Java 8 New Features

- Lambada Expression
- Stream Api
- Default Methods in the interface
- Static Methods
- Functional Interface
- Optional
- Method References
- Date Api
- Nashorn Javascript engine

Advantages of Java 8

- Compact Code(Remove Boiler Plate Code)
- More readable reusable code
- More testable code
- Parallel Operation

Lambada Function

Lambda function anonymous function without name return type and access modifier and having one lambada symbol. ->

```
Public void add(int a , int b){

        System.out.println(a+b);

}

(a,b)->System.out.println(a+b);
```

How to Create Custom Immutable Class?

- There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable.
- We can also create immutable class by creating final class that have final data members.
- No setter methods.

  What is immutable object? Can you write immutable object?
- Immutable classes are Java classes whose objects can not be modified once created. Any modification in Immutable object result in new object. For example is String is immutable in Java. Mostly Immutable are also final in Java, in order to prevent sub class from

overriding methods in Java which can compromise Immutability. You can achieve same functionality by making member as non final but private and not modifying them except in constructor.

-

ImmutableDemo.java

```
public final class Employee
{
final String pancardNumber;
public Employee(String pancardNumber)
{
this.pancardNumber=pancardNumber;
}
public String getPancardNumber(){
return pancardNumber;
}
}
```

Priority Que Java

- The **PriorityQueue** class implements the Queue interface in Java
- A PriorityQueue is beneficial when the objects are supposed to be processed based on the priority rather than the First-In-First-Out
- The internal working of the PriorityQueue is based on the Binary Heap.

HashMap Internal

Hashing:-

- Process of converting Object to into form
- Necessary write to method hashcode for better performance and avoid issues.
- hash code of null will always be 0.

Hashcode Method

- hashCode() method is used to get the hash Code of an object.

- hashCode() method of object class returns the memory reference of object in integer form

Equal Method

- Equals method is used to check that 2 objects are equal or not.
- This method is provided by Object class.
- You can override this in your class to provide your own implementation.
- HashMap uses equals() to compare the key whether the are equal or not.
- If equals() method return true, they are equal otherwise not equal.

Buckets

- A bucket is one element of HashMap array.
- It is used to store nodes.
- Two or more nodes can have the same bucket. In that case link list structure is used to connect the nodes.

Index
index = hashCode(key) & (n-1).
Initially Empty hashMap: Here, the hashmap is size is taken as 16.

HashMap map = new HashMap();

Does not overriding hashcode() method has any performance implication ?

This is a good question and open to all , as per my knowledge a poor hashcode function will result in frequent

Callable and Future in Java
Need of Callable

- There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable.
- **Limitation** :- Can't return when threads end
- **Callable:-**
    - the call() method needs to be implemented which returns a result on completion.
    - Note that a thread can't be created with a Callable, it can only be created with a Runnable.

TreeSet

- Java TreeSet class contains unique elements only like HashSet.

- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

Producer & Consumer Problem Using Blocking Queue

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.logging.Level;
import java.util.logging.Logger;
public class ProducerConsumerPattern {
   public static void main(String args[]){
     //Creating shared object
    BlockingQueue sharedQueue = new LinkedBlockingQueue();
     //Creating Producer and Consumer Thread
    Thread prodThread = new Thread(new Producer(sharedQueue));
    Thread consThread = new Thread(new Consumer(sharedQueue));
    //Starting producer and Consumer thread
    prodThread.start();
    consThread.start();
   }
}
//Producer Class in java
class Producer implements Runnable {
   private final BlockingQueue sharedQueue;
   public Producer(BlockingQueue sharedQueue) {
      this.sharedQueue = sharedQueue;
   }



   @Override
```

```
    public void run() {
        for(int i=0; i<10; i++){
            try {
                System.out.println("Produced: " + i);
                sharedQueue.put(i);
            } catch (InterruptedException ex) {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }

}

//Consumer Class in Java
class Consumer implements Runnable{
    private final BlockingQueue sharedQueue;
    public Consumer (BlockingQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        while(true){
            try {
                System.out.println("Consumed: "+ sharedQueue.take());
            } catch (InterruptedException ex) {
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);
```

```
        }
      }
    }


}
```

Output:

```
Produced: 0
Produced: 1
Consumed: 0
Produced: 2
Consumed: 1
Produced: 3
Consumed: 2
Produced: 4
Consumed: 3
Produced: 5
Consumed: 4
Produced: 6
Consumed: 5
Produced: 7
Consumed: 6
Produced: 8
Consumed: 7
```

Serialization

- Well, serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine.
- And deserialization allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

Transient – do not serialize

No way to prevent Subclass to serialize

- There is no direct way to prevent subclass from serialization in java. One possible way by which a programmer can achieve this is by implementing the writeObject() and readObject() methods in the subclass and needs to throw NotSerializableException from these methods.
- These methods are executed during serialization and de-serialization respectively. By overriding these methods, we are just implementing our own custom serialization.

Abstraction

- Simple things represent complex things.

- how to turn on tv (do not know how it work)
- Objects classes variable simple outside complex inside

Encapsulation

- Private fields Keeping fields within private
- Access Though public method :- Provide access through public method
- Protective Barrier :- It's a protective barrier that keeps the data and code safe within the class itself

Inheritance

- This is a special feature of Object Oriented Programming in Java.
- It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

Polymorphism

- This Java OOP concept lets programmers use the same word to mean different things in different contexts.
- One form of polymorphism in Java is method overloading.
- That's when different meanings are implied by the code itself. The other form is method overriding. That's when the different meanings are implied by the values of the supplied variables. See more on this below.

What is the difference between creating String as new() and literal?

When we create string with new() Operator, it's created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.

What is a Lambda Expression ? What's its use ? Core Java

Ans. Its an anonymous method without any declaration.

Java lambda expression is consisted of three components**.**

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Lambda Expression are useful to write shorthand Code and hence saves the effort of writing lengthy Code.

It promotes Developer productivity, Better Readable and Reliable code**.**

```
Drawable d2=()->{
    System.out.println("Drawing "+width);
};
```

What is optional?

- Optional is container
- Helpful to handle null pointer

Methods

- ifPresent - Do execution if present
- isPresent - isPresent
- orElse - Returns the value if present, otherwise returns other.

```
Integer value1 = a.orElse(new Integer(0));

//Optional.get - gets the value, value should be present

Integer value2 = b.get();

return value1 + value2;
```

What is bifunction?

- Java 8 introduced functional style programming,

- -This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.
- BiFunction interface is a built in functional interface which accepts two arguments and produces a result

```
Function<Integer, Integer> printNumber = a -> a*10;

System.out.println("The number is: "+ printNumber.apply(10));


BiFunction<Integer, Integer, Integer> add = (a, b) -> a+b;

System.out.println("The addition of two numbers are: "+ add.apply(3,2));
```

Stream APi what function use?

- Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements.

- Using collections framework in Java, a developer has to use loops and make repeated checks. Another concern is efficiency; as multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.44

- Java 8 introduced the concept of stream that lets the developer to process data declaratively and leverage multicore architecture without the need to write any specific code for it.

Collection interface has two methods to generate a Stream.

- **stream()** - Returns a sequential stream considering collection as its source.
- **parallelStream() -** Returns a parallel Stream considering collection as its source.

forEach

- Stream has provided a new method 'forEach' to iterate each element of the stream.
- The following code segment shows how to print 10 random numbers using forEach.

```
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);
```

filter

The 'filter' method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

int count = strings.stream().filter(string -> string.isEmpty()).count();
```

limit

The 'limit' method is used to reduce the size of the stream. The following code segment shows how to print 10 random numbers using limit.

```
Random random = new Random();
```

random.ints().limit(10).forEach(System.out::println);

sorted

The 'sorted' method is used to sort the stream. The following code segment shows how to print 10 random numbers in a sorted order.

Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);

Parallel Processing

- parallelStream is the alternative of stream for parallel processing**.**
- Take a look at the following code segment that prints a count of empty strings using parallelStream.

List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string

long count = strings.parallelStream().filter(string -> string.isEmpty()).count()

Collectors

Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

List<String> filtered = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.toList());

Statistics

With Java 8, statistics collectors are introduced to calculate all statistics when stream processing is being done.

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);


IntSummaryStatistics stats = numbers.stream().mapToInt((x) -> x).summaryStatistics();


System.out.println("Highest number in List : " + stats.getMax());

System.out.println("Lowest number in List : " + stats.getMin());

System.out.println("Sum of all numbers : " + stats.getSum());

System.out.println("Average of all numbers : " + stats.getAverage());
```

How to remove duplicate stream?

1. Stream.distinct() to remove duplicates

2 The distinct() method returns a stream consisting of the distinct elements of given stream. The element equality is checked according to element's equals() method.

```
ArrayList<Integer> numbersList

   = new ArrayList<>(Arrays.asList(1, 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, 8));


List<Integer> listWithoutDuplicates = numbersList.stream()

        .distinct()

        .collect(Collectors.toList());
```

ArrayList , HashMap , HashSet

Internal Implementation of Hashmap?

Treemap Null key and Null Value allow? what is reason?

Sorted by key so they not allow other wise give null pointer

Stringbuilder vs StringBuffer

String abc = "abc" multiple time concat

StringBuilder multiple time concat

which once preferable.

compatible future https://dzone.com/articles/java-8-parallel-processing-with-completable-future

Completable Future

CompletableFuture.supplyAsync — In case if you want the return value.

CompletableFuture.runAsync — In case if you don't want the return value.

```
List<CompletableFuture<Integer>> futuresList = new
ArrayList<CompletableFuture<Integer>>();

 CompletableFuture<Integer> addAsy = CompletableFuture.supplyAsync(()->(addFun1(10,5)));

CompletableFuture<Integer> subAsy = CompletableFuture.supplyAsync(()->(subFun1(10,5)));

CompletableFuture<Integer> mulAsy = CompletableFuture.supplyAsync(()->(mulFun1(10,5)));
```

Design Pattern - Factory Pattern

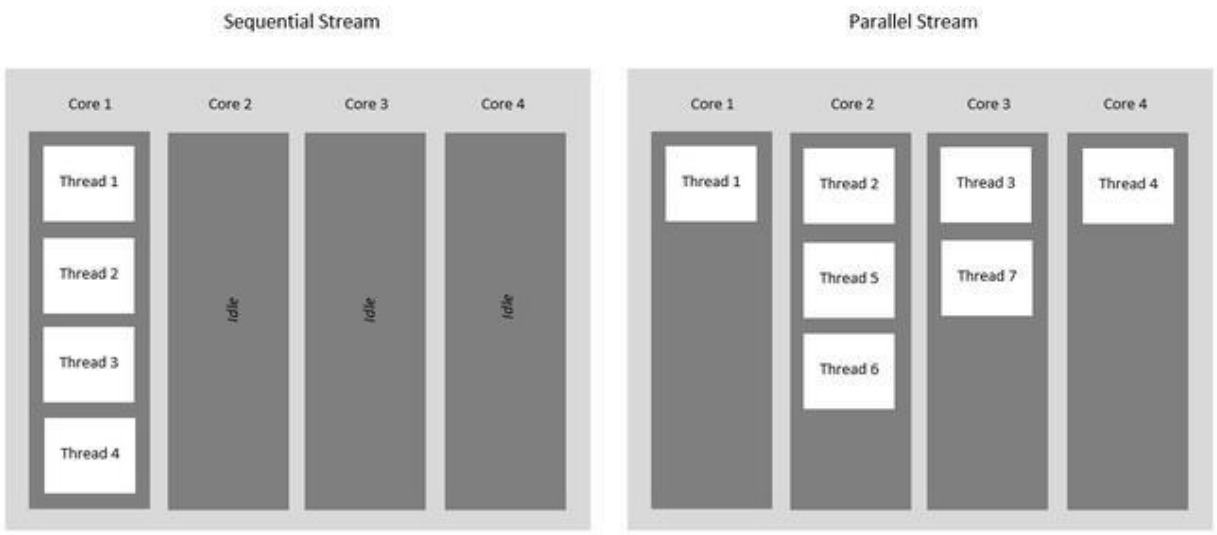
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

```
public interface Shape {
  void draw();
}

public class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}

public class Square implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}

public class Circle implements Shape {
```

```
  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}


public class ShapeFactory {

  //use getShape method to get object of type shape
  public Shape getShape(String shapeType){
    if(shapeType == null){
      return null;
    }
    if(shapeType.equalsIgnoreCase("CIRCLE")){
      return new Circle();

    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
      return new Rectangle();

    } else if(shapeType.equalsIgnoreCase("SQUARE")){
      return new Square();
    }

    return null;
```

```
    }
}
```

Stream vs ParralelStream



CopyOnWrite ArrayList

Here are few points about CopyOnWriteArrayList:

- As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at a certain point both will be synchronized automatically, which is taken care of by JVM. Therefore, there is no effect for threads that are performing read operation.
- It is costly to use because for every update operation a cloned copy will be created. Hence, CopyOnWriteArrayList is the best choice if our frequent operation is read operation.
- The underlined data structure is a grow-able array.
- It is a thread-safe version of ArrayList.
- Insertion is preserved, duplicates, null, and heterogeneous Objects are allowed.
- The main important point about CopyOnWriteArrayList is the Iterator of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying **UnsupportedOperationException.** add() and set() methods on CopyOnWriteArrayList iterator also throws **UnsupportedOperationException.** Also Iterator of CopyOnWriteArrayList will never throw **ConcurrentModificationException**.

Concurrent Hashmap

Key points of ConcurrentHashMap:

- The underlined data structure for ConcurrentHashMap is Hashtable.
- ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.

- At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
- In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
- The default concurrency-level of ConcurrentHashMap is 16.
- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as **Segment locking or bucket locking**. Hence at a time, 16 update operations can be performed by threads.
- Inserting null objects is not possible in ConcurrentHashMap as a key or value.

Java 8 mechanism for hashmap

- a balanced tree instead of a linked list after certain threshold is reached.
- HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable.

Fail Fast Fail Safe

Fail Fast and Fail Safe Systems

The Fail Fast system is a system that shuts down immediately after an error is reported. All the operations will be aborted instantly in it.

The Fail Safe is a system that continues to operate even after an error or fail has occurred. These systems do not abort the operations instantly; instead, they will try to hide the errors and will try to avoid failures as much as possible.