Contents

[Pre-Requisites: 3](#_Toc145585231)

[Memory Management in Java 4](#_Toc145585232)

[How memory management works 4](#_Toc145585233)

[Java (JVM) Memory Model 4](#_Toc145585234)

[Generations 6](#_Toc145585235)

[Available Collectors 6](#_Toc145585236)

[Serial Collector 6](#_Toc145585237)

[Parallel Collector 6](#_Toc145585238)

[Garbage-First (G1) Garbage Collector 6](#_Toc145585239)

[The Z Garbage Collector 6](#_Toc145585240)

[Choosing Best Garbage Collectors 7](#_Toc145585241)

[1. Memory 7](#_Toc145585242)

[2. Throughput 7](#_Toc145585243)

[3. Latency 7](#_Toc145585244)

[Concurrency and Multithreading: 7](#_Toc145585245)

[Implementing Thread: 7](#_Toc145585246)

[Callable Interface: 8](#_Toc145585247)

[Race conditions 8](#_Toc145585248)

[Synchronization 8](#_Toc145585249)

[Joins 8](#_Toc145585250)

[Executors 9](#_Toc145585251)

[Executor Interfaces 9](#_Toc145585252)

[Executor 9](#_Toc145585253)

[ExecutorService 9](#_Toc145585254)

[Thread Pools 9](#_Toc145585255)

[Fork/Join 9](#_Toc145585256)

[Immutable Objects 9](#_Toc145585257)

[Java 8 Features 10](#_Toc145585258)

[Default Methods 10](#_Toc145585259)

[Static Method 10](#_Toc145585260)

[Optional class 11](#_Toc145585261)

[Lambda Expressions 11](#_Toc145585262)

[Functional Interface and Lambda 11](#_Toc145585263)

[Functional Interfaces in Java Library 12](#_Toc145585264)

[Consumer Interface 12](#_Toc145585265)

[Bi-Consumer interface: 13](#_Toc145585266)

[Supplier interface: 13](#_Toc145585267)

[Function Interface: 14](#_Toc145585268)

[Bi-Function interface: 14](#_Toc145585269)

[Streams 15](#_Toc145585270)

[Streams vs Collection 15](#_Toc145585271)

[Reflection 16](#_Toc145585272)

[Unit Testing & TDD 17](#_Toc145585273)

[Junit 17](#_Toc145585274)

[DesignPatterns**:** 18](#_Toc145585275)

[Singleton 19](#_Toc145585276)

[Factory 19](#_Toc145585277)

[Observer 20](#_Toc145585278)

[Strategy 20](#_Toc145585279)

[Decorator 20](#_Toc145585280)

# Pre-Requisites:

* JVM
* JRE
* Class & Object
* Anonymous class
* Method
* Process & Threads

Part -1

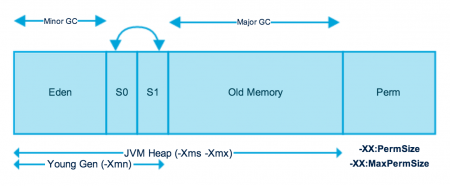
# Memory Management in Java

The process of allocating and releasing memory resources for objects in a Java program

## How memory management works

* Automatic Garbage Collection
* Heap and Stack
  + Objects are dynamically allocated.
* Object Creation and Deletion
  + Primitive types and object references
* Finalization
* Reference Types:
* OutOfMemoryError
* Memory Leaks

### Java (JVM) Memory Model

[](https://journaldev.nyc3.digitaloceanspaces.com/2014/05/Java-Memory-Model.png)

As you can see in the above image, JVM memory is divided into separate parts. At broad level, JVM Heap memory is physically divided into two parts - Young Generation and Old Generation.

heap size is bigger than stack because heap is the main region for holding objects.

A blue squares with white text

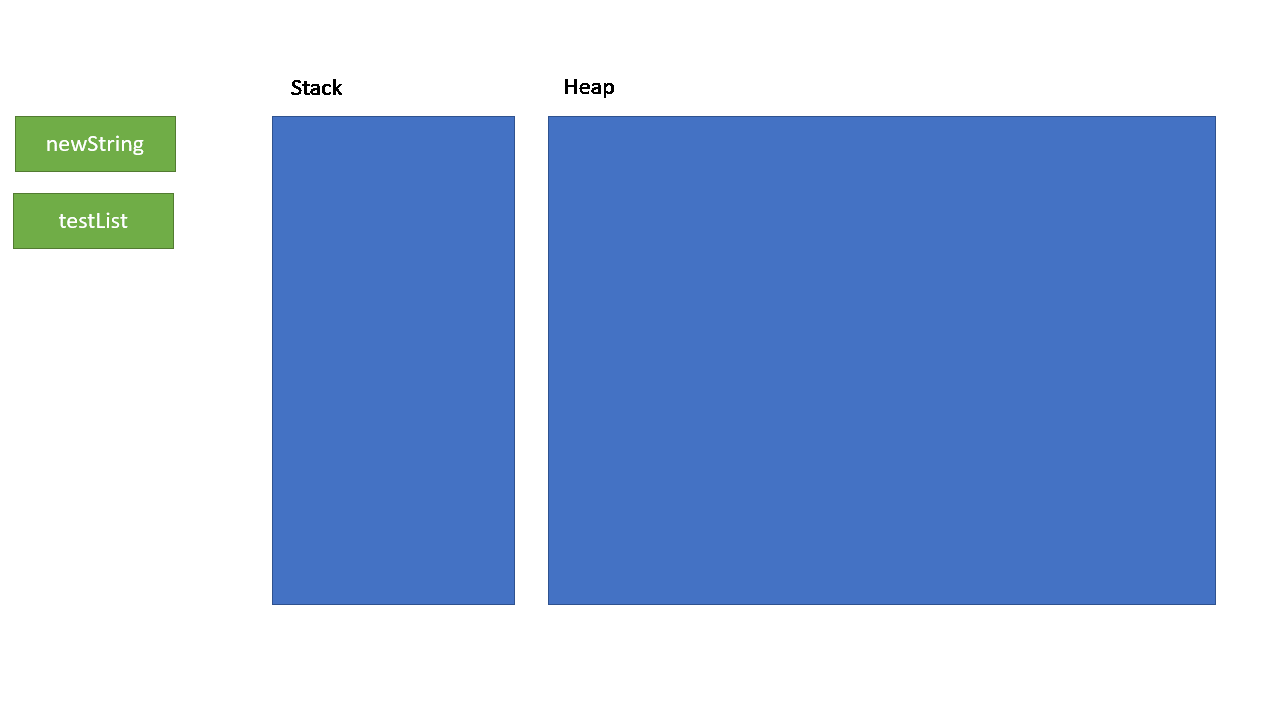
Description automatically generated

Removing Objects (FIFO)

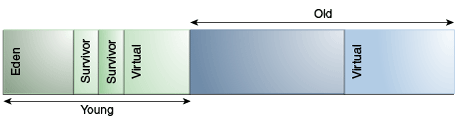
A black background with a black square

Description automatically generated with medium confidence

public List<String> test() {  
 String newString = "test";  
 List<String> testList = new ArrayList<>();  
 testList.add(newString);  
 return testList;  
 }



## Generations



* At startup, the Java HotSpot VM reserves the entire Java heap in the address space but doesn't allocate any physical memory for it unless needed.
* The entire address space covering the Java heap is logically divided into young and old generations.

## Available Collectors

The Java HotSpot VM includes three different types of collectors, each with different performance characteristics.

### Serial Collector

* + Uses a single thread to perform all garbage collection work,
  + Best-suited to single processor machines because it can't take advantage of multiprocessor hardware,
  + Selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option -XX:+UseSerialGC.

### Parallel Collector

* + Also known as *throughput collector*,
  + Similar to the serial collector
  + The primary difference is: parallel collector has multiple threads that are used to speed up garbage collection.
  + Intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. You can enable it by using the -XX:+UseParallelGC option.

### Garbage-First (G1) Garbage Collector

* + G1GC has been available since JDK 7 and is designed to provide high throughput while maintaining predictable pause times and is now default in JDK 17
  + Instead of having specific young and old generations for Heap this collector uses its entire heap and divides it into multiple regions.
  + Garbage collection is performed on each region independently. This allows G1GC to more efficiently manage large heaps and reduce the frequency and duration of stop-the-world garbage collection pauses.
  + When we start our application, we can pass on this variable that the maximum pause time (maxTargetPauseTime) that our application can withstand say 10ms for example. The G1 collector will try to ensure that the Garbage collection is done only for 10 ms and even if there is some garbage left it will take care in the next cycle.

### The Z Garbage Collector

* + A newer algorithm introduced in JDK 11 that focuses on providing low pause times even for very large heaps, making it suitable for applications with high memory requirements.ZGC is more flexible in configuring its size and scheme.
  + ZGC does provide lower pause times than G1GC, it also comes at a cost of higher CPU usage.
  + As a concurrent garbage collector, ZGC guarantees not to exceed application latency by 10 milliseconds, even for bigger heap sizes.

## Choosing Best Garbage Collectors

### **1. Memory**

This is the amount of memory that is assigned to the program and this is called HEAP memory.

### **2. Throughput**

* How much amount of time that code is run compared to how much amount of time your Garbage collection has run.
* For example, if your throughput is 99% that means 99% of the time the code was running and 1% of the time the Garbage collection was running.
* For any high volume applications, we want the higher throughput as much as possible in any load tests that we run.

### **3. Latency**

* Whenever the Garbage collection runs, how much amount of time our program stops for the Garbage collection to run properly.
* All these are measured in milliseconds but they can go up to a few seconds depending upon the size of the memory and the Garbage collection algorithm that we choose for our load tests.
* Ideally, we would want the LATENCY to be as low as possible or to be as predictable as possible

**Part-2**

# **Concurrency and Multithreading**:

* + A thread represents a single flow of execution in a program. In a single process, multiple threads can run concurrently, allowing tasks to be performed in parallel.
  + Threads share the same memory space and resources within a process, which allows them to communicate and share data easily.
  + Java provides a built-in Thread class for creating and managing threads.

### Implementing Thread:

1. Using Thread class
2. Using Runnable interface

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

### Callable Interface:

* + The Callable interface is similar to Runnable, but it can return a result or throw an exception. It's a part of the java.util.concurrent package.
  + It's typically used in scenarios where you want to get a result from a task running in a separate thread.

A screenshot of a computer program

Description automatically generated

### Race conditions

* When two or more threads try to access shared data or resources simultaneously, leading to unpredictable or incorrect behavior of a program

### Synchronization

Race Condition Examples:

* RaceConditionIssue.java
* RaceConditionFix\_1.java
* RaceConditionFix\_2.java
* RaceConditionFix\_3.java
* CounterWithLock

### Joins

The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,

t.join();

* Causes the current thread to pause execution until t's thread terminates.
* Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.

Thread Join Examples:

* ThreadJoinExample.java

### Executors

* Thread and Runnable object work well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

Executor Components:

## Executor Interfaces

* define the three executor object types.

### [Executor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html)

* A simple interface that supports launching new tasks.

### [ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html)

* (A subinterface of Executor, which adds features that help manage the life cycle, both of the individual tasks and of the executor itself)
* ScheduledExecutorService (a subinterface of ExecutorService)

### [Thread Pools](https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html)

* are the most common kind of executor implementation.

### [Fork/Join](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html)

* is a framework (new in JDK 7) for taking advantage of multiple processors.

Examples:

* WithExecutor.java
* WithExecutorService.java
* WithScheduledExecutorService.java

## Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   * Don't provide methods that modify the mutable objects.
   * Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Example:

* ImmutableTest.java

Part-3

# Java 8 Features

* Static and Default Methods
* Lambda Expressions
* Functional Interfaces
* Method Reference
* Streams
* Comparable and Comparator
* Optional Class
* Date/Time API

## Default Methods

* In Traditional interface, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.
* Default methods allow to add new methods to an interface that are automatically available in the implementations.

## Static Method

* Java interface static method is similar to default method except that we can’t override them in the implementation classes.
* This feature helps us in avoiding undesired results in case of poor implementation in implementation classes.

Examples:

* MyStaticMethodExample.java
* DefaultMethodTest.java

## Optional class

## Lambda Expressions

* Added in Java 8.
* Expresses an instance of the functional interface,
* provides a clear and concise way to represent a method of the functional interface using an expression.
* Any functionality, such as persisting a record, encrypting a file, or deleting a to-do list, et cetera, are all good candidates for **consumer** interface, as long as we don't expect returns from these methods

Examples:

* ThreadWithLambda.java
* WithCollection.java
* WithCustomFunctinalInterface.java

## Functional Interface and Lambda

* A functional interface:
  + Also known as "SAM interfaces" (Single Abstract Method interfaces) or "SAM types".
  + An interface that contains only one abstract method.
  + Some Examples:
    - Runnable
    - Callable
    - Comparator
    - Consumer
    - Supplier
  + Used to represent a single unit of behavior, making them suitable for use with lambda expressions and method references.
* Lambda expressions:
  + A shorthand notation to define and implement functional interfaces.
  + It's like a method without a name.
  + Primarily used to implement functional interfaces.
  + Provide a concise way to express behavior without the need to create a separate class for it.
* Refer example: VerySimpleFunctionalInterface.java

## Functional Interfaces in Java Library

### Consumer Interface

A screenshot of a computer code

Description automatically generated

* Represents a function which takes in one argument and produces a result.
* Doesn’t return any value.
* By use of Consumer, you can move the operation to one place.
* It improves code maintenance if you want to make any change in one place.

Examples:

* ConsumerExample.java

Best example:

list.forEach(new Consumer<Trade>() {

public void accept(Trade trade) {

System.out.println("Trade is : " + trade);

}

});

VS

list.forEach(trade -> {

System.out.println("Trade is : " + trade);

});

### Bi-Consumer interface:

A screenshot of a computer program

Description automatically generated

* Similar to Consumer but it takes two arguments and produces a result.
* Doesn’t return any value.

Examples:

* BiConsumerSampleApp.java
* BiConsumerCustomImpl.java
* BiConsumerExampleWithInternalImplementation.java

### Supplier interface:

A screenshot of a computer code

Description automatically generated

* Exactly opposite the consumer in that it doesn't take any input arguments but returns data out.
* It is a generic interface with one method GET which has no input arguments but returns the same type of object back.
* We may have situations like loading a list of employees in our application cache, or maybe fetching newly created trades, or simply getting some default configuration values of a customer into our application. Such requirements are well-suited with Supplier as the function.
* Supplier is a good candidate for fetching data into an application. It doesn't need any inputs but returns the results to us.

Examples:

* SupplierExample.java

### Function Interface:

* A function which takes in one argument and returns a result.
* The *Function* interface also has a default *compose* method that allows us to combine several functions into one and execute them sequentially.

**Best use-case of Function interface:**Taking the key from the user as input and searching for the value in the map/DB for the given key.

A computer screen shot of a program code

Description automatically generated

Examples:

* FunctionExample.java

### Bi-Function interface:

* It represents a function which takes in two arguments and produces a result.

A screenshot of a computer code

Description automatically generated

Examples:

SimpleBiFunctionExample.java

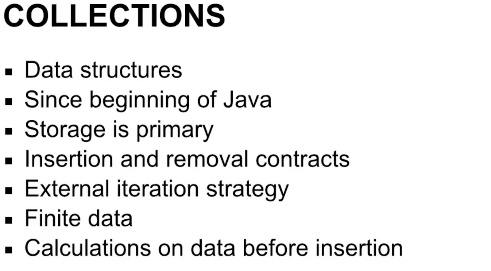
## Streams

* A new java.util.stream package
* A sequence of elements
* No storage (unlike Collections)
* Pipeline pattern
* Free parallelization out of box

A diagram of a pipeline

Description automatically generated

### Streams vs Collection

 A close-up of a text

Description automatically generated

#### Stream of Collection

Collection<String> collection = Arrays.asList("a", "b", "c");

Stream<String> streamOfCollection = collection.stream();

#### Stream of Collection

Collection<String> collection = Arrays.asList("a", "b", "c"); Stream<String> streamOfCollection = collection.stream();

#### Stream.builder()

Stream<String> streamBuilder = Stream.<String>builder().add("a").add("b").add("c").build();

#### Stream of Collection

#### Stream of Collection

Examples:

* GuessTheOutputSteram\_1.java
* JavaStreamExample.java
* FilterWithoutStream.java
* FilterWithStream.java
* ForEachWithStream.java
* MinMaxWithStream.java

Question:

In how many ways an instance can be created in Java?

**Part-4**

# **Reflection**

* Allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time.
* The required classes for reflection are provided under ***java.lang.reflect*** package

Some uses:

* Dependency Injection Frameworks:
  + Many dependency injection frameworks, like Spring, use reflection extensively to automatically wire dependencies at runtime.
* Testing Frameworks:
  + Reflection is commonly used in testing frameworks to dynamically discover and execute test methods.
* Proxy Pattern:
  + Reflection is often used in conjunction with the Proxy Pattern to create dynamic proxies. These are objects that intercept and handle method invocations on behalf of another

It's important to note that while reflection provides a lot of flexibility, it comes with a performance overhead, and it can make code less maintainable and harder to understand. Therefore, it should be used judiciously and only when truly necessary

Example:

* TestReflection.java

Part-5

# **Unit Testing** & TDD

* A software development approach where developers write automated tests before writing the actual code.
* The test is expected to fail initially.
* Then, the code is written to pass the test.
* This process is repeated for each new feature or functionality.

#### **Benefits of TDD**

* Early Bug Detection: TDD allows for the early identification of bugs, ensuring they are caught and fixed at a stage when they are easier and cheaper to resolve.
* Improved Code Quality: TDD promotes writing code in small, manageable chunks, leading to cleaner, more maintainable, and less error-prone code.
* Regression Testing: With a comprehensive suite of tests, developers can confidently make changes or additions to the codebase, knowing that existing functionality won't be unintentionally broken.
* Documentation and Specification: The tests themselves serve as living documentation, providing clear examples of how the code is intended to be used and what behavior is expected. This is particularly valuable for small teams or solo developers where knowledge sharing is crucial.

### Junit

* Widely used Java testing framework that allows developers to write and run automated tests for their Java code.
* provides a simple and organized way to verify that code behaves as expected.
* JUnit makes it efficient to test individual units of code, helping developers catch and fix bugs early in the development process.

#### Some Useful Methods

* Assertions:
  + assertEquals(expected, actual): Checks that two values are equal.
  + assertTrue(condition): Checks that a condition is true.
  + assertFalse(condition): Checks that a condition is false.
  + assertNull(object): Checks that an object is null.
  + assertNotNull(object): Checks that an object is not null.
  + assertSame(expected, actual): Checks that two references point to the same object.
  + assertNotSame(expected, actual): Checks that two references point to different objects.
  + assertArrayEquals(expectedArray, resultArray): Checks that two arrays are equal.
* Timeout and Exception Testing:
  + @Test(timeout = milliseconds): Sets a timeout for a test, ensuring it completes within a specified time.
  + @Test(expected = Exception.class): Expects a specific exception to be thrown during the test.
* Annotations:
  + @Test: Denotes a method as a test method.
  + @Before: Indicates a method that should be run before each test.
  + @After: Indicates a method that should be run after each test.
  + @BeforeClass: Indicates a method that should be run once before all tests in a class.
  + @AfterClass: Indicates a method that should be run once after all tests in a class.
* Parameterized Testing:
  + @RunWith(Parameterized.class): Allows you to run a test multiple times with different sets of data.
  + @Parameters: Provides the data sets for parameterized testing.
* Ignoring Tests:
  + @Ignore: Skips a test method.
* Assertions for Floating-Point Numbers:
  + assertEquals(expected, actual, delta): Checks that two floating-point values are approximately equal within a certain delta.

#### Examples:

* *ForUnitTestingApplicationTests.java*
* *HttpRequestTest.java*
* *TestControllerTest.java*
* *EmployeeServiceTest.java*

Part-6

# **Design Patterns**

What do you understand by design pattern?

* A general, reusable solution to a commonly occurring problem within a specific context in software design.
* it's a description or template for how to solve a problem that can be used in many different situations.
* Design patterns are not specific to any particular programming language. They are high-level concepts and guidelines for designing software in a way that promotes maintainability, scalability, and flexibility.
* The idea of design patterns originated from the field of architecture, where architects would document common design solutions to common problems. In software development, design patterns were popularized by the book "Design Patterns: Elements of Reusable Object-Oriented Software" by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides).

### **Singleton**

* Ensures a class has only one instance and provides a global point of access to that instance. It is useful when you want to restrict the instantiation of a class to a single object.

**Way 1:** Eager Initialization (Thread-Safe)

* Instance is created when the class is loaded,
* regardless of whether it's needed or not, the instance is created,
* Ensures thread safety,
* Example: *EagerInitializationSingleton.java*

**Way 2:** Lazy Initialization (Not Thread-Safe):

* Instance is created when it is first requested.
* Not thread-safe
* Ensures thread safety
* Example: *LazyInitializationSingleton.java*

**Way 3: Synchronized method:**

* Example: *LazyInitializationSingleton.java*

**Way 4: Synchronized block:**

* Example: *LazyInitializationSingleton.java*

**Prevent Serialization:**

protected Object readResolve() {

return getInstance();

}

**Prevent Cloning:**

@Override

protected Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException("Clone is not allowed.");

}

Example: *FinalCompleteSingleton.java*

### **Factory**

* The client: code needs to create an object but does not need to know the details of how the object is created. (e.g. Juice Buyer)
* Factory class: responsible for creating objects. (e.g. MakeJuice(orange/apple/banana)
* Product: This is the interface or base class that represents the type of objects the factory can create. The concrete products are implementations of this interface or inherit from this base class. (e.g. Juice)
* Example: *ShareFactory.java*

### **Observer**

### **Strategy**

### **Decorator**

# **Behavioral Pattern**

* Subset of design patterns that deal with how objects interact and communicate with each other.
* These patterns focus on the delegation of responsibilities among objects and the ways they collaborate. Below are some common behavioral design patterns in Java:

1. Observer Pattern:

* Allows a subject to notify its observers about state changes without them being tightly coupled.
* Examples: Java's built-in ***java.util.Observer*** and ***java.util.Observable***, and the Spring Framework's event listeners.

1. Strategy Pattern:

* Defines a family of interchangeable algorithms and lets the client choose the one to use at runtime.
* Example: Java's ***java.util.Comparator*** interface used for sorting collections.

1. Command Pattern:

* Encapsulates a request as an object, thereby allowing for parameterization of clients with requests, queuing of requests, and logging of requests.
* Example: Swing's ***Action*** and ***ActionListener*** classes for handling UI actions.

1. Chain of Responsibility Pattern:

* Passes a request along a chain of handlers. Each handler decides either to process the request or to pass it to the next handler in the chain.
* Example: Java's exception handling mechanism with multiple catch blocks.

1. Iterator Pattern:

* Provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.
* Example: Java's ***Iterator*** interface for traversing collections.

1. State Pattern:

* Allows an object to alter its behavior when its internal state changes, making it appear as if the object changed its class.
* Example: Swing's ***JComponent*** states like ***setEnabled(),*** ***setVisible()***, etc.

1. Template Method Pattern:

* Defines the skeleton of an algorithm in a method, allowing subclasses to provide specific implementations of some steps.
* Example: Java's ***AbstractList*** class, where concrete subclasses provide implementations for ***get(), size()***, etc.

1. Visitor Pattern:

* Represents an operation to be performed on elements of an object structure. It lets you define a new operation without changing the classes of the elements.
* Example: The Java Document Object Model (DOM) API, which uses visitors to traverse XML documents.

1. Memento Pattern:

* Captures and externalizes an object's internal state so the object can be restored to this state later.
* Example: The serialization and deserialization of objects in Java.

1. Interpreter Pattern:

* Defines a grammar for interpreting a language and provides an interpreter to execute the language's grammar.
* Example: Java's regular expression ***(java.util.regex)*** package.

These behavioral design patterns help in improving the communication and collaboration between objects in your Java applications, making the code more maintainable and flexible. Depending on the specific problem you're trying to solve, you can choose the appropriate pattern to apply.