {
public static void main(String args[]){
Eagle eagleObj = new Eagle();
eagleObj.canBreathe();
}
}
public class Eagle implements Bird{
}


2nd Way:
public interface LivingThing {
default boolean canBreathe(){
return true;
}
}
Parent
public interface Bird extends LivingThing{
boolean canBreathe();
}
Child
public class Eagle implements Bird{
}
public class Eagle implements Bird{
@Override
public boolean canBreathe() {
return true;
}
}

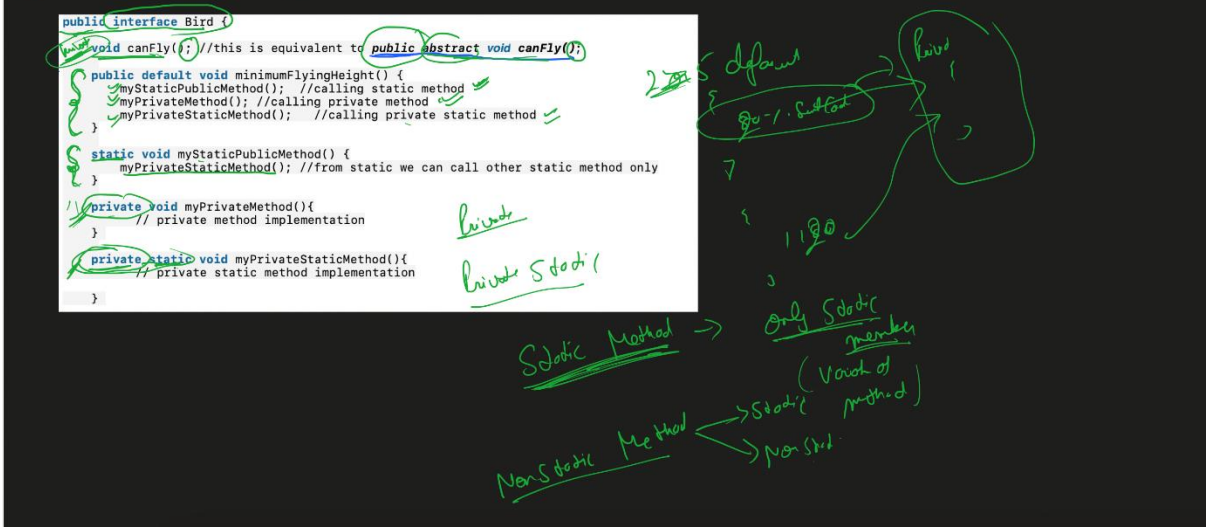

3rd Way:
public interface LivingThing {
default boolean canBreathe(){
return true;
}
}
public interface Bird extends LivingThing {
default boolean canBreathe() {
boolean canBreatheOrNot = LivingThing.super.canBreathe();
// do something else
return canBreatheOrNot;
}
}
public class Eagle implements Bird{
}


2. Static Method (Java8):
- We can provide the implementation of the method in interface.
- But it can not be overridden by classes which implement the interface.
- We can access it using Interface name itself.
- Its by default public.
public interface Bird {
static boolean canBreathe() {
return true;
}
}
public class Eagle implements Bird{
public void digestiveSystemTestMethod() {
if(Bird.canBreathe()){
//do something
}
}
}
If you try to override it,
it will just be treated as new method in Eagle class
public class Eagle implements Bird{
public boolean canBreathe() {
System.out.println("in interface");
return true;
}
}
If you try to add @override annotation,
it will throw compilation error
public class Eagle implements Bird{
@Override
public boolean canBreathe() {
System.out.println("in interface");
return true;
}
}

### 3. Private Method and Private Static method (Java9):

- We can provide the implementation of method but as a private access modifier in interface.
- It brings more readability of the code. For example if multiple default method share some code, that this can help.
- It can be defined as static and non-static.
- From Static method, we can call only private static interface method.
- Private static method, can be called from both static and non static method.
- Private interface method can not be abstract. Means we have to provide the definition.
- It can be used inside of the particular interface only.

```
public interface Bird {

    void canFly(); //this is equivalent to public abstract void canFly();

    public default void minimumFlyingHeight() {
        myStaticPublicMethod();  //calling static method
        myPrivateMethod(); //calling private method
        myPrivateStaticMethod();   //calling private static method
    }

    static void myStaticPublicMethod() {
        myPrivateStaticMethod(); //from static we can call other static method only
    }

    private void myPrivateMethod(){
        // private method implementation
    }

    private static void myPrivateStaticMethod(){
        // private static method implementation
    }
```



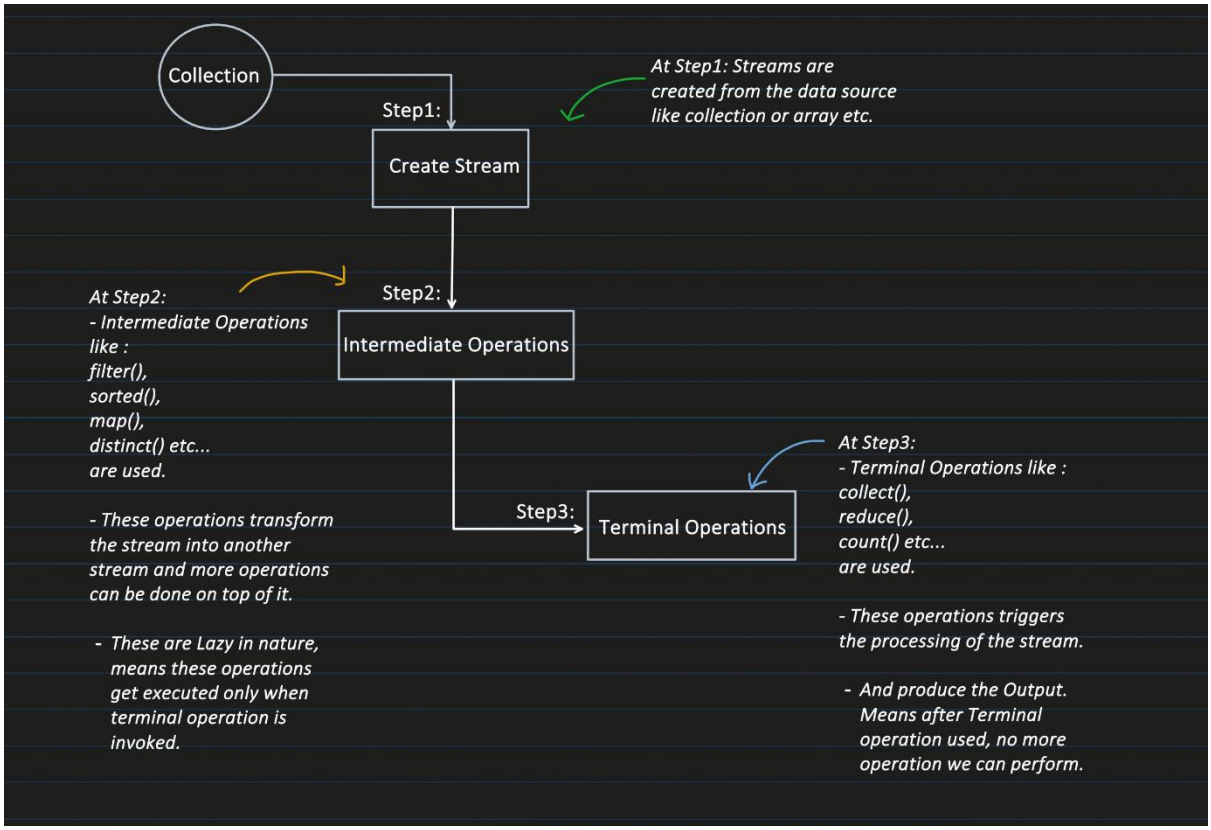
# STREAMS in Java8

## Stream (Java8)

Sunday, 1 October 2023 12:33 PM

**What is Stream?**

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



For Example:

```java
public class StreamExample {

    public static void main(String args[]){

        List<Integer> salaryList = new ArrayList<>();
        salaryList.add(3000);
        salaryList.add(4100);
        salaryList.add(9000);
        salaryList.add(1000);
        salaryList.add(3500);

        int count = 0;
        for(Integer salary : salaryList){
            if(salary > 3000){
                count++;
            }
        }

        System.out.println("Total Employee with salary > 3000: " + count);
    }
}
```

using Stream

```java
public class StreamExample {

    public static void main(String args[]){

        List<Integer> salaryList = new ArrayList<>();
        salaryList.add(3000);
        salaryList.add(4100);
        salaryList.add(9000);
        salaryList.add(1000);
        salaryList.add(3500);

        long output = salaryList.stream().filter((Integer sal) -> sal>3000).count();
        System.out.println("Total Employee with salary > 3000: " + output);
    }
}
```

Output:

```
Total Employee with salary > 3000: 3
```

## Different ways to create a stream:

1. From Collection:

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

2. From Array:

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

3. From Static Method:

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

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 Operations:

We can chain multiple Intermediate operations together to perform more complex processing before applying terminal operation to produce the result.

| No. | Intermediate Operation | Description |
|---|---|---|
| 1. | filter(Predicate<T> predicate)<br>`Stream<String> nameStream = Stream.of( ...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING");`<br>`Stream<String> filteredStream = nameStream.filter((String name) -> name.length() <= 3);`<br>`List<String> filteredNameList = filteredStream.collect(Collectors.toList());`<br>`//OUTPUT: HOW, ARE, YOU` | Filters the element. |
| 2. | map(Function<T, R> mapper)<br>`Stream<String> nameStream = Stream.of( ...values: "HELLO", "EVERYBODY", "HOW", "ARE", "YOU", "DOING");`<br>`Stream<String> filteredNames = nameStream.map((String name) -> name.toLowerCase());`<br>`//OUTPUT: hello, everybody, how, are, you, doing` | Used to transform each element |

3. 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
```

Used to iterate over each element of the complex collection, and helps to flatten it.

4. distinct()

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

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

Removes duplicate from the stream.

5. sorted()

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

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

Sorts the elements.

```
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
```

| | | |
|---|---|---|
| 6. | peek(Consumer<T> action)<br>`List<Integer> numbers = Arrays.asList(2,1,3,4,6);`<br>`Stream<Integer> numberStream = numbers.stream()`<br>`.filter((Integer val)-> val>2)`<br>`.peek((Integer val) -> System.out.println(val)) //it will print 3, 4, 6`<br>`.map((Integer val) -> -1*val);`<br>`List<Integer> numberList = numberStream.collect(Collectors.toList());` | Helps you to see the intermediate result of the stream which is getting processed. |
| 7. | limit(long maxSize)<br>`List<Integer> numbers = Arrays.asList(2,1,3,4,6);`<br>`Stream<Integer> numberStream = numbers.stream().limit( maxSize: 3);`<br>`List<Integer> numberList = numberStream.collect(Collectors.toList());`<br>`//Output: 2, 1, 3` | Truncate the stream, to have no longer than given maxSize |
| 8. | skip(long n)<br>`List<Integer> numbers = Arrays.asList(2,1,3,4,6);`<br>`Stream<Integer> numberStream = numbers.stream().skip( n: 3);`<br>`List<Integer> numberList = numberStream.collect(Collectors.toList());`<br>`//Output: 4, 6` | Skip the first n elements of the stream. |

| | | |
|---|---|---|
| 9. | mapToInt(ToIntFunction<T> mapper)<br>`List<String> numbers = Arrays.asList("2", "1", "4", "7");`<br>`IntStream numberStream = numbers.stream().mapToInt((String val) -> Integer.parseInt(val));`<br>`int[] numberArray = numberStream.toArray();`<br>`//Output: 2, 1, 4, 7`<br>`int[] numbersArray = {2, 1, 4, 7};`<br>`IntStream numbersStream = Arrays.stream(numbersArray);`<br>`numbersStream.filter((int val) -> val > 2);`<br>`int[] filteredArray = numbersStream.toArray();`<br>`//Output: 4, 7` | helps to work with primitive "int" data types |
| 10. | mapToLong(ToLongFunction<T> mapper)<br>(pls try it out....) | helps to work with primitive "long" data types |
| 11. | mapToDouble(ToDoubleFunction<T> mapper)<br>(pls try it out....) | helps to work with primitive "double" data types |

## Why we call Intermediate operation "Lazy":

```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

```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:

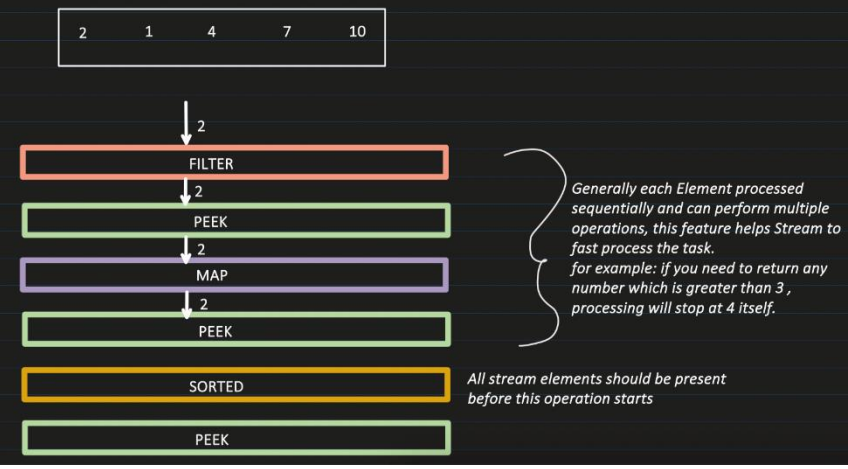
```java
public class StreamExample {

    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("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 ones that produces the result. It triggers the processing of the stream.

| No. | Terminal Operations | Description |
|---|---|---|
| 1. | forEach(Consumer<T> action)<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br>`numbers.stream()`<br>`.filter((Integer val) -> val >=3)`<br>`.forEach((Integer val)-> System.out.println(val));`<br><br>`//OUTPUT: 4, 7, 10` | Perform action on each element of the Stream.<br>**DO NOT Returns any value.** |
| 2. | toArray()<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br><br>`Object[] filteredNumberArrType1 = numbers.stream()`<br>`.filter((Integer val) -> val >=3)`<br>`.toArray();`<br><br>`Integer[] filteredNumberArrType2 = numbers.stream()`<br>`.filter((Integer val) -> val >=3)`<br>`.toArray((int size) -> new Integer[size]);` | Collects the elements of the stream into an Array. |
| 3. | reduce(BinaryOperator<T> accumulator)<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br><br>`Optional<Integer> reducedValue = numbers.stream()`<br>`.reduce((Integer val1, Integer val2) -> val1+val2);`<br><br>`System.out.println(reducedValue.get());`<br>`//output: 24` | does reduction on the elements of the stream. Perform associative aggregation function. |
| 4. | collect(Collector<T, A, R> collector)<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br><br>`List<Integer> filteredNumber = numbers.stream()`<br>`.filter((Integer val) -> val >=3)`<br>`.collect(Collectors.toList());` | can be used to collects the elements of the stream into an List. |

| | | |
|---|---|---|
| 5. | min(Comparator<T> comparator) and max(Comparator<T> comparator)<br><br>List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);<br><br>Optional<Integer> minimumValueType1 = numbers.stream()<br>.filter((Integer val) -> val>=3)<br>.min((Integer val1, Integer val2) -> val1-val2);<br><br>System.out.println(minimumValueType1.get());<br>//output: 4<br><br>Optional<Integer> minimumValueType2 = numbers.stream()<br>.filter((Integer val) -> val>=3)<br>.min((Integer val1, Integer val2) -> val2-val1);<br><br>System.out.println(minimumValueType2.get());<br>//output: 10<br><br>*Can you pls comment the output for **max** for both the types* | Finds the minimum or maximum element from the stream based on the comparator provided. |
| 6. | count()<br><br>List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);<br><br>long noOfValuesPresent = numbers.stream()<br>.filter((Integer val1) -> val1>=3)<br>.count();<br><br>System.out.println(noOfValuesPresent);<br>//output: 3 | returns the count of element present in the stream |

| | | |
|---|---|---|
| 7. | anyMatch(Predicate<T> predicate)<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br>`boolean hasValueGreaterThanThree = numbers.stream()`<br>`.anyMatch((Integer val) -> val>3);`<br>`System.out.println(hasValueGreaterThanThree);`<br>`//output: true` | Checks if any value in the stream match the given predicate and return the boolean. |
| 8. | allMatch(Predicate<T> predicate)<br>*Can you pls try it out...* | Checks if all value in the stream match the given predicate and return the boolean. |
| 9. | noneMatch(Predicate<T> predicate)<br>*Can you pls try it out...* | Checks if no value in the stream match the given predicate and return the boolean. |
| 10. | findFirst()<br>`List<Integer> numbers = Arrays.asList(2, 1, 4, 7, 10);`<br>`Optional<Integer> firstValue = numbers.stream()`<br>`.filter((Integer val) -> val >= 3)`<br>`.findFirst();`<br>`System.out.println(firstValue.get());`<br>`//output: 4` | finds the first element of the stream. |
| 11. | findAny()<br>*Can you pls try it out...* | finds any random element of the stream. |

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.

```
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:

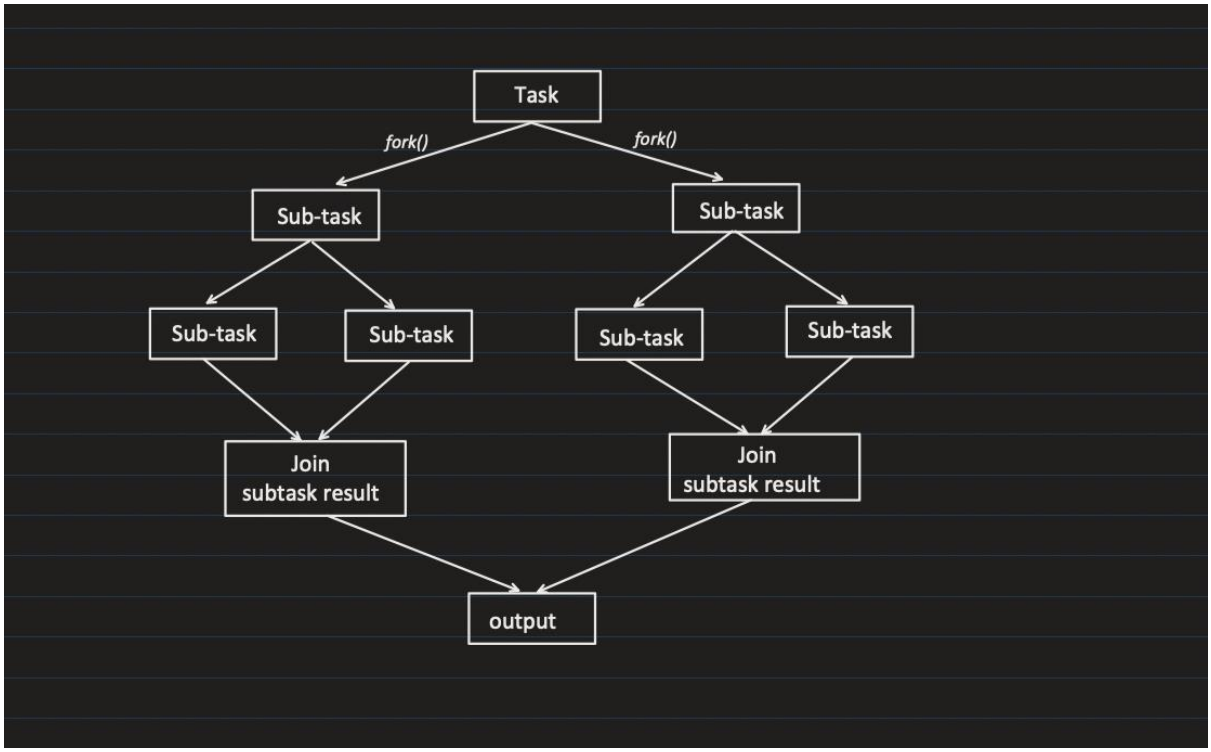
- Task splitting : it uses "spliterator" function to split the data into multiple chunks.
- Task submission and parallel processing : Uses Fork-Join pool technique.

*Note: i will cover Fork-Join pool implementation in Multithreading topic*

```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");
    }
}
```

Output:

```
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
```


Task
fork()
fork()
Sub-task
Sub-task
Sub-task
Sub-task
Sub-task
Sub-task
Join
subtask result
Join
subtask result
output

# (Concept && Coding) Java Collections: Page1

Java Collections Framework

Monday, 7 August 2023 8:16 PM

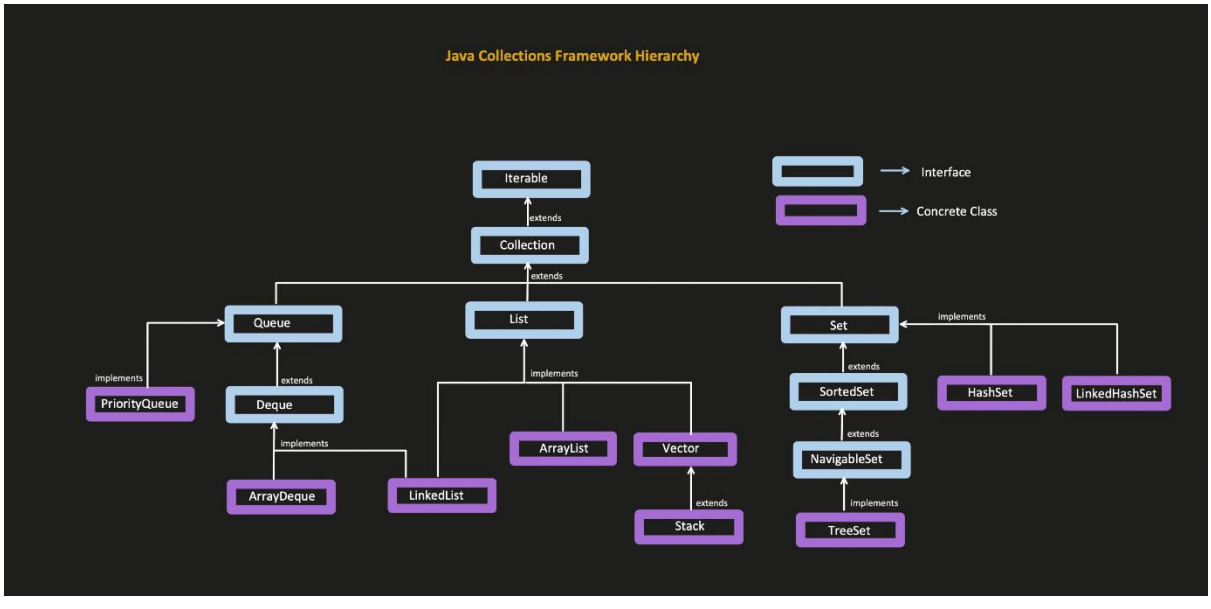
- What is Java Collections Framework?

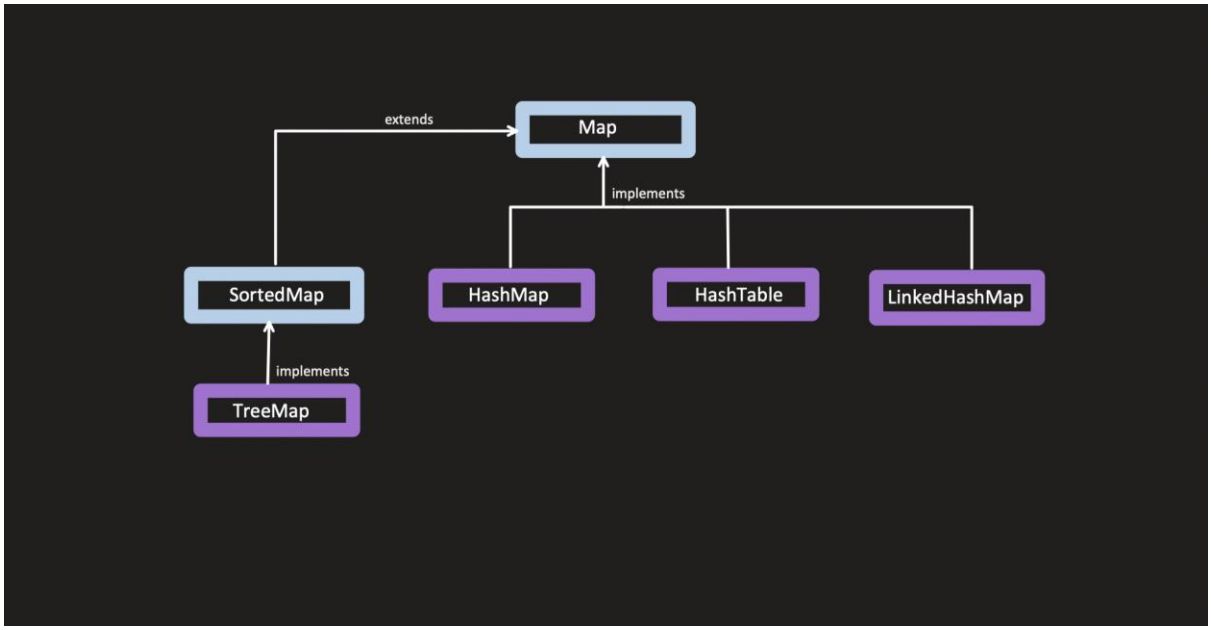
  - Added in Java version 1.2
  - Collections is nothing but a group of Objects.
  - Present in ***java.util*** package.
  - Framework provide us the architecture to manage these "group of objects" i.e. add, update, delete, search etc.

Why we need Java Collections Framework?

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

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

        int arr[] = new 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);
    }
}
```






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

        List<Integer> values = new ArrayList<>();
        values.add(1);
        values.add(2);
        values.add(3);
        values.add(4);


        //using iterator
        System.out.println("Iterating the values using iterator method");
        Iterator<Integer> valuesIterator = values.iterator();
        while (valuesIterator.hasNext()){
            int val = valuesIterator.next();
            System.out.println(val);
            if(val == 3){
                valuesIterator.remove();
            }
        }

        System.out.println("Iterating the values using for-each loop");
        for(int val : values){
            System.out.println(val);
        }

        //using forEach method
        System.out.println("testing forEach method");
        values.forEach((Integer val) -> System.out.println(val));
    }
}
```

**Output:**

```
Iterating the values using iterator method
1
2
3
4
Iterating the values using for-each loop
1
2
4
testing forEach method
1
2
4
```

**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.**

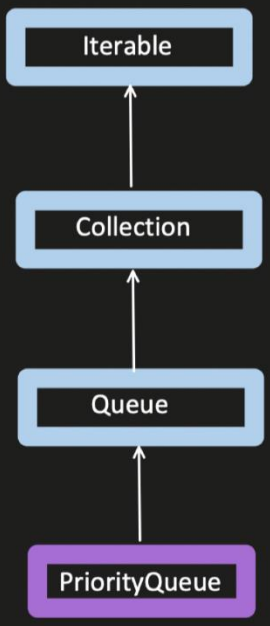
| S.NO | METHODS | Available in | USAGE |
|---|---|---|---|
| 1. | size() | Java1.2 | It returns the total number of elements present in the collection. |
| 2. | isEmpty() | Java1.2 | Used to check if collection is empty or has some value. It return true/false. |
| 3. | contains() | Java1.2 | Used to 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. |

## QUEUE:


- 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. |

## PriorityQueue:



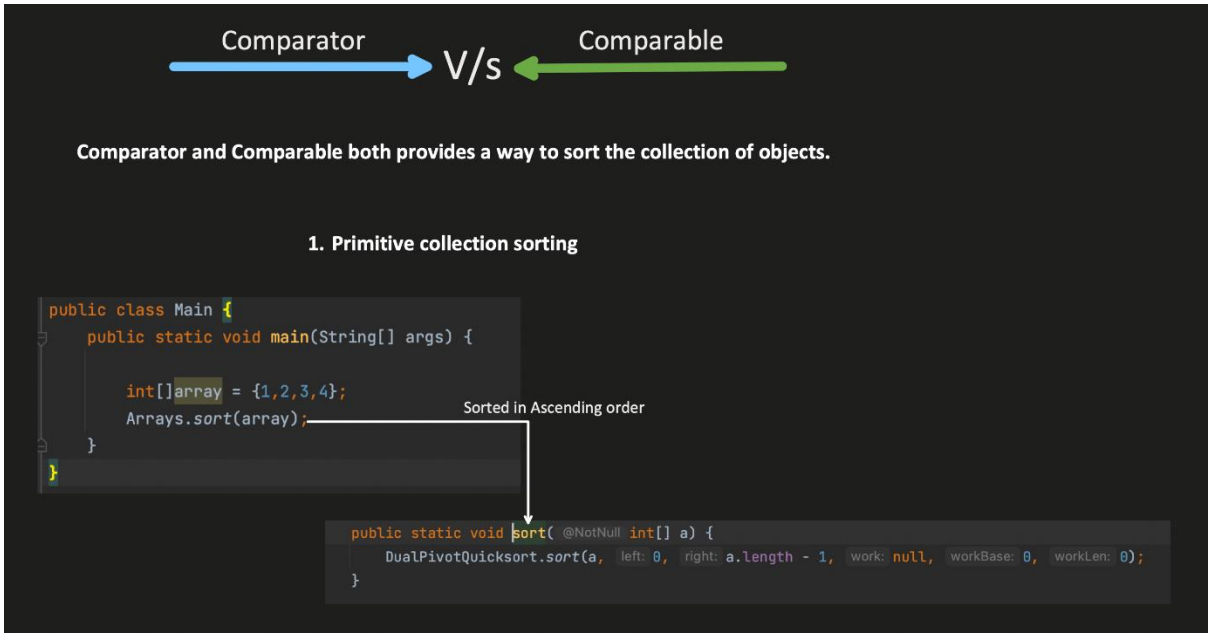

- Its of 2 types, **Minimum Priority Queue** and **Maximum Priority Queue**
- It is based on priority Heap (Min Heap and Max Heap).
- Elements are ordered according to either Natural Ordering (by default) or by ***Comparator*** provided during queue construction time.

**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
```


Comparator
V/s
Comparable
Comparator and Comparable both provides a way to sort the collection of objects.
1. Primitive collection sorting
public class Main {
public static void main(String[] args) {
int[]array = {1,2,3,4};
Arrays.sort(array);
}
}
Sorted in Ascending order
public static void sort( @NotNull int[] a) {
DualPivotQuicksort.sort(a, left: 0, right: a.length - 1, work: null, workBase: 0, workLen: 0);
}


2. Object collection sorting
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);
}
}
public class Car {
String carName;
String carType;
Car(String name, String type){
this.carName = name;
this.carType = type;
}
}
Exception in thread "main" java.lang.ClassCastException Create breakpoint : Car cannot be cast to java.lang.Comparable
at java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:320)
at java.util.ComparableTimSort.sort(ComparableTimSort.java:188)
at java.util.Arrays.sort(Arrays.java:1246)
at Main.main(Main.java:11)

```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);
        }
    }
}
```

Output:

```
1
2
4
6
9
11
```

```java
public class Car {

    String carName;
    String carType;

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

```java
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
```

```java
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) -> obj1.carName.compareTo(obj2.carName));

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

Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

```java
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, (Car ob1, Car ob2) -> ob2.carName.compareTo(ob1.carName));

        cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." + carObj.carType));

    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```java
public class Car {

    String carName;
    String carType;

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

```java
public class CarNameComparator implements Comparator<Car> {

    @Override
    public int compare(Car o1, Car o2) {
        return o2.carName.compareTo(o1.carName);
    }
}
```

```java
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 CarNameComparator());

        cars.forEach((Car carObj) -> System.out.println(carObj.carName + ".." + carObj.carType));

    }
}
```

Output:

```
suv..petrol
sedan..diesel
hatchback..cng
```

```java
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 o1, Car o2) {
        return o1.carName.compareTo(o2.carName);
    }
}
```

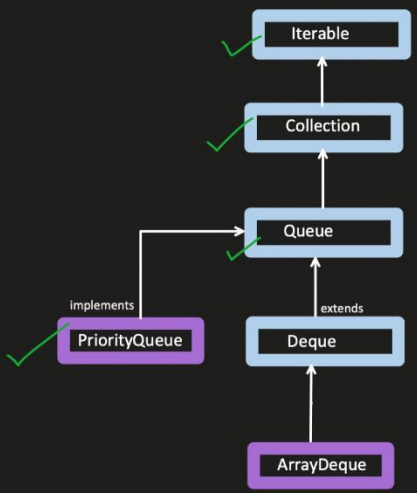
```java
public class Main {
    public static void main(String[] args) {

        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));

    }
}
```

## Output:

```
hatchback..cng
sedan..diesel
suv..petrol
```

**Deque:**

Stands for Double Ended Queue. Means addition and removal can be done from both the sides of the queue.



**ArrayDeque:**

ArrayDeque (Concrete class), implements the methods which are available in Queue and Deque Interface.

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


        ArrayDeque<Integer> arrayDequeAsQueue = new ArrayDeque<>();
        //Insertion
        arrayDequeAsQueue.addLast( e: 1);
        arrayDequeAsQueue.addLast( e: 5);
        arrayDequeAsQueue.addLast( e: 10);

        //Deletion
         int element = arrayDequeAsQueue.removeFirst();
         System.out.println(element);

         //LIFO (last in first out)
        ArrayDeque<Integer> arrayDequeAsStack = new ArrayDeque<>();
        arrayDequeAsStack.addFirst( e: 1);
        arrayDequeAsStack.addFirst( e: 5);
        arrayDequeAsStack.addFirst( e: 10);

        //Deletion
        int removedElem = arrayDequeAsStack.removeFirst();
        System.out.println(removedElem);
    }
}
```

add() -> internally calls addLast() method
offer() -> calles offerLast() method
poll() -> calls pollFirst() method
remove() -> calls removeFirst() method
peek() -> calls peekFirst() method
element() -> calls getFirst() method

Time Complexity:

- Insertion: Amortized(Most of the time or Average) complexity is O(1) except few cases like
  O(n) : when queue size threshold reached and try to insert an element at end or front, then its O(n) as values are copied to new queue with bigger size.
- Deletion: O(1)
- Search: O(1)

Space Complexity:
O(n)

## Methods available in List Interface:

**Methods available in Collection Interface + new Methods defined in List Interface**

**Collection Interface Methods as explained previously:**

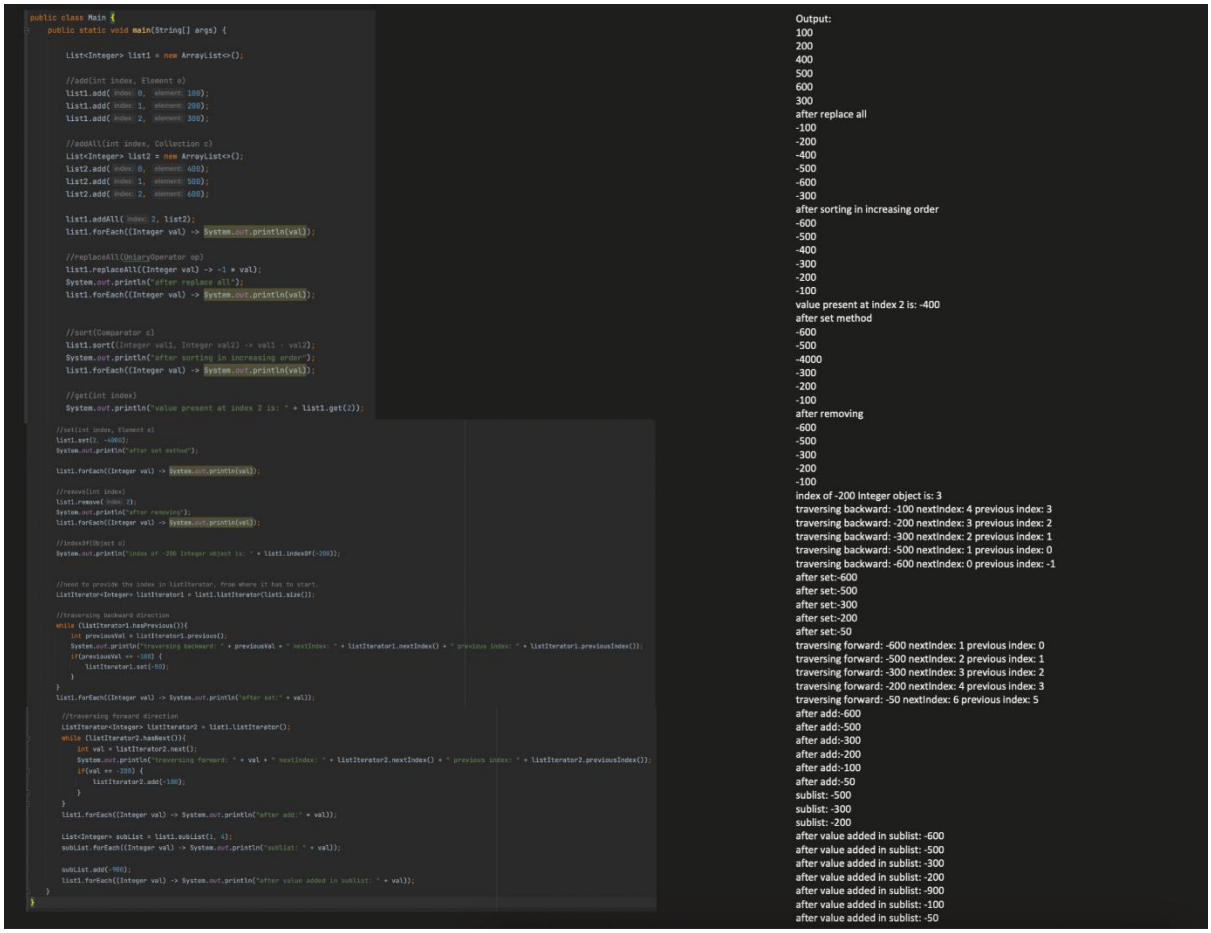
| S.NO | METHODS | USAGE |
|---|---|---|
| 1. | size() | It returns the total number of elements present in the collection. |
| 2. | isEmpty() | Used to check if collection is empty or has some value. It return true/false. |
| 3. | contains() | Used to search an element in the collection, returns true/false.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

**New Methods defined in the List Interface:**

| S.No. | Methods Available in Queue Interface | Usage |
|---|---|---|
| 1. | add(int index, E element) | - Insert the element at the specific position in the list. if there is any element present at that position, it shift it to its Right |
| 2. | addAll(int index, Collection c) | Insert all the elements of the specified collection into this list at specific index, and shift the element at this index and subsequent element to the right. |
| 3. | replaceAll(UniaryOperator op) | relaces each element of the list, with the resulf of appliying the operator the element. |
| 2. | sort(Comparator c) | sort based on the order provided by the comparator. |
| 3. | get(int index) | Return the element at the specified position in the list. |
| 4. | set(int index, Element e) | Replace the element at the specified index in the list with the element provided. |
| 5. | remove(int index) | Remove the element from the index and shift subsequent elements to the left. |
| 6. | indexOf(Object o) | Returns the index of the first occurrence of the specified element in the list. -1 if list does not contains the element. |
| 7. | lastIndexOf(Object o) | Returns the index of the lastoccurrence of the specified element in the list. -1 if list does not contains the element. |

<table>
<tr>
<td>8.</td>
<td>ListIterator<E> listIterator()</td>
<td>
listIterator return the object of ListIterator.<br>
ListIterator (interface) extends from the Iterator (interface)<br><br>
It has all the methods which are avaiable in Iterator and helps to iterate in forward direction like:
<table>
<tr><th>Method Name</th><th>Usage</th></tr>
<tr><td>hasNext()</td><td>Returns true,<br>if there are more elements in collection</td></tr>
<tr><td>next()</td><td>Returns the next element in the iteration</td></tr>
<tr><td>remove()</td><td>Removes the last element returned by iterator</td></tr>
</table>
+<br><br>
New methods which are intorduced in ListIterator, which help to iterate in backward direction:
<table>
<tr><th>Method Name</th><th>Usage</th></tr>
<tr><td>boolean hasPrevious()</td><td>Returns true,<br>if there are more elements in list while traversing in reverse order. Else throw exception</td></tr>
<tr><td>E previous()</td><td>Returns the previous element and move the cursor position backward.</td></tr>
<tr><td>int nextIndex()</td><td>Returns the index of the next element.<br>(return the size of the list, if its at the end of the list)</td></tr>
<tr><td>int previousIndex()</td><td>Returns the index of the previous element.<br>(return -1 of the list, if its at the begining of the list)</td></tr>
<tr><td>set(E e)</td><td>Replaces the last element returned by next or previous with the specified element.</td></tr>
<tr><td>add(E e)</td><td>Insert the specified element immediately before the element that would be returned by the next and after the element that would be returned by the previous.<br><br>Subsequent call to next() would be uaffected.</td></tr>
</table>
</td>
</tr>
</table>

| | | |
|---|---|---|
| 9. | ListIterator<E> listIterator(int index) | start the iterator from the specific index.<br>specified index indicates the first element that would be returned by an initial call to next. |
| 10. | List<E> subList(int fromIndex, int toIndex) | return the portion of the list.<br>fromIndex - Inclusive<br>toIndex - Exclusive<br><br>if fromIndex == toIndex, return sublist is empty.<br>Any changes in sublist, will change in main list also and vise versa. |



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

        List<Integer> list1 = new ArrayList<>();

        //add(int index, Element e)
        list1.add( index: 0,  element: 100);
        list1.add( index: 1,  element: 200);
        list1.add( index: 2,  element: 300);

        //addAll(int index, Collection c)
        List<Integer> list2 = new ArrayList<>();
        list2.add( index: 0,  element: 400);
        list2.add( index: 1,  element: 500);
        list2.add( index: 2,  element: 600);

        list1.addAll( index: 2, list2);
        list1.forEach((Integer val) -> System.out.println(val));

        //replaceAll(UnaryOperator op)
        list1.replaceAll((Integer val) -> -1 * val);
        System.out.println("after replace all");
        list1.forEach((Integer val) -> System.out.println(val));


        //sort(Comparator c)
        list1.sort((Integer val1, Integer val2) -> val1 - val2);
        System.out.println("after sorting in increasing order");
        list1.forEach((Integer val) -> System.out.println(val));

        //get(int index)
        System.out.println("value present at index 2 is: " + list1.get(2));

        //set(int index, Element e)
        list1.set(2, -4000);
        System.out.println("after set method");

        list1.forEach((Integer val) -> System.out.println(val));

        //remove(int index)
        list1.remove( index: 2);
        System.out.println("after removing");
        list1.forEach((Integer val) -> System.out.println(val));

        //indexOf(Object o)
        System.out.println("index of -200 Integer object is: " + list1.indexOf(-200));


        //need to provide the index in listIterator, from where it has to start.
        ListIterator<Integer> listIterator1 = list1.listIterator(list1.size());

        //traversing backward direction
        while (listIterator1.hasPrevious()){
            int previousVal = listIterator1.previous();
            System.out.println("traversing backward: " + previousVal + " nextIndex: " + listIterator1.nextIndex() + " previous index: " + listIterator1.previousIndex());
            if(previousVal == -100) {
                listIterator1.set(-50);
            }
        }
        list1.forEach((Integer val) -> System.out.println("after set:" + val));

        //traversing forward direction
        ListIterator<Integer> listIterator2 = list1.listIterator();
        while (listIterator2.hasNext()){
            int val = listIterator2.next();
            System.out.println("traversing forward: " + val + " nextIndex: " + listIterator2.nextIndex() + " previous index: " + listIterator2.previousIndex());
            if(val == -200) {
                listIterator2.add(-100);
            }
        }
        list1.forEach((Integer val) -> System.out.println("after add:" + val));

        List<Integer> subList = list1.subList(1, 4);
        subList.forEach((Integer val) -> System.out.println("sublist: " + val));

        subList.add(-900);
        list1.forEach((Integer val) -> System.out.println("after value added in sublist: " + val));
    }
}
```

```
Output:
100
200
400
500
600
300
after replace all
-100
-200
-400
-500
-600
-300
after sorting in increasing order
-600
-500
-400
-300
-200
-100
value present at index 2 is: -400
after set method
-600
-500
-4000
-300
-200
-100
after removing
-600
-500
-300
-200
-100
index of -200 Integer object is: 3
traversing backward: -100 nextIndex: 4 previous index: 3
traversing backward: -200 nextIndex: 3 previous index: 2
traversing backward: -300 nextIndex: 2 previous index: 1
traversing backward: -500 nextIndex: 1 previous index: 0
traversing backward: -600 nextIndex: 0 previous index: -1
after set:-600
after set:-500
after set:-300
after set:-200
after set:-50
traversing forward: -600 nextIndex: 1 previous index: 0
traversing forward: -500 nextIndex: 2 previous index: 1
traversing forward: -300 nextIndex: 3 previous index: 2
traversing forward: -200 nextIndex: 4 previous index: 3
traversing forward: -50 nextIndex: 6 previous index: 5
after add:-600
after add:-500
after add:-300
after add:-200
after add:-100
after add:-50
sublist: -500
sublist: -300
sublist: -200
after value added in sublist: -600
after value added in sublist: -500
after value added in sublist: -300
after value added in sublist: -200
after value added in sublist: -900
after value added in sublist: -100
after value added in sublist: -50
```

**LinkedList: Data structure used is Linked List**

- **LinkedList implements both Deque and List interface.**
- Means it support Dequeue methods like : "getFirst", "getLast", "removeFirst", "removeLast" etc...
  +
  It also support index based operations like List : "get(index)" , "add(index, object)" etc.

```java
public class LinkedListExample {

    public static void main(String args[]){

        LinkedList<Integer> list = new LinkedList<>();

        //using deque functionality
        list.addLast( e: 200);
        list.addLast( e: 300);
        list.addLast( e: 400);
        list.addFirst( e: 100);
        System.out.println(list.getFirst());

        //using list functionality
        LinkedList<Integer> list2 = new LinkedList<>();
        list2.add( index: 0,  element: 100);
        list2.add( index: 1,  element: 300);
        list2.add( index: 2,  element: 400);
        list2.add( index: 1,  element: 200);
        System.out.println(list2.get(1) + " and " + list2.get(2));
    }
}
```

Output:

```
100
200 and 300
```

Time Complexity:

- Insertion at start and end : O(1)
- Insertion at particular index : O(n) for lookup of the index + O(1) for adding
- Search: O(n)
- Deletion at start or end: O(1)
- Deletion at specific index: O(n) for the lookup of the index + O(1) for removal

Space Complexity:
O(n)

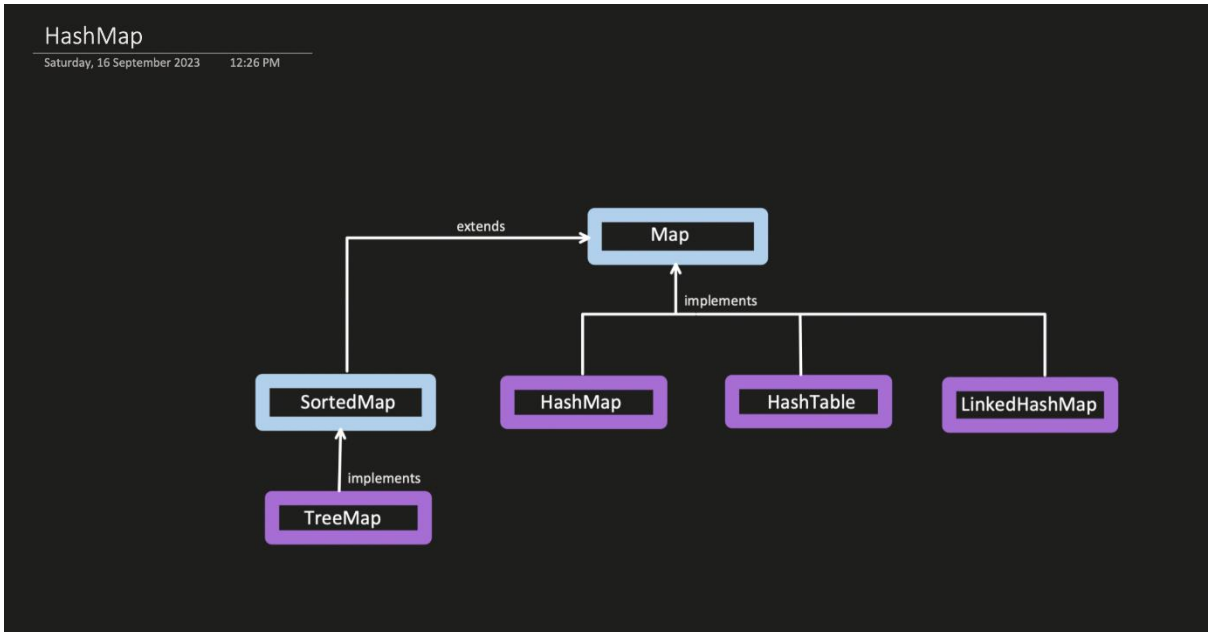
| Collection Till Now | isThreadSafe | Maintains Insertion Order | Null Elements allowed | Duplicate elements allowed | Thread safeVersion |
|---|---|---|---|---|---|
| LinkedList | No | Yes | Yes | Yes | CopyOnWriteArrayList |

```java
public class Test {

    public static void main(String args[]){
        List<Integer> list = new CopyOnWriteArrayList<>();
        list.add( index: 0,  element: 100);
        System.out.println(list.get(0));
    }
}
```

| Collection Till Now | isThreadSafe | Maintains Insertion Order | Null Elements allowed | Duplicate elements allowed | Thread safeVersion |
|---|---|---|---|---|---|
| Stack | Yes | NO | Yes | Yes | N/A |

## Java Collection: Page2

**"Concept && Coding" YT Video Notes**



**Map Properties:**

- its an interface and its implementatins are:
    - HashMap: do not maintains the order.
    - HashTable: Synchronized version of HashMap
    - LinkedHashMap: Maintains the insertion order.
    - TreeMap: sorts the data internally.
- Object that maps key to values.
- Can not contain duplicate key.

Methods available in Map interface:

**HashMap:**

- can store null key or value (HashTable do not contains null key or value)
- Hash Map do not maintains the insertion order
- Its not thread safe (instead use ConcurrentHashMap or HashTable for thread safe HashMap implemenetation)

```java
public class HashMapExample {

    public static void main (String args[]){

        Map<Integer, String> rollNumberVsNameMap = new HashMap<>();
        rollNumberVsNameMap.put(null, "TEST");
        rollNumberVsNameMap.put(0, null);
        rollNumberVsNameMap.put(1, "A");
        rollNumberVsNameMap.put(2, "B");

        //compute if present
        rollNumberVsNameMap.putIfAbsent(null, "test");
        rollNumberVsNameMap.putIfAbsent(0, "ZERO");
        rollNumberVsNameMap.putIfAbsent(3, "C");

        for(Map.Entry<Integer, String> entryMap : rollNumberVsNameMap.entrySet()){
            Integer key = entryMap.getKey();
            String value = entryMap.getValue();
            System.out.println("Key: " + key + " value: " + value);
        }

        //isEmpty
        System.out.println("isEmpty(): " + rollNumberVsNameMap.isEmpty());

        //size
        System.out.println("size: " + rollNumberVsNameMap.size());

        //containsKey
        System.out.println("containsKey(3): " + rollNumberVsNameMap.containsKey(3));

        //get(key)
        System.out.println("get(1): " + rollNumberVsNameMap.get(1));

        //getOrDefault(key)
        System.out.println("get(9): " + rollNumberVsNameMap.getOrDefault( key: 9,  defaultValue: "default value"));

        //remove(key)
        System.out.println("remove(nul): " + rollNumberVsNameMap.remove( key: null));

        for(Map.Entry<Integer, String> entryMap : rollNumberVsNameMap.entrySet()){
            Integer key = entryMap.getKey();
            String value = entryMap.getValue();
            System.out.println("Key: " + key + " value: " + value);
        }

        //keySet()
        for(Integer key: rollNumberVsNameMap.keySet()){
            System.out.println("Key: " + key);
        }

        //values()
        Collection<String> values = rollNumberVsNameMap.values();
        for(String value : values){
            System.out.println("value: " + value);
        }

    }
}
```

Output:

```
Key: null value: TEST
Key: 0 value: ZERO
Key: 1 value: A
Key: 2 value: B
Key: 3 value: C
isEmpty(): false
size: 5
containsKey(3): true
get(1): A
get(9): default value
remove(nul): TEST
Key: 0 value: ZERO
Key: 1 value: A
Key: 2 value: B
Key: 3 value: C
Key: 0
Key: 1
Key: 2
Key: 3
value: ZERO
value: A
value: B
value: C
```

# Time complexity:

add : amortized O(1)
remove: Amortized O(1)
get: Amortized O(1)

## LinkedHashMap:

1. Helps in Maintain Insertion order
   or
   Helps in Maintain Access order

2. Similar to HashMap, but also uses Double LinkedList.

```
map.put(1,"A");
map.put(21,"B");
map.put(23,"C");
map.put(141,"D");
map.put(25,"E");
```

with ACCESS ORDER = TRUE

```
public class LinkedHashMapExample {

    public static void main(String args[]){

        System.out.println("----------below is LInkedHashMap output ----------");

        Map<Integer, String> map = new LinkedHashMap<>( initialCapacity: 16,  loadFactor: .75F,  accessOrder: true);
        map.put(1, "A");
        map.put(21, "B");
        map.put(23, "C");
        map.put(141, "D");
        map.put(25, "E");

        //accessing some data
        map.get(23);
        map.forEach((Integer key, String val) -> System.out.println(key + ":" + val));
    }
}
```

**Output:**

```
----------below is LInkedHashMap output ----------
1:A
21:B
141:D
25:E
23:C
```

-Time complexity is same as of HashMap : Average O(1)

- Its not thread safe and there is no thread safe version available for this.
  so we have to explicitly make it this collection thread safe like this:

```
Map<Integer,String>map2=Collections.synchronizedMap(newLinkedHashMap<>());
```

## TreeMap:

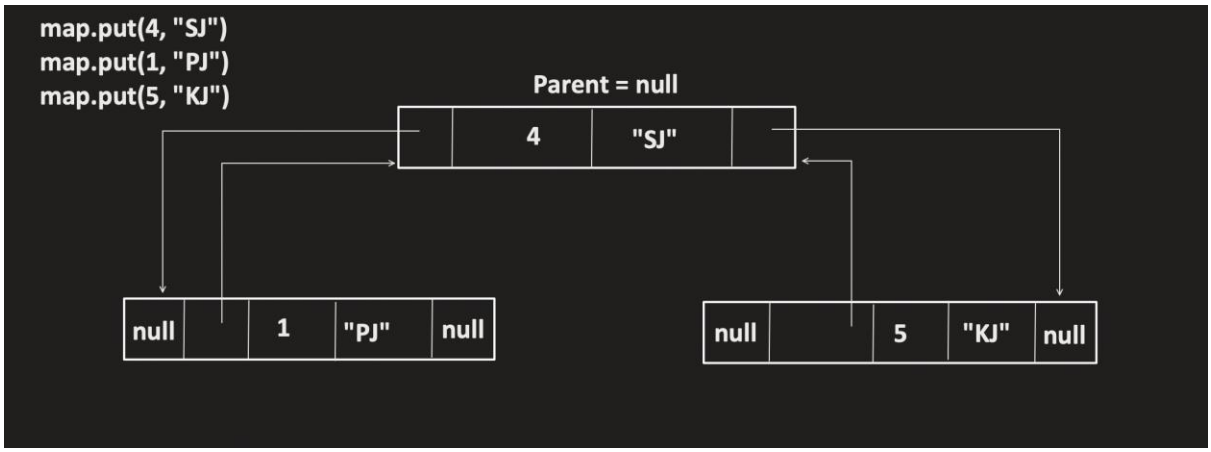
- Map is Sorted according to its ***natural ordering*** of its Key or by ***Comparator*** provided during map creation.
- Its based on Red-Black tree (Self balancing Binary Search Tree)
- O(logn) time complexity of insert, remove, get operations.

```java
public class TreeMapExample {

    public static void main(String args[]){

        Map<Integer, String> map1 = new TreeMap<>((Integer key1, Integer key2) -> key2 - key1);
        map1.put(21, "SJ");
        map1.put(11, "PJ");
        map1.put(13, "KJ");
        map1.put(5, "TJ");
        //decreasing order
        map1.forEach((Integer key, String value) -> System.out.println(key + ":" + value));

        System.out.println("_____");
        Map<Integer, String> map2 = new TreeMap<>();
        map2.put(21, "SJ");
        map2.put(11, "PJ");
        map2.put(13, "KJ");
        map2.put(5, "TJ");
        //increasing order
        map2.forEach((Integer key, String value) -> System.out.println(key + ":" + value));
    }
}
```

**Output:**

```
21:SJ
13:KJ
11:PJ
5:TJ
_____
5:TJ
11:PJ
13:KJ
21:SJ
```


| Left | Parent | Key | Value | Right |
|---|---|---|---|---|



## Methods Available in SORTEDMAP interface:

| S.NO. | METHOD NAME |
|---|---|
| 1. | SortedMap<K,V> headMap(K toKey); |
| 2. | SortedMap<K,V> tailMap(K fromKey); |
| 3. | K firstKey(); |
| 4. | K lastKey(); |

```
public class TreeMapExample {

    public static void main(String args[]){
        SortedMap<Integer, String> map2 = new TreeMap<>();
        map2.put(21, "SJ");
        map2.put(11, "PJ");
        map2.put(13, "KJ");
        map2.put(5, "TJ");

        System.out.println(map2.headMap( toKey: 13));

        System.out.println(map2.tailMap( fromKey: 13));

        System.out.println(map2.firstKey());

        System.out.println(map2.lastKey());
    }
```

**Output:**

```
{5=TJ, 11=PJ}
{13=KJ, 21=SJ}
5
21
```

## Methods Available in NAVIGABLEMAP interface:

```
NavigableMap<Integer,String> map = new LinkedHashMap<>();
map.put(1,"A");
map.put(21,"B");
map.put(23,"C");
map.put(141,"D");
map.put(25,"E");
```

| S.NO. | METHOD NAME | USAGE |
|---|---|---|
| 1. | Map.Entry<K,V> lowerEntry (K key); | map.lowerEntry(23)<br>output: 21,B<br>returns the Entry(Key, value both) node less than 23. If there is no value returns null |
| 2. | K lowerKey(K key); | map.lowerKey(23)<br>output: 21<br>returns the Key only which is less than 23. If there is no value returns null |
| 3. | Map.Entry<K,V> floorEntry(K key); | map.floorEntry(24)<br>output: 23,C<br><br>map.floorEntry(23)<br>output: 23,C<br>returns the Entry(Key, value both) which is less or equal than key. If there is no value returns null |

| | | |
|---|---|---|
| 4. | K floorKey(K key); | map.floorKey(24)<br>output: 23<br><br>map.floorKey(23)<br>output: 23<br>returns the Key only which is less or equal than key. If there is no value returns null |
| 5. | Map.Entry<K,V> ceilingEntry(K key); | map.ceilingEntry(23)<br>output: 23,C<br><br>map.ceilingEntry(24)<br>output: 25,E<br>returns the Entry(Key, value both) which is greater or equal than key. If there is no value returns null |
| 6. | K ceilingKey(K key); | map.ceilingKey(23)<br>output: 23<br><br>map.ceilingKey(24)<br>output: 25<br>returns the Key only which is greater or equal than key. If there is no value returns null |
| 7. | Map.Entry<K,V> higherEntry(K key); | map.higherEntry(23)<br>output: 25,E<br><br>map.higherEntry(25)<br>output: 141, D<br>returns the Entry(Key, value both) which is greater than key. If there is no value returns null |

| | | |
|---|---|---|
| 8. | K higherKey(K key); | map.higherKey(23)<br>output: 25<br><br>map.higherKey(25)<br>output: 141<br>returns the Entry(Key, value both) which is greater than key. If there is no value returns null |
| 9. | Map.Entry<K,V> firstEntry(); | map.firstEntry(23)<br>output: 1,A<br>retruns the least entry in the Map |
| 10. | Map.Entry<K,V> lastEntry(); | map.lastEntry(23)<br>output: 141,D<br>retruns the least entry in the Map |
| 11. | Map.Entry<K,V> pollFirstEntry(); | map.pollFirstEntry()<br>output: 1,A<br>remove and return the least element from the map<br>Now 1 would be removed from the map |
| 12. | Map.Entry<K,V> pollLastEntry(); | map.pollLastEntry()<br>output: 141,D<br>remove and return the least element from the map.<br>Now 141 would be removed from the map |
| 13. | NavigableMap<K,V> descendingMap(); | reverse the map and returns. |
| 14. | NavigableSet<K> navigableKeySet(); | [1, 21, 23, 25, 141]<br>(if treeMap is sorted in ascending, keys will come in ascending, else in descending) |
| 15. | NavigableSet<K> descendingKeySet(); | [141, 25, 23, 21, 1]<br>(if treeMap is sorted in ascending, keys will come in descending else in ascending) |
| 16. | NavigableMap<K,V>headMap(K toKey, boolean inclusive); | map.headMap(23, true)<br>{1=A, 21=B, 23=C} |
| 17. | NavigableMap<K,V> tailMap(K fromKey, boolean inclusive); | map.tailMap(23, true)<br>{23=C, 25=E, 141=D} |

Few properties of SET:

---

1. Collections of Objects, but it does not contains duplicate value (any only one 'null' value you can insert).
2. Unlike List, Set is not an Ordered Collection, means objects inside set does not follow the insertion order.
3. Unline List, Set can not be accessed via index.

Few questions should have come to our mind:

---

1. What Data Structure is used in Stack internally (as it does not allow duplicate values) ?
2. As order is not gurantee, then what if we want to Sort the Set collection?

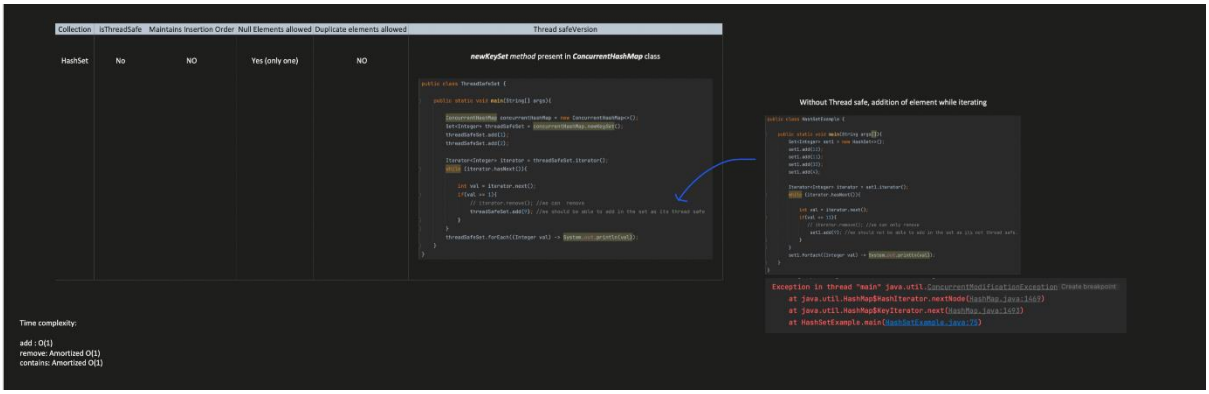
We will try to find the answer of both the above quesitons , as we go further.....

What all methods SET Interface contains:

All methods which are declared in Collections interface, generally that only is avaiable in SET interace. No new method specific to SET is added.

**Collection Interface Methods as explained previously:**

| S.NO | METHODS | USAGE |
|---|---|---|
| 1. | size() | |
| 2. | isEmpty() | |
| 3. | contains() | |
| 4. | toArray() | |
| 5. | add(E element) | returns true, after it insert element in the set only if element is not already present.<br>else if same value is already present, then it returns false. |
| 6. | remove( E element) | |
| 7. | addAll(Collection c) | performs **UNION** of 2 Set Collection.<br><br>set1 = [12, 11, 33, 4]<br>set2 = [11, 9, 88, 10, 5, 12]<br>set1 + set2 = [12, 11, 33, 4, 9, 88, 10, 5] |
| 8. | removeAll(Collection c) | performs **DIFFERENCE** of 2 Set Collection. Delete the values from set which are present in another set.<br><br>set1 = [12, 11, 33, 4]<br>set2 = [11, 9, 88, 10, 5, 12]<br>set1 - set2 = [33, 4] |
| 9. | retainAll(Collection c) | performs **Intersection** of 2 Set Collection. Returns element which are present in both<br>set1 = [12, 11, 33, 4]<br>set2 = [11, 9, 88, 10, 5, 12]<br>set1 intersect set2 = [12, 11] |
| 10. | clear() | |
| 11. | equals() | |
| 12. | stream()<br>and<br>parallelStream() | |
| 13. | iterator() | |


Collection isThreadSafe Maintains Insertion Order Null Elements allowed Duplicate elements allowed Thread safeVersion
HashSet No NO Yes (only one) NO
newKeySet method present in ConcurrentHashMap class
Without Thread safe, addition of element while iterating
Exception in thread "main" java.util.ConcurrentModificationException
at java.util.HashMap$HashIterator.nextNode(HashMap.java:1469)
at java.util.HashMap$KeyIterator.next(HashMap.java:1491)
at HashSetExample.main(HashSetExample.java:29)
Time complexity:
add : O(1)
remove: Amortized O(1)
contains: Amortized O(1)

**LinkedHashSet:**

- Internally it uses: LinkedHashMap
- Maintains the insertion Order of the element
- Its not thread safe:

```
Set<Integer>set=Collections.synchronizedMap(new LinkedHashSet<>());
```

```
public class LinkedHashSetExample {

    public static void main(String args[]){

        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

       Iterator<Integer> iterable =  intSet.iterator();
       while(iterable.hasNext()){
           int val = iterable.next();
           System.out.println(val);
       }
    }
}
```

**Output:**

```
2
77
82
63
5
```

**TreeSet:**

- Internally it uses: TreeMap
- It can not store null value

```
public class SetExample {

   public static void main(String args[]){

      Set<Integer> treeSet = new TreeSet<>();
       treeSet.add(2);
       treeSet.add(77);
       treeSet.add(82);
       treeSet.add(63);
       treeSet.add(5);
     Iterator<Integer> iterable2 =  treeSet.iterator();

       while(iterable2.hasNext()){
           int val = iterable2.next();
           System.out.println(val);
       }
   }
}
```

Output:

```
2
5
63
77
82
```