Contents

[1. Functional Interface 2](#_Toc178241296)

[2. Java 8 Features 2](#_Toc178241297)

[3. Heap Memory & Garbage Collection 3](#_Toc178241298)

[4. Metaspace 3](#_Toc178241299)

[5. Map vs FlatMap in Stream API 4](#_Toc178241300)

[6. Stream vs Parallel Stream 6](#_Toc178241301)

[7. Executor Framework, Future & CompletableFuture: 6](#_Toc178241302)

[Important methods of Executor service. 7](#_Toc178241303)

[Shutdown operations in ExecutorService 7](#_Toc178241304)

[Completable Future 8](#_Toc178241305)

[Runnable vs Callable Interface 9](#_Toc178241306)

[Daemon Thread: 10](#_Toc178241307)

[Thread Life Cycle 11](#_Toc178241308)

[Entity Relationship Management 12](#_Toc178241309)

[@One to One Mapping 12](#_Toc178241310)

[One to Many Mapping 12](#_Toc178241311)

[Many to One Mapping 12](#_Toc178241312)

[Many to Many Mapping 13](#_Toc178241313)

[Immutable class and creation of custom immutable class 13](#_Toc178241314)

[How to break Singleton class 14](#_Toc178241315)

[Serialization 16](#_Toc178241316)

[Purpose of SerialVersionUID 16](#_Toc178241317)

[How to block serialization in child class if parent class implements Serialization? 17](#_Toc178241318)

[Design Patterns in Java 17](#_Toc178241319)

[Creational Design Pattern 17](#_Toc178241320)

[Singleton Design Pattern 17](#_Toc178241321)

[Prototype Design Pattern 17](#_Toc178241322)

[Builder Design Pattern 17](#_Toc178241323)

[Factory Design Pattern 17](#_Toc178241324)

[Abstract Factory Design Pattern 17](#_Toc178241325)

[Structural Design Pattern 17](#_Toc178241326)

[Proxy Design Pattern 17](#_Toc178241327)

[Behavioral design pattern 18](#_Toc178241328)

[Generics In Java 18](#_Toc178241329)

# 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.

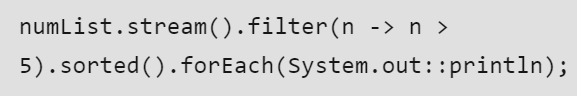
**Binary Operator** – Accepts two argument and return value same as argument type.

* **Supplier** – Won’t accept any argument and return some value after processing. e.g., generating Fibonacci series.

# 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
* Method Reference

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.  
  

# 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.

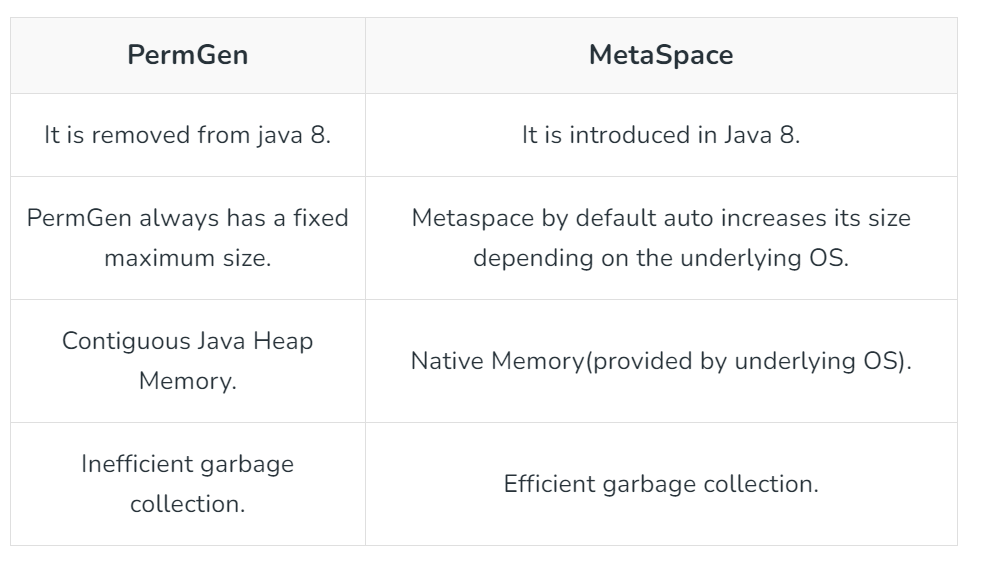
* **Young Generation:** New objects are stored in this memory. When memory is filled, garbage collection is performed. This is called **Minor Garbage Collection**.
* **Old Generation:** All the long lived objects which survived many round of garbage collection are stored in this memory. When memory is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

**Garbage Collection Algorithms**

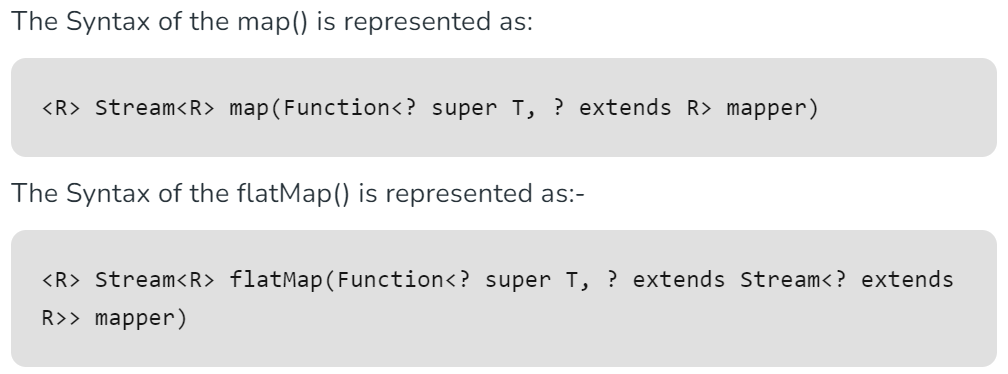
The Java garbage collection process uses a **mark-and-sweep algorithm**. Here’s how that works:

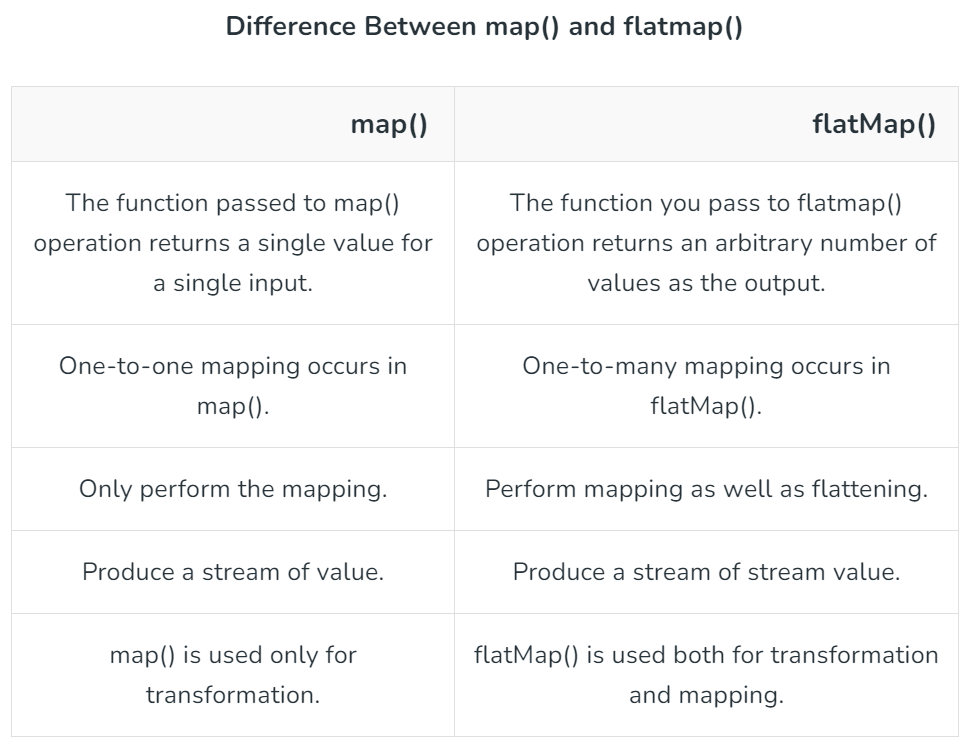
* There are two phases in this algorithm: **mark** followed by **sweep**.
* When a Java object is created in the heap, it has a mark bit that is set to 0 (false).
* During the **mark** phase, the garbage collector traverses object trees starting at their roots. When an object is reachable from the root, 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).

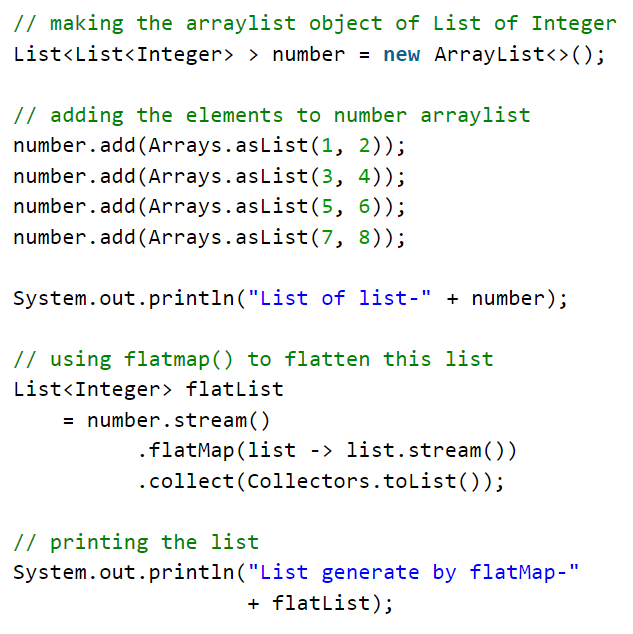
# Metaspace

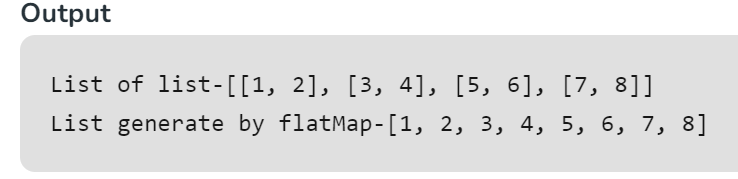


# Map vs FlatMap in Stream API

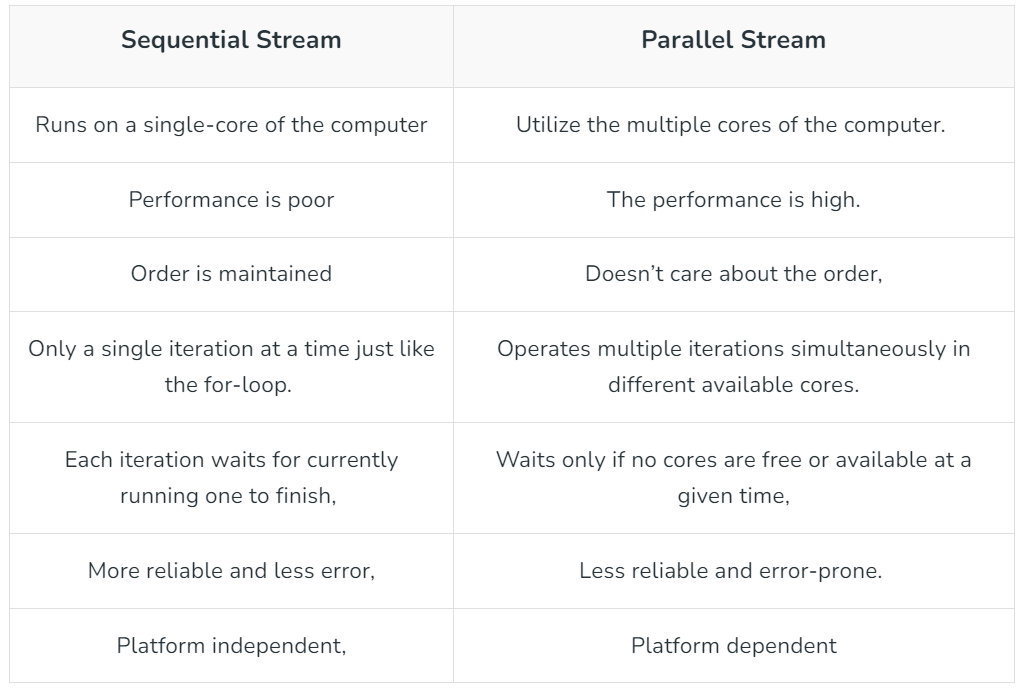








# Stream vs Parallel Stream



# 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);

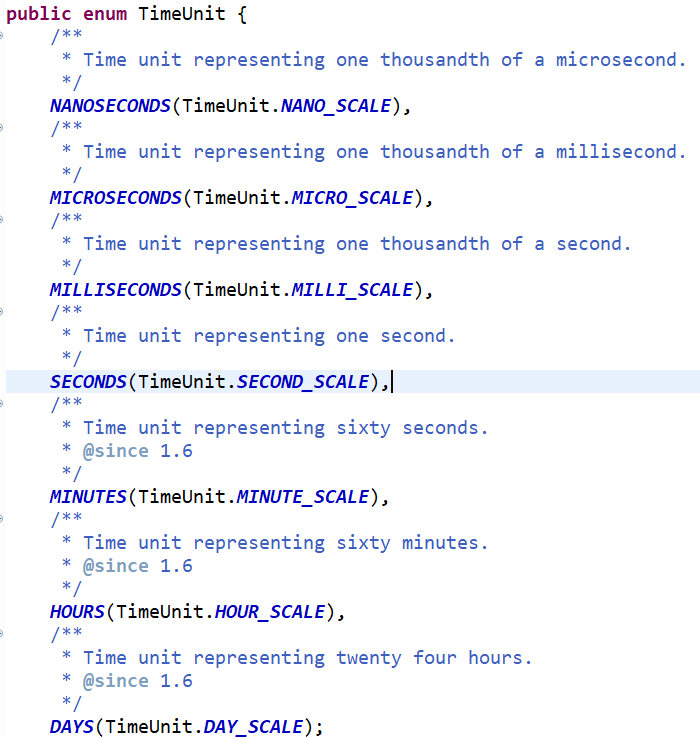
scheduleAtFixedRate: Executes task with a fixed interval irrespective of the previous task ended.

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.



## 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.*

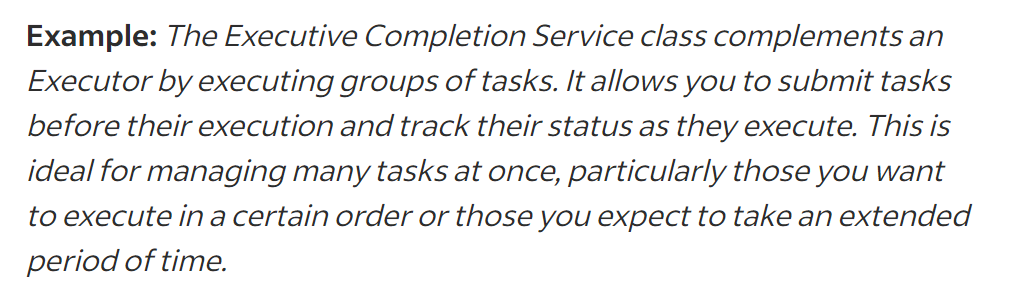
<T> T invokeAny(Collection<? **extends** Callable<T>> tasks)

**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:**

**

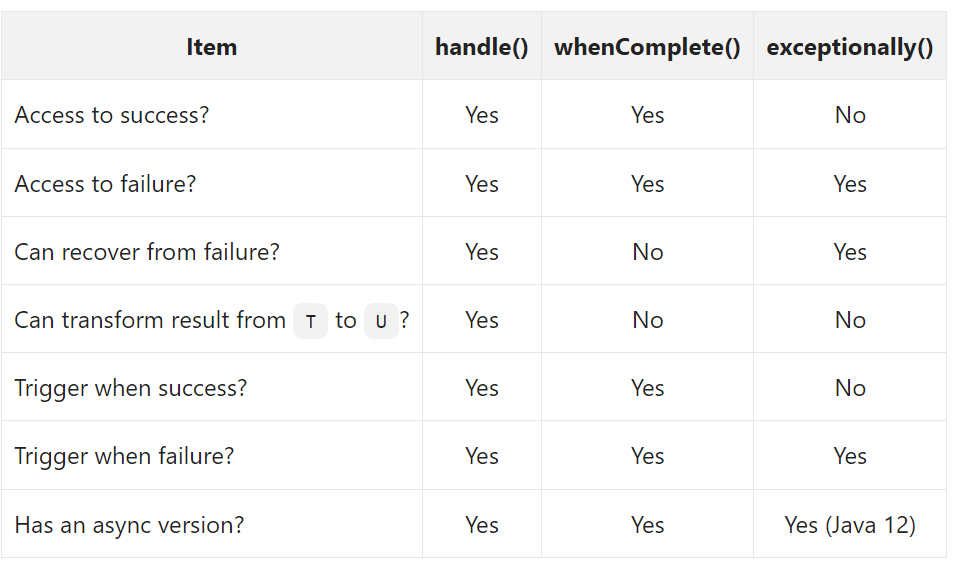
## Completable Future

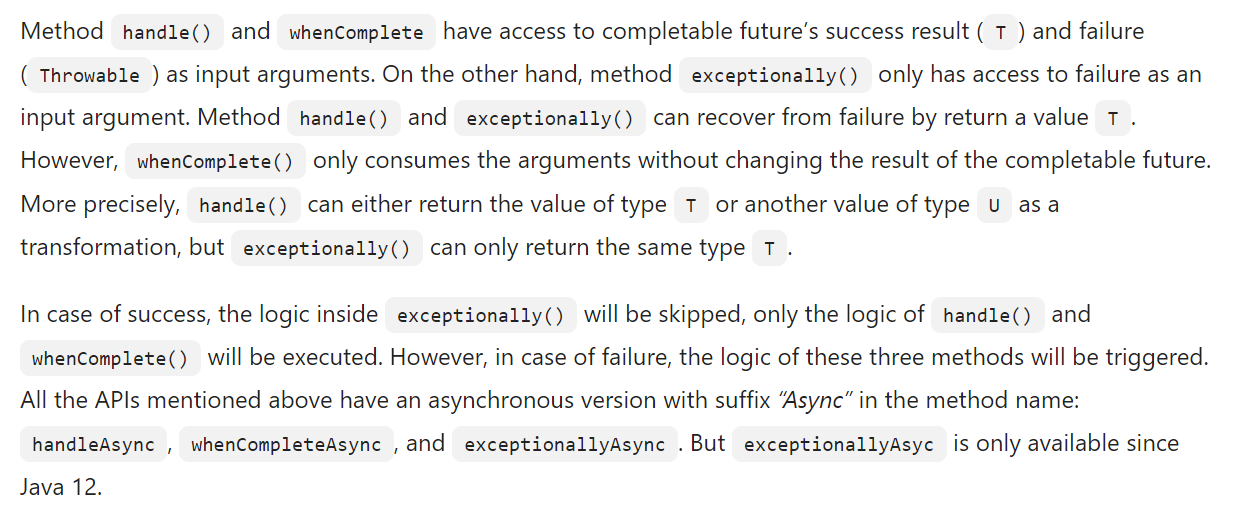
Exception handling in Completable Future



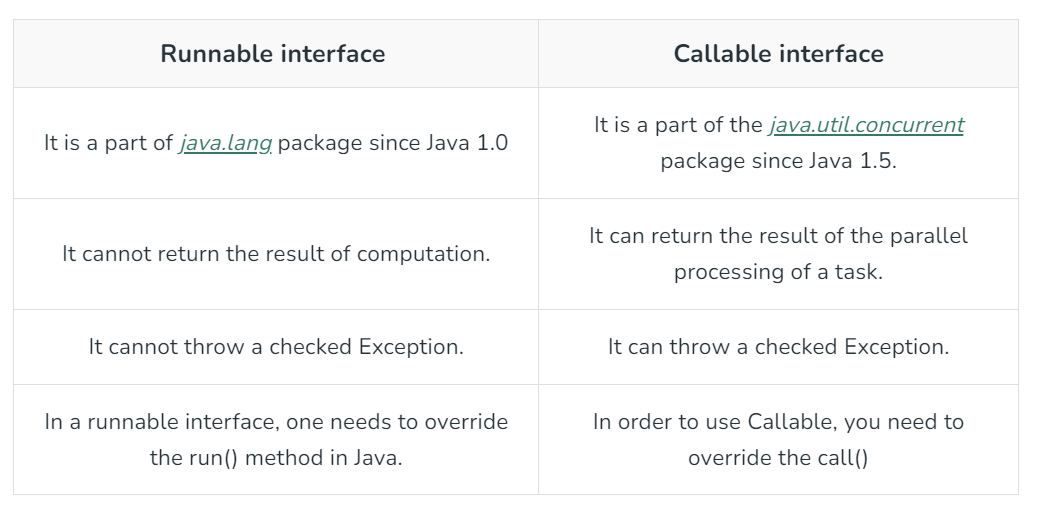


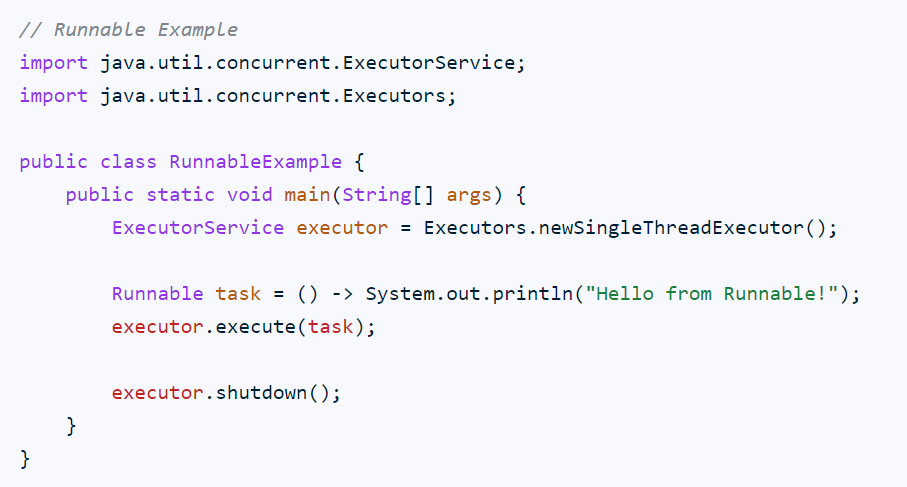


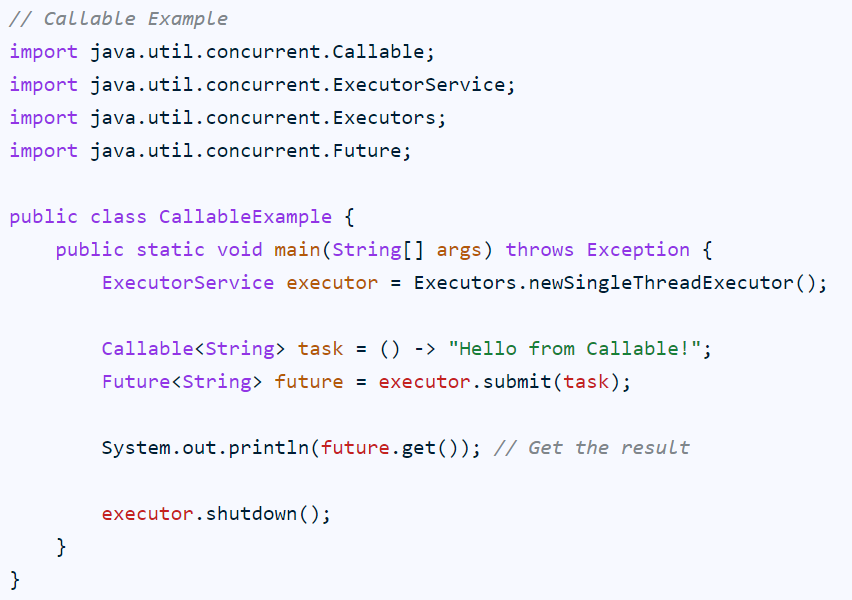




# Runnable vs Callable Interface







# 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



<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?

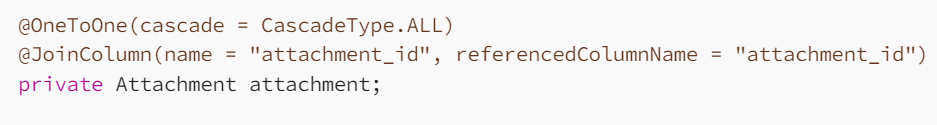
Comparable vs Comparator

Hibernate single Connection pooling multitenancy implementation 🡪 AbstractMultitenantConnectionProvider.

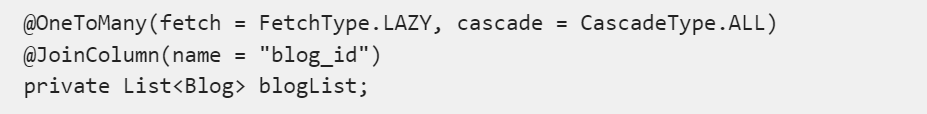
@Configuration

# Entity Relationship Management

## @One to One Mapping



## One to Many Mapping



## Many to One Mapping

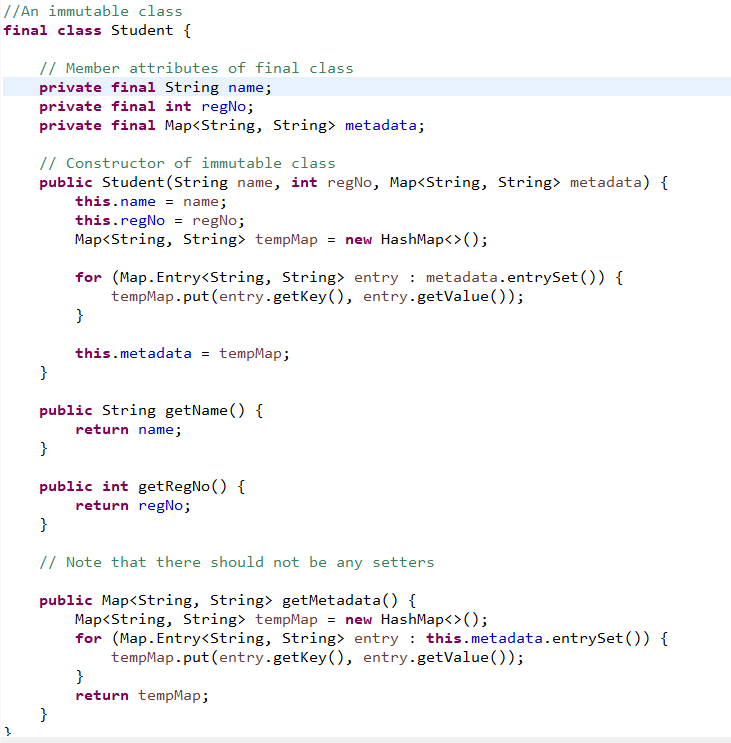


## Many to Many Mapping



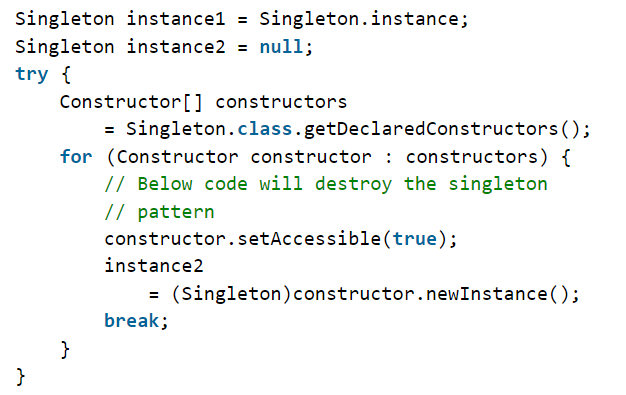
# Immutable class and creation of custom immutable class

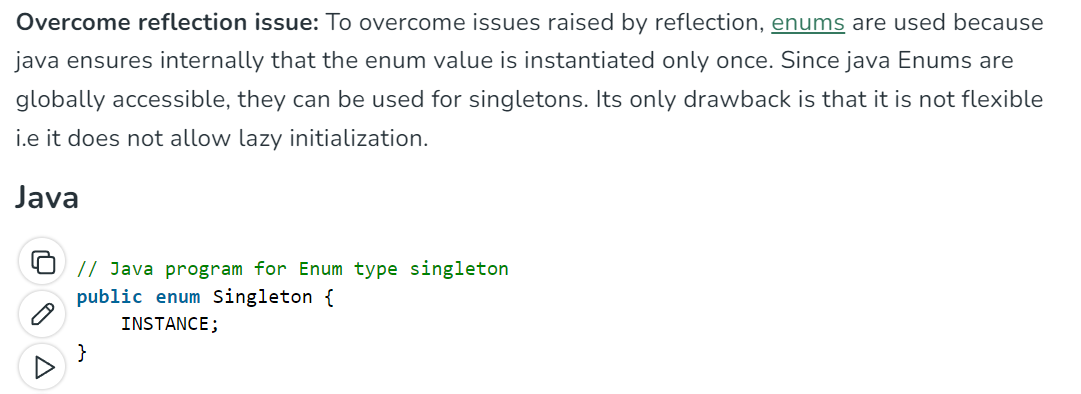
* Immutable class objects can never by modified, e.g., String, Wrapper classes, etc.
* Class and its attributes should be final.
* Attributes will be private and final so cannot be accessed and modified.
* Parameterized constructor and Getter methods should return a deep copy of the mutable objects.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



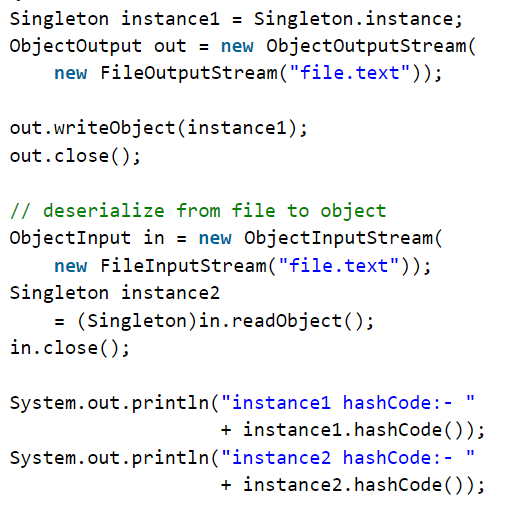
# How to break Singleton class

Using Reflection:

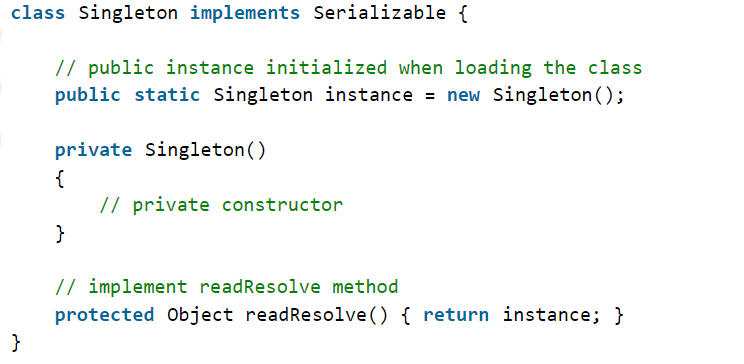




Serialization: If we serialize the Singleton class object. While deserializing, it will create new object.

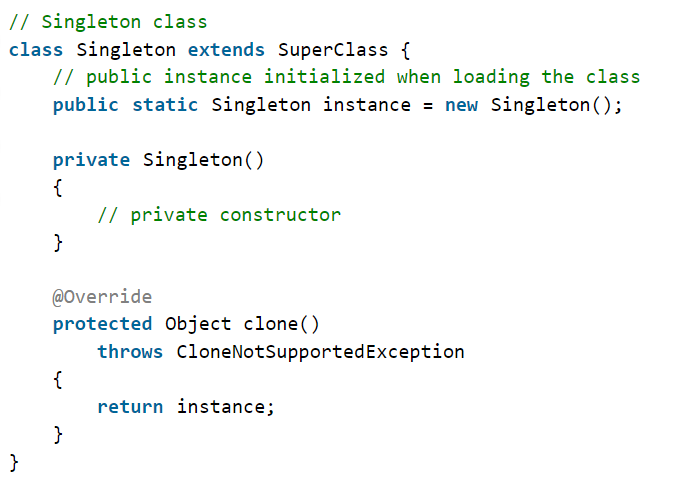


To overcome this, we have to implement readResolve() method to return the same instance.



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

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



# 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](https://www.geeksforgeeks.org/serialization-in-java/). 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

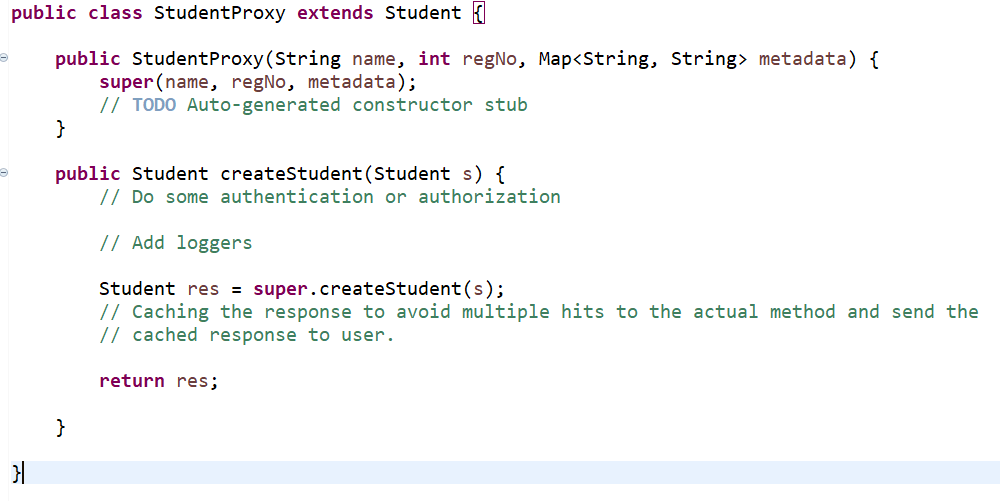
### Abstract Factory Design Pattern

## 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.



## 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:

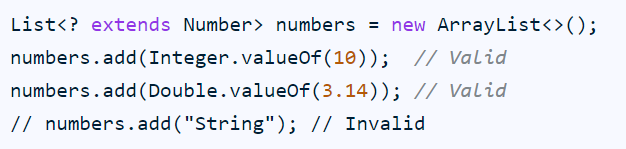
* **Unbounded Wildcard (?)**

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



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

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



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

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

