Day 03

- Generics
- Collections Framework
- Lambda Expressions and Functional Interfaces
- Java Database Connectivity (JDBC)
- Junit With Mockito

Generics

Let’s generalize it

<T>

Generics

Generics means **parameterized types**.
Generics allow various types (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces.
Using Generics, it is possible to create classes that work with different data types.
An entity such as class, interface or method that operates on a parameterized type is a *generic entity*.
Generics works only with reference type. We can not use primitive types like char, int etc

Generics (Contd)

Programs that use Generics has got many benefits over non-generic code.

- ✓Code Reusability
- ✓Type Safety
- ✓Avoids individual type casting

```
public class MyCustomList<T> {
        ArrayList<T> list = new ArrayList<>();
        public void addElement(T element) {
                list.add(element);
        }
        public void removeElement(T element) {
                list.remove(element);
        }
        public String toString() {
                return list.toString();
        }
}
```

Collection Framework

A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

Why Collection Framework?

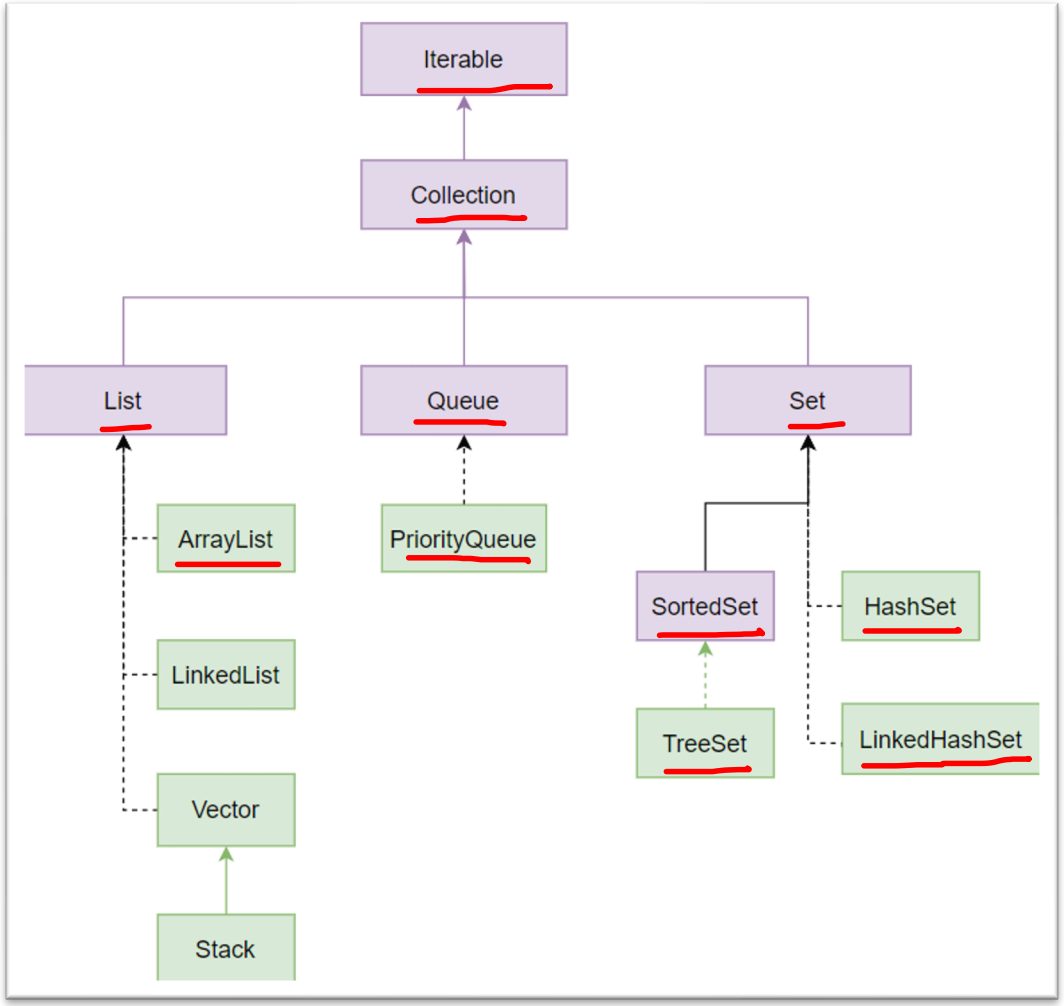
Before the Collection framework, it was very difficult for programmers to write algorithms that can work for all kinds of Collections (Arrays, Vector, Hashtable).

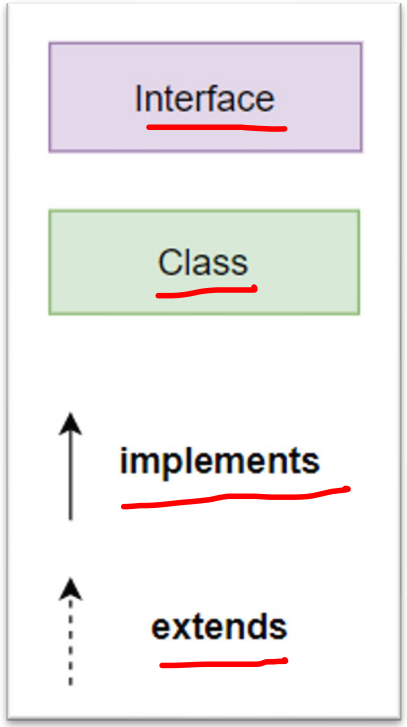
Java developers decided to produce a common interface to deal with the inconsistencies in collections and introduced the Collection Framework in JDK 1.2

Following are the advantages of the collection framework-

- The API has a basic set of interfaces like *Collection*, *Set*, *List*, or *Map* to bring consistency
- We do not have to worry about the design of the Collection but rather we can focus on its best use in the program.
- We can simply use the best implementation to drastically boost the performance of the algorithm/program.

Hierarchy Of Collection Framework

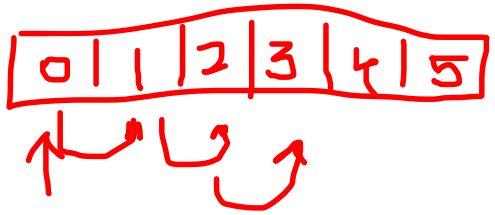




The List Interface

The List interface is used to implement an ordered collection in Java programs.

- An ordered collection is one, where the programmer has control over the position where an element can be inserted into, or accessed from the collection.
- If an element's insertion position is not specified, it is added at the end of the List.
- A List typically allows duplicate elements.
- Lists created using the static **of** method are immutable.
- The way to create List data structures that can be updated over time, is to instantiate built-in collection classes that implement the List interface.
- Examples are ArrayList, LinkedList and Vector.



ArrayList vs LinkedList

ArrayList

ArrayList uses an array as data structure to store elements.

- Positional access and modification of elements is very efficient, with constant-time algorithmic complexity.
- Insertion and deletion of elements are expensive. In a 20 elements list, to insert an element at first position, all 20 elements should be moved.

LinkedList

The data structure used to implement is of type linked-list.

- Inserting and Deleting values is easy. This is because in a chain of blocks, each link is nothing but a reference to the next block. Insertion only involves adjustment of these links to accommodate new values, and does not require extensive copying and shifting of existing elements.
- Positional access and modification of elements is less efficient than in an ArrayList, because access always involves traversal of links.

ArrayList vs Vector

Vector has been with Java since v1.0, whereas ArrayList was added later, in v1.2.

Both of them use an array as the underlying data structure.

Vector is thread-safe. In Vector, all methods are ***synchronized***.

List : Operations

Let’s examine common List operations on an ArrayList. Similar operations can also be done on a LinkedList or a Vector.

- Insertion at end (default)
- Positional insertion
- Inserting duplicate entries is allowed
- Modifying elements at a specific position.
- Deleting elements
- Iterate around the contents
- Sorting the List (sort() in Collection Interface)

Comparable & Comparator

Java provides two interfaces to sort objects using data members of the class.

Comparable — override the method compareTo()

- The class itself must implements the Comparable interface to compare its instances.
- Comparing itself with another object
- We get only one chance to implement the compareTo() method.

Comparator — override the method compare()

- Comparator is external to the element type we are comparing
- We create multiple separate classes (that implement Comparator) to compare by different members.
- Collections class has a second sort() method and it takes Comparator.

The Set Interface

Java Set interface specifies a contract for collections having unique elements.

- If `object1.equals(object2)` returns *true*, then only one of `object1` and `object2` have a place in a Set implementation.
- Similar to List, `of()` provides immutable Set
- Create a HashSet collection instead, which supports the add(), to test the uniqueness property of a Set.
- Set collection does not give any importance to element order, and therefore, does not support positional access.

Set Implementation

HashSet	In a HashSet, elements are neither stored in the order of insertion, nor in sorted order.
LinkedHashSet	In a LinkedHashSet, elements are stored in the order of insertion.
TreeSet	In a TreeSet, elements are stored in sorted order.

Task : Set

Create a ``List`` of characters, such as:

```
List<Character> list = List.of('A', 'Z', 'A', 'B', 'Z', 'F');
```

- ❑ Write a procedure to list out the unique characters in this list
- ❑ Write a procedure to list out these unique characters in sorted order
- ❑ Write a procedure to list out these unique characters in the order in which they were present in the original list

The Queue Interface

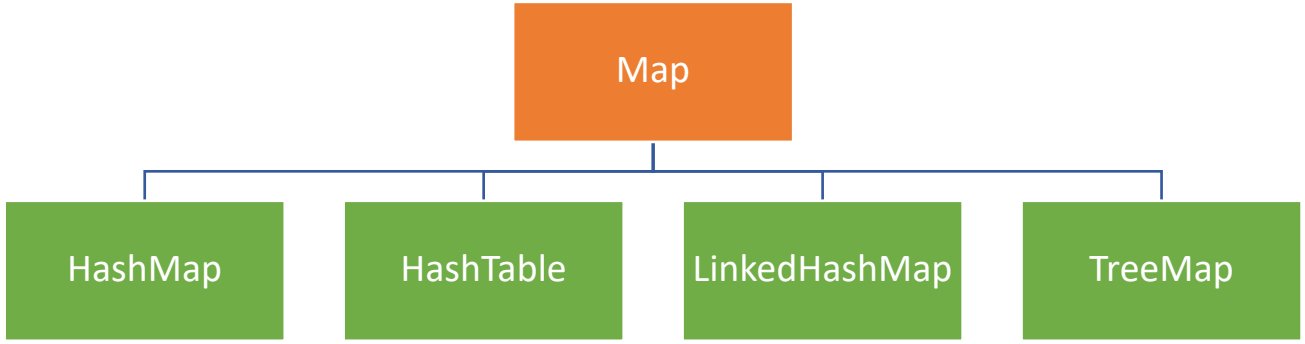
Elements are arranged in order of processing, such as in a To-Do List.

- The `PriorityQueue` collection is a built-in Java class, that implements the Queue interface.
- Elements are stored in a sorted natural order, by default.
- We can also specify a different custom order, called *the order of priority.*
- A custom priority order on the elements in a `PriorityQueue` can be specified by passing an implementation of `Comparator` interface to the `PriorityQueue<T>` constructor

The Map Interface

The Map interface specifies a contract to implement collections of elements, that are in the form of (key, value) pairs.

- Since the kind of elements stored in a map (<key, value> pairs) are different from any other collection categories in Java, the `Map` interface is unique. So unique, that it even does not extend the `Collection` interface.




```
public interface Map<K, V> {
        //Method Declarations
}
```

Map Collection : Concepts

HashMap	HashTable	LinkedHashMap	TreeMap
• Unordered • Unsorted • Key's hashCode() value is used in the hashing function • Allows a key with a null value.	• Thread-safe version of HashMap. Has synchronized methods where required. • Unordered • Unsorted • Key's hashCode() value is used in the hashing function • Does not allow a null key.	• Insertion order of elements is maintained (which is optional as well) • Unsorted • Iteration is faster • Insertion and Deletion are slower	• Additionally implements the NavigableMap interface • Elements are maintained in sorted order of keys

Exercise : Map

Given the string: "*This is an awesome occasion. This has never happened before.",* do the following processing on it:

1. Find the number of occurrences of each unique character in this string
2. Find the number of occurrences of each unique word in this string

Collections : Conclusion with Tips

When you use a "Hashed" Java collection (hash table based), such as `HashMap` or `HashSet`, it will be unordered, unsorted and will iterate over its elements in no particular order.

When you encounter a "Linked" Java collection (linked list based), such as a `LinkedHashSet` or a `LinkedHashMap`, it will maintain insertion order but will be unsorted.

When we make use of a "Tree"-based Java collection (stored in a tree data structure), such as `TreeSet` or `TreeMap` it always maintains natural sorted order. It would implement one of the navigable category of interfaces, such as `NavigableSet` or `NavigableMap`

Functional Programming

Functions are first class citizens

Functional Programming

It is a declarative style of programming rather than imperative

The basic objective of this style of programming is to make code more concise, less complex, more predictable, and easier to test

Functional programming deals with “What” as compared to “How” in traditional programming style.

Functional Programming(Contd)

Higher Order Functions

1. We can pass a function to a function
2. We can create a function within function
3. We can return a function from a function

Lambda Expressions

- A Lambda expression is an anonymous method
- It has only parameter list and a body
- The return type is always inferred based on the context
- Lambda expressions work in parallel with the functional interface.

```
(parameter) -> body
```

Functional Programming : Streams

Streams

- A Stream is a sequence of elements.
- You can perform different kinds of operations on a stream.

Types of Streams

- **Intermediate Operations**: An operation that *takes a stream* - for example, applies a lambda expression - and produces *another stream* of elements as its result. For example – map, filter, sorted.
- **Terminal Operations**: A stream operation that takes a stream - for example, applies a lambda expression - and returns a single result. reduce() and forEach() are a couple of such operations.

Exercise : Functional Programming

- Write a program to print the squares of the first 10 positive integers.
- Create a list of the strings "Apple", "Banana" and "Cat". Print all of them in lower-case.
- Create a list of the character strings "Apple", "Banana" and "Cat". Print the length of each string.

Functional Programming : Behind the Scene

When we define a lambda expression, a lot of things happen behind the scenes.

- An important concept is a *functional interface*.
- A **functional interface** is an interface that contains only Single Abstract Method (SAM)
- Functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations.

Consumer

Predicate

Function

Supplier

Functional Programming : Behind the Scene

Consumer	Accepts only one argument
	The consumer interface has no return value. It returns nothing.
	This implementation of the Java Consumer functional interface can be seen using forEach()
Predicate	A predicate functional interface of java is a type of function which accepts a single value or argument and does some sort of processing on it and returns a boolean (True/ False) answer.
	The implementation of the Predicate functional interface also encapsulates the logic of filtering in Java.

Functional Programming : Behind the Scene

Function	Receives only a single argument and returns a value after the required processing.
	This implementation of the Java Function interface can be seen using map()
Supplier	A type of functional interface that does not take any input or argument and yet returns a single output.
	This type of functional interface is generally used in the lazy generation of values

JDBC

Let’s Connect With Database

JDBC & JDBC Drivers

JDBC (Java Database Connectivity) is a Java API for connecting and interacting with databases

JDBC Drivers are software components that provide necessary functionality to connect Java applications to different types of databases

Four types of JDBC Drivers -

- Type 1: JDBC-ODBC Bridge Driver
- Type 2: Native API Partly Java Driver
- Type 3: Network Protocol Pure Java Driver
- Type 4: Thin Driver (known as Direct to Database Pure Java Driver)

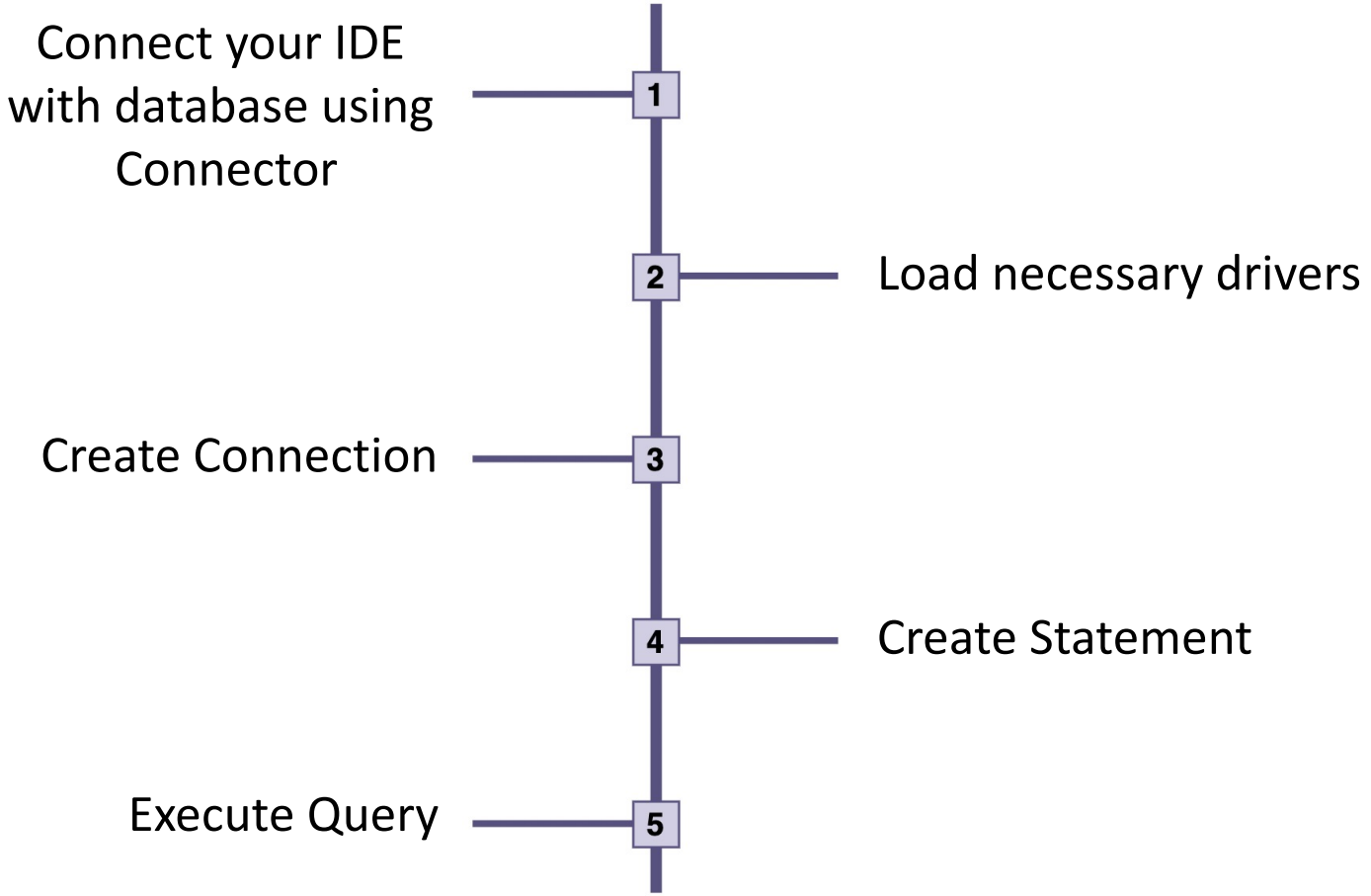
JDBC Components

In addition to JDBC Drivers, there are several other components that make up JDBC API including -

- `DriverManager Class`
- `Connection Interface`
- `Statement and PreparedStatement Interfaces`
- `ResultSet Interface`

These components work together to provide a powerful and flexible API for working with databases in Java

JDBC Program Flow



Testing

Assertions, Matchers, Stub, Mocking and much more

JUnit Introduction

JUnit is a robust Java testing framework that simplifies creating reliable and efficient tests.

Offers a suite of features including support for diverse test cases, robust assertions, and comprehensive reporting capabilities.

Two main objectives of using Junit testing -

- Ensures that the software behaves as intended.
- Catch errors in the code early on.

Different Types of Tests

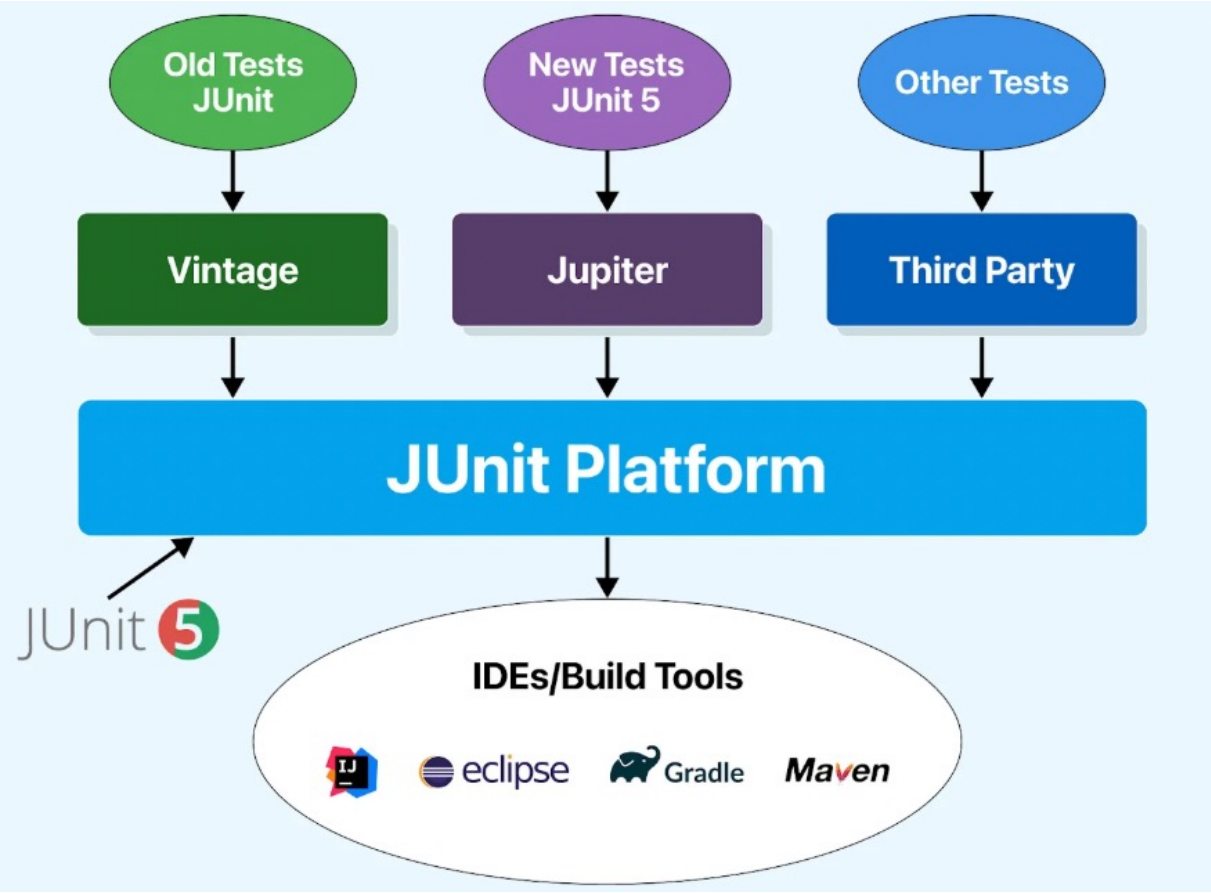
Unit Tests	Integration Tests	System Tests
Unit tests focus on individual code snippets within a class or method.	Integration tests assess how all the components work together.	System tests examine entire systems like web servers.

JUnit 5 - Architecture

JUnit Platform provides a launching mechanism for testing frameworks on the JVM.

JUnit Jupiter component provides new programming techniques for developing the test cases in JUnit 5.

JUnit Vintage main functionality is allowing JUnit 3 and JUnit 4 Test cases on the JUnit 5 Platform.



Features Of JUnit

IDE Integration	Annotation Usage	Assertion Support
Quality Code Assurance	HTML Test Reports	CI/CD Compatibility
Visual Feedback	User-Friendly Interface	Automation and Feedback

Annotations

`@Test`	Marks a method as a test method.
`@BeforeEach`	Indicates that the annotated method should be executed before each test.
`@AfterEach`	Indicates that the annotated method should be executed after each test.
`@BeforeAll`	Indicates that the annotated method should be executed before all tests in the test class.
`@AfterAll`	Indicates that the annotated method should be executed after all tests in the test class.
`@DisplayName`	Provides a custom name for the test class or test method.
`@Disabled`	Disables the test method or class.

Assertions

Assertions are used to check if a condition is true. If the condition is false, the test fails.

assertEquals(expected, actual)	Checks if two values are equal.
assertTrue(condition)	Checks if a condition is true.
assertFalse(condition)	Checks if a condition is false.
assertNotNull(object)	Checks if an object is not null.

Mockito - Introduction

Mockito is an open-source test automation framework that internally uses Java Reflection API to create mock objects.

The main purpose of using a dummy/mock object is to simplify the development of a test by mocking external dependencies and using them in the testing code.

We should mock objects in unit tests when the real object has a non-deterministic behaviour or the real object is yet to be implemented

In Mockito, we can create mock objects in two ways:

1. Using `mock()` method
2. Using `@Mock` annotation

Day 03

Conclusion

- Generics
- Functional Programming
- Collection Framework
- JDBC
- Writing test using JUnit & Mockito