Contents

1. Functional Interface

Contains only 1 abstract method. Can contain any number of default or static methods.

@FunctionalInterface annotation can be used. This is optional. If used, it will restrict to interface to have only single abstract method.
Four main kinds of Functional interface.

- **Consumer** – Accepts only one argument but has no return value.
  Bi Consumer – Accepts two arguments but no return value.
  e.g., Consumer<Integer> consumer = (value) -> System.out.println(value);
- **Predicate** – Accepts one argument and return Boolean value
  Bi-Predicate – Accepts two argument and return Boolean value
- **Function** – Accepts one argument and return some value after processing.
  Bi-Function – Accepts two argument and return some value after processing.
  Unary Operator – Accepts one argument and return value same as argument type.
  Binary Operator – Accepts two argument and return value same as argument type.
- **Supplier** – Won't accept any argument and return some value after processing. e.g., generating Fibonacci series.

2. Java 8 Features

- Stream API
- Lambda Expression
- Functional Interface
- New Date/Time APIs (e.g., LocalDate, LocalTime, LocalDateTime classes)
- Static and Default methods in Interface
- forEach() method added to Iterable interface (Collection parent interface)
- Comparable and Comparator
- Optional class
- CompletableFuture class
- <u>Method Reference</u>
  Method reference is a shorthand notation of a lambda expression to call a method.
  There are four types of method references that are as follows:
  - Static Method Reference
  - Instance Method Reference of a particular object
  - Instance Method Reference of an arbitrary object of a particular type

- Constructor Reference.

```
numList.stream().filter(n -> n >
5).sorted().forEach(System.out::println);
```

3. Heap Memory & Garbage Collection

Heap memory is used to store all the objects. It is created when the JVM starts. Heap area is generally divided into two parts.

- **Young Generation:** New objects are stored in this memory. When memory is filled, garbage collection is performed. This is called **Minor Garbage Collection**.
- **Old Generation:** All the long lived objects which survived many round of garbage collection are stored in this memory. When memory is filled, garbage collection is performed. This is called **Major Garbage Collection**.

Garbage Collection Algorithms

The Java garbage collection process uses a **mark-and-sweep algorithm**. Here's how that works:

- There are two phases in this algorithm: **mark** followed by **sweep**.
- When a Java object is created in the heap, it has a mark bit that is set to 0 (false).
- During the **mark** phase, the garbage collector traverses object trees starting at their roots. When an object is reachable from the root, the mark bit is set to 1 (true). Meanwhile, the mark bits for unreachable objects is unchanged.
- During the **sweep** phase, the garbage collector traverses the heap, reclaiming memory from all items with a mark bit of 0 (false).

4. Metaspace

PermGen	MetaSpace
It is removed from java 8.	It is introduced in Java 8.
PermGen always has a fixed maximum size.	Metaspace by default auto increases its size depending on the underlying OS.
Contiguous Java Heap Memory.	Native Memory(provided by underlying OS).
Inefficient garbage collection.	Efficient garbage collection.

5. Map vs FlatMap in Stream API

The Syntax of the map() is represented as:

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

The Syntax of the flatMap() is represented as:-

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends
R>> mapper)
```

Difference Between map() and flatmap()

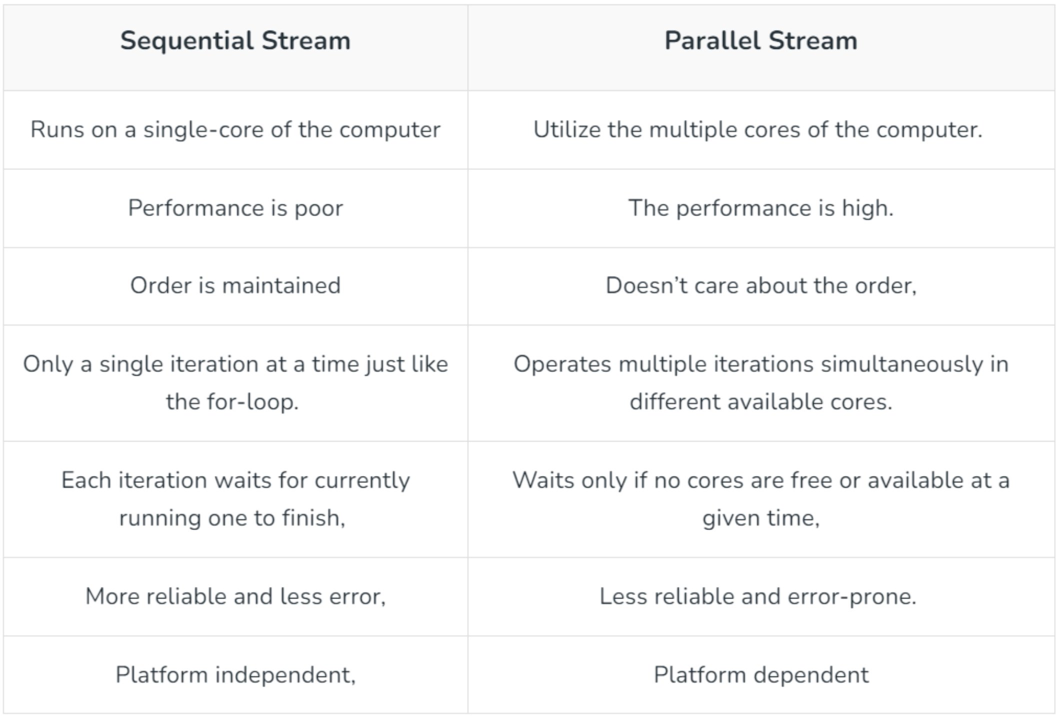
map()	flatMap()
The function passed to map() operation returns a single value for a single input.	The function you pass to flatmap() operation returns an arbitrary number of values as the output.
One-to-one mapping occurs in map().	One-to-many mapping occurs in flatMap().
Only perform the mapping.	Perform mapping as well as flattening.
Produce a stream of value.	Produce a stream of stream value.
map() is used only for transformation.	flatMap() is used both for transformation and mapping.

```java
// making the arraylist object of List of Integer
List<List<Integer> > number = new ArrayList<>();

// adding the elements to number arraylist
number.add(Arrays.asList(1, 2));
number.add(Arrays.asList(3, 4));
number.add(Arrays.asList(5, 6));
number.add(Arrays.asList(7, 8));

System.out.println("List of list-" + number);

// using flatmap() to flatten this list
List<Integer> flatList
    = number.stream()
          .flatMap(list -> list.stream())
          .collect(Collectors.toList());

// printing the list
System.out.println("List generate by flatMap-"
                   + flatList);
```

Output

```
List of list-[[1, 2], [3, 4], [5, 6], [7, 8]]
List generate by flatMap-[1, 2, 3, 4, 5, 6, 7, 8]
```

6. Stream vs Parallel Stream



Sequential Stream	Parallel Stream
Runs on a single-core of the computer	Utilize the multiple cores of the computer.
Performance is poor	The performance is high.
Order is maintained	Doesn't care about the order,
Only a single iteration at a time just like the for-loop.	Operates multiple iterations simultaneously in different available cores.
Each iteration waits for currently running one to finish,	Waits only if no cores are free or available at a given time,
More reliable and less error,	Less reliable and error-prone.
Platform independent,	Platform dependent

7. Executor Framework, Future & CompletableFuture:

ExecutorService is used to create and manage thread pool contains reusable threads and return the result into Future object. It supports async calls.

Four types of Thread Pool objects created by Executor service.

- ➔ SingleThreadPoolExecutor:
  Create single thread pool for execution. Used for sequential execution.
  ExecutorService executor = Executors.*newSingleThreadExecutor*();

- ➔ FixedThreadPool(int nThreads)
  Created thread pool of fixed size. Rest of tasks will be kept in LinkedBlockingQueue.
  ExecutorService executor = Executors.newFixedThreadPool(2);

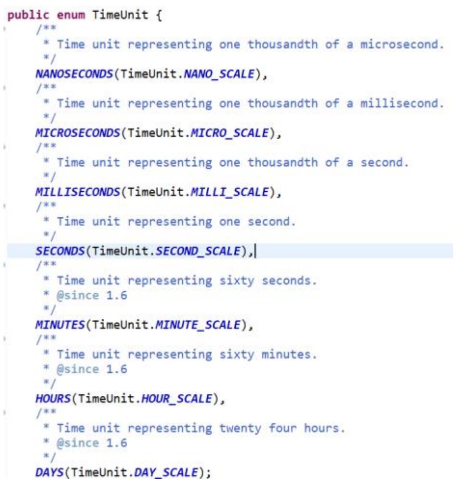
- ➔ CachedThreadPool
  Utilizes the idle threads for the task. Else creates new threads on demand in the pool.
  ExecutorService executor = Executors.*newCachedThreadPool*();

- ➔ ScheduledThreadPool(int corePoolSize)
  This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.
  ScheduledExecutorService executor = Executors.*newScheduledThreadPool*(2);
  scheduleAtFixedRate: Executes task with a fixed interval irrespective of the previous task ended.

ScheduledExecutorService.scheduleAtFixedRate(Runnable command, **long** initialDelay, **long** period, TimeUnit unit);

scheduledAtFixedDelay: Executes task with a fixed delay post the completion of existing task.
ScheduledExecutorService.scheduledAtFixedDelay(Runnable command, **long** initialDelay, **long** period, TimeUnit unit);

Note: TimeUnit is an enum.



```
public enum TimeUnit {
    /**
     * Time unit representing one thousandth of a microsecond.
     */
    NANOSECONDS(TimeUnit.NANO_SCALE),
    /**
     * Time unit representing one thousandth of a millisecond.
     */
    MICROSECONDS(TimeUnit.MICRO_SCALE),
    /**
     * Time unit representing one thousandth of a second.
     */
    MILLISECONDS(TimeUnit.MILLI_SCALE),
    /**
     * Time unit representing one second.
     */
    SECONDS(TimeUnit.SECOND_SCALE),
    /**
     * Time unit representing sixty seconds.
     * @since 1.6
     */
    MINUTES(TimeUnit.MINUTE_SCALE),
    /**
     * Time unit representing sixty minutes.
     * @since 1.6
     */
    HOURS(TimeUnit.HOUR_SCALE),
    /**
     * Time unit representing twenty four hours.
     * @since 1.6
     */
    DAYS(TimeUnit.DAY_SCALE);
```

Important methods of Executor service.

➔ Executor.execute(Runnable command) → *Used to run a single task which implements Runnable Interface.*

➔ ExecutorService.submit() → *Used to execute a task using Callable or Runnable interfaces.*
<T> Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);
<T> Future<T> submit(Callable<T> task);

➔ ExecutorService.invokeAll() → *Invokes all the tasks submitted and return the list of Future object.*

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;
```

➔ ExecutorService.invokeAny() → *submits a task collection and returns the result of one of the tasks that you submitted.*

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;
```

Shutdown operations in ExecutorService

1. ExecutorService.shutdown() → *The basic shutdown method prompts the tasks to terminate by only allowing for the execution of tasks you have previously submitted.* It throws java.util.concurrent.RejectedExecutionException

2. ExecutorService.shutdownNow() → *method attempts to forcibly stop all actively executing tasks and waiting tasks, which leads to a quick shutdown.*
3. ExecutorService.awaitTermination(**long** timeout, TimeUnit unit) → *This method prevents shutdown until all tasks have completed execution or the timeout period ends. This is helpful when you do not want to initiate a shutdownNow but want all tasks to execute instead of just the active ones.*

Executive Completion Service:

Example: *The Executive Completion Service class complements an Executor by executing groups of tasks. It allows you to submit tasks before their execution and track their status as they execute. This is ideal for managing many tasks at once, particularly those you want to execute in a certain order or those you expect to take an extended period of time.*

Completable Future

Exception handling in Completable Future

```
public <U> CompletableFuture<U> handle(
    BiFunction<? super T, Throwable, ? extends U> fn) {
  ...
}
```

```
public CompletableFuture<T> whenComplete(
    BiConsumer<? super T, ? super Throwable> action) {
  ...
}
```

```
public CompletableFuture<T> exceptionally(
    Function<Throwable, ? extends T> fn) {
  ...
}
```

Item	handle()	whenComplete()	exceptionally()
Access to success?	Yes	Yes	No
Access to failure?	Yes	Yes	Yes
Can recover from failure?	Yes	No	Yes
Can transform result from `T` to `U` ?	Yes	No	No
Trigger when success?	Yes	Yes	No
Trigger when failure?	Yes	Yes	Yes
Has an async version?	Yes	Yes	Yes (Java 12)

Method `handle()` and `whenComplete` have access to completable future's success result (`T`) and failure (`Throwable`) as input arguments. On the other hand, method `exceptionally()` only has access to failure as an input argument. Method `handle()` and `exceptionally()` can recover from failure by return a value `T` . However, `whenComplete()` only consumes the arguments without changing the result of the completable future. More precisely, `handle()` can either return the value of type `T` or another value of type `U` as a transformation, but `exceptionally()` can only return the same type `T` .

In case of success, the logic inside `exceptionally()` will be skipped, only the logic of `handle()` and `whenComplete()` will be executed. However, in case of failure, the logic of these three methods will be triggered. All the APIs mentioned above have an asynchronous version with suffix *"Async"* in the method name: `handleAsync` , `whenCompleteAsync` , and `exceptionallyAsync` . But `exceptionallyAsyc` is only available since Java 12.

Runnable vs Callable Interface

Runnable interface	Callable interface
It is a part of *java.lang* package since Java 1.0	It is a part of the *java.util.concurrent* package since Java 1.5.
It cannot return the result of computation.	It can return the result of the parallel processing of a task.
It cannot throw a checked Exception.	It can throw a checked Exception.
In a runnable interface, one needs to override the run() method in Java.	In order to use Callable, you need to override the call()

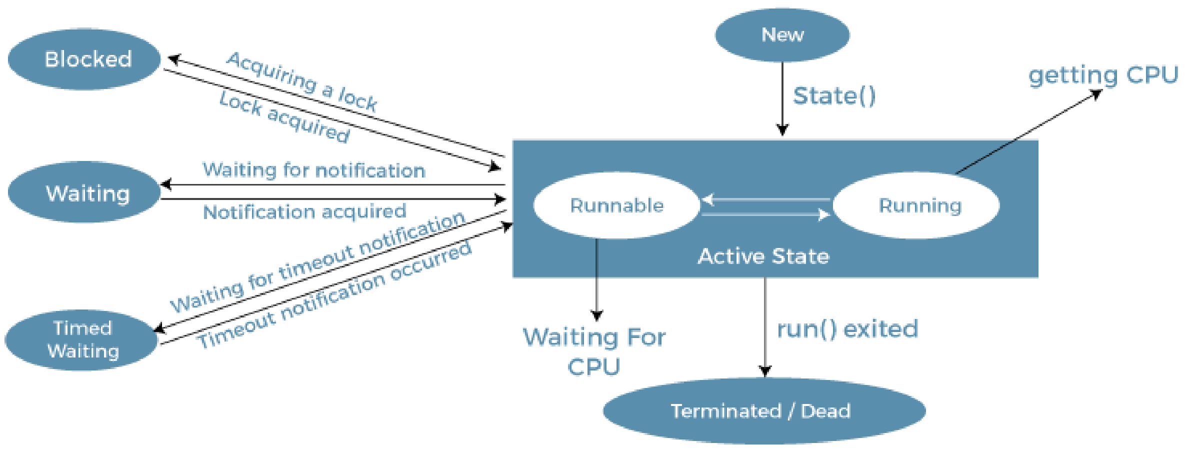
```java
// Runnable Example
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class RunnableExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Runnable task = () -> System.out.println("Hello from Runnable!");
        executor.execute(task);

        executor.shutdown();
    }
}
```

```java
// Callable Example
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<String> task = () -> "Hello from Callable!";
        Future<String> future = executor.submit(task);

        System.out.println(future.get()); // Get the result

        executor.shutdown();
    }
}
```

Daemon Thread:

These are low priority threads used to support user threads. GC (Garbage Collector) and Finalizer are the examples of Daemon thread. JVM terminates when all the user threads completed or terminated and it won't wait for daemon threads to terminate. Instead, it terminates them but itself.
Two methods-
public final void setDaemon(boolean on)
public final boolean isDaemon()

Thread Life Cycle



Life Cycle of a Thread

https://www.baeldung.com/spring-qualifier-annotation

15. Explain Class Loading Systems or Class Loaders in Java

13. Overriding scenario (private, static, final)

8. Difference between Map and Flat Map

What is Exception Chaining (How Exception is occurred and its propagation)?

31. Explain SOLID Principle

39. What is Fail Fast and Failsafe Iterator?

Comparable vs Comparator

Hibernate single Connection pooling multitenancy implementation → AbstractMultitenantConnectionProvider.

@Configuration

Entity Relationship Management

@One to One Mapping

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "attachment_id", referencedColumnName = "attachment_id")
private Attachment attachment;
```

One to Many Mapping

```
@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
@JoinColumn(name = "blog_id")
private List<Blog> blogList;
```

Many to One Mapping

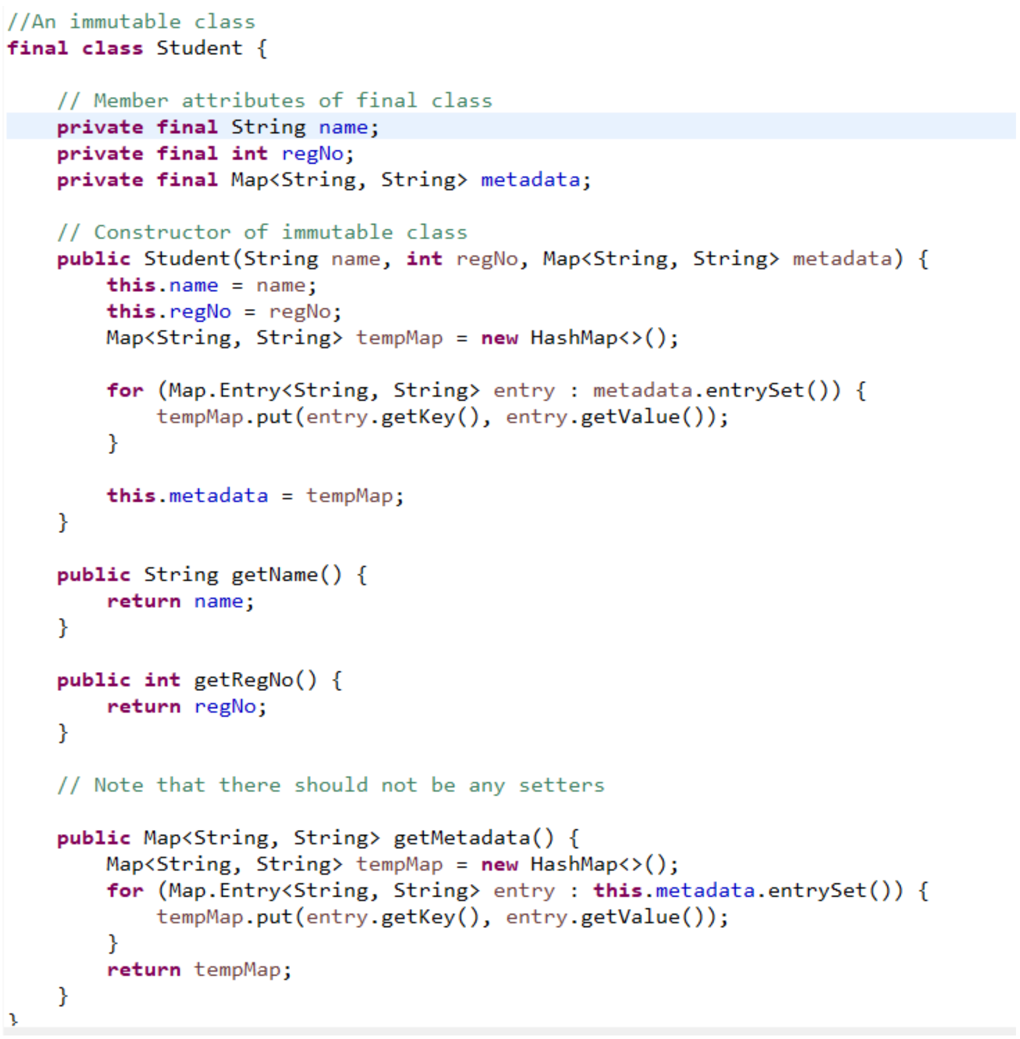
```
@Table(name = "tbl_message")
public class Message {
    @Id
    @Column(name="msg_id")
    @SequenceGenerator(
            name = "message_sequence",
            sequenceName = "message_sequence",
            allocationSize = 1
    )
    @GeneratedValue(
            strategy = GenerationType.SEQUENCE,
            generator = "message_sequence"
    )
    private Long messageId;

    @ManyToOne
    @JoinColumn( // this will add a user_id column to the message table as a for
            name = "sender_id", // specifies the name of the foreign key column
            referencedColumnName = "user_id" // primary key of the user who owns
    )
    private User sender;
```

Many to Many Mapping

```
@ManyToMany(
        cascade = CascadeType.ALL,
        fetch = FetchType.EAGER // fetches the child entities along with parent (n
)
@JoinTable(
        name = "tbl_user_group",
        joinColumns = @JoinColumn(
                name = "group_id",
                referencedColumnName = "group_id"
        ),
        inverseJoinColumns = @JoinColumn(
                name = "user_id",
                referencedColumnName = "user_id"
        )
)
private List<User> members;
```

Immutable class and creation of custom immutable class

- Immutable class objects can never by modified, e.g., String, Wrapper classes, etc.
- Class and its attributes should be final.
- Attributes will be private and final so cannot be accessed and modified.
- Parameterized constructor and Getter methods should return a deep copy of the mutable objects.
- Setters methods are not implemented.

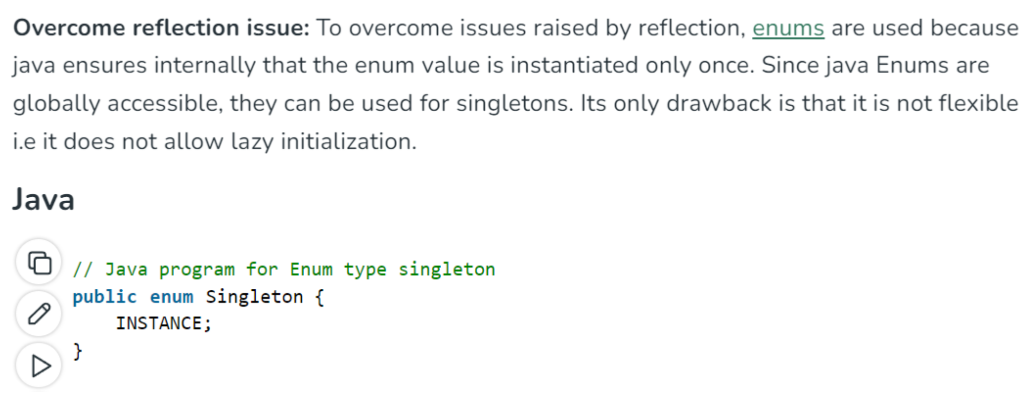


```java
//An immutable class
final class Student {

    // Member attributes of final class
    private final String name;
    private final int regNo;
    private final Map<String, String> metadata;

    // Constructor of immutable class
    public Student(String name, int regNo, Map<String, String> metadata) {
        this.name = name;
        this.regNo = regNo;
        Map<String, String> tempMap = new HashMap<>();

        for (Map.Entry<String, String> entry : metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }

        this.metadata = tempMap;
    }

    public String getName() {
        return name;
    }

    public int getRegNo() {
        return regNo;
    }

    // Note that there should not be any setters

    public Map<String, String> getMetadata() {
        Map<String, String> tempMap = new HashMap<>();
        for (Map.Entry<String, String> entry : this.metadata.entrySet()) {
            tempMap.put(entry.getKey(), entry.getValue());
        }
        return tempMap;
    }
}
```

How to break Singleton class

Using Reflection:

```java
Singleton instance1 = Singleton.instance;
Singleton instance2 = null;
try {
    Constructor[] constructors
        = Singleton.class.getDeclaredConstructors();
    for (Constructor constructor : constructors) {
        // Below code will destroy the singleton
        // pattern
        constructor.setAccessible(true);
        instance2
            = (Singleton)constructor.newInstance();
        break;
    }
}
```



Overcome reflection issue: To overcome issues raised by reflection, enums are used because java ensures internally that the enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

Java

```java
// Java program for Enum type singleton
public enum Singleton {
    INSTANCE;
}
```

Serialization: If we serialize the Singleton class object. While deserializing, it will create new object.

```java
Singleton instance1 = Singleton.instance;
ObjectOutput out = new ObjectOutputStream(
    new FileOutputStream("file.text"));

out.writeObject(instance1);
out.close();

// deserialize from file to object
ObjectInput in = new ObjectInputStream(
    new FileInputStream("file.text"));
Singleton instance2
    = (Singleton)in.readObject();
in.close();

System.out.println("instance1 hashCode:- "
                   + instance1.hashCode());
System.out.println("instance2 hashCode:- "
                   + instance2.hashCode());
```

To overcome this, we have to implement readResolve() method to return the same instance.

```java
class Singleton implements Serializable {

    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }

    // implement readResolve method
    protected Object readResolve() { return instance; }
}
```

Cloning: It will create a new copy of the object.

To overcome this issue, we have to implement clone() method and throw CloneNotSupportedException.

```
// Singleton class
class Singleton extends SuperClass {
    // public instance initialized when loading the class
    public static Singleton instance = new Singleton();

    private Singleton()
    {
        // private constructor
    }

    @Override
    protected Object clone()
        throws CloneNotSupportedException
    {
        return instance;
    }
}
```

Serialization

Serialization in Java is the process of converting an object into a stream of bytes, which can then be stored in a file, sent across a network, or used for other purposes. This allows you to save the state of an object and recreate it later.

To serialize a class, we must implement java.io.Serializable interface.

Serializable class is a marker interface means it won't have methods in it but can only be used to indicate that class can be serialized.

To serialize a class, we use ObjectOutputStream class (writes object state to a stream) and to deserialize it, we are using ObjectInputStream class (Reads object state from stream and recreates the object).

Purpose of SerialVersionUID

SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.

At the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.
Deserialization at the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise, we will get Runtime Exception saying InvalidClassException. This unique Identifier is nothing but **SerialVersionUID**.

Cons of JVM created SerialVersionUID:

- Both sender and receiver should use the same JVM with respect to platform and version also. Otherwise, there can be change in SerialVersionUID which won't allow deserialization.
- If any change observed in .class file, receiver won't be able to deserialize.
- To generate SerialVersionUID internally JVM may use complex algorithms which may create performance problems.

To solve the above problems, we can provide custom value to the variable serialVersionUUID. In this case, sender and receiver won't require same JVM versions as well.

e.g., private static final long SerialVersionUID=10L;

How to block serialization in child class if parent class implements Serialization?

Design Patterns in Java

Creational Design Pattern

Singleton Design Pattern

Prototype Design Pattern

Builder Design Pattern

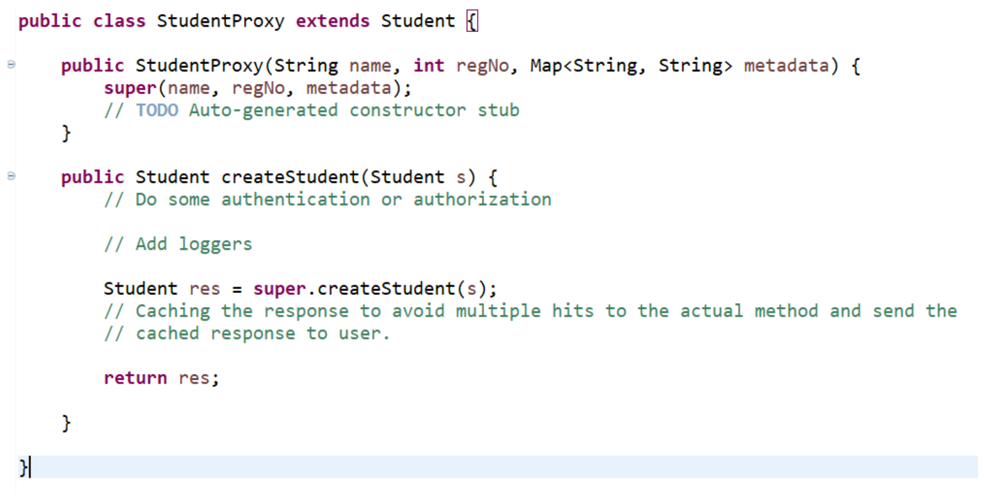
Factory Design Pattern

Abstract Factory Design Pattern

Structural Design Pattern

Proxy Design Pattern

Proxy class is created as the child of original class which focuses on doing some task that are required before the execution of actual method call.
e.g, Do some kind of authorization, or logging, or caching responses so to avoid multiple similar calls to the actual method.



```java
public class StudentProxy extends Student {

    public StudentProxy(String name, int regNo, Map<String, String> metadata) {
        super(name, regNo, metadata);
        // TODO Auto-generated constructor stub
    }

    public Student createStudent(Student s) {
        // Do some authentication or authorization

        // Add loggers

        Student res = super.createStudent(s);
        // Caching the response to avoid multiple hits to the actual method and send the
        // cached response to user.

        return res;

    }

}
```

Behavioral design pattern

Generics In Java

Generics are parameterized types. It adds the type safety feature. In Java Generics, wildcards are represented by the question mark (?) symbol. They allow you to work with generic types in a more flexible way, especially when dealing with unknown types or hierarchies of types.

There are three types of wildcards in Java:

- **Unbounded Wildcard (?)**

  Used when you don't know the specific type of the generic, but you want to work with it in a generic way.

```java
List<?> list = new ArrayList<>();
```

- **Upper Bounded Wildcard (? extends T)**

  Used when you want to work with a collection of objects that are of type T or its subclasses.

```java
List<? extends Number> numbers = new ArrayList<>();
numbers.add(Integer.valueOf(10));  // Valid
numbers.add(Double.valueOf(3.14)); // Valid
// numbers.add("String"); // Invalid
```

- **Lower Bounded Wildcard (? super T)**

  Used when you want to add objects of type T or its supertypes to a collection.

```java
List<? super Integer> integers = new ArrayList<>();
integers.add(Integer.valueOf(10)); // Valid
// integers.add(Double.valueOf(3.14)); // Invalid
```

How to create your own spring boot starter?