**Abstract class vs Interface**

| Points | Abstract Class | Interface |
| --- | --- | --- |
| Type of Methods | Can have both abstract and concrete methods | Can have only abstract methods (until Java 7), and from Java 8, can have default and static methods, and from Java 9, can have private methods. |
| Variables | Can have final, non-final, static, and non-static variables. | Only static and final variables |
| Inheritance | Can extend only one class (abstract or not) | A class can implement multiple interfaces |
| Constructors | Can have constructors | Cannot have constructors |
| Implementation | Can provide the implementation of interface methods | Cannot provide the implementation of abstract class methods |

**Multithreading**

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

**Advantages**

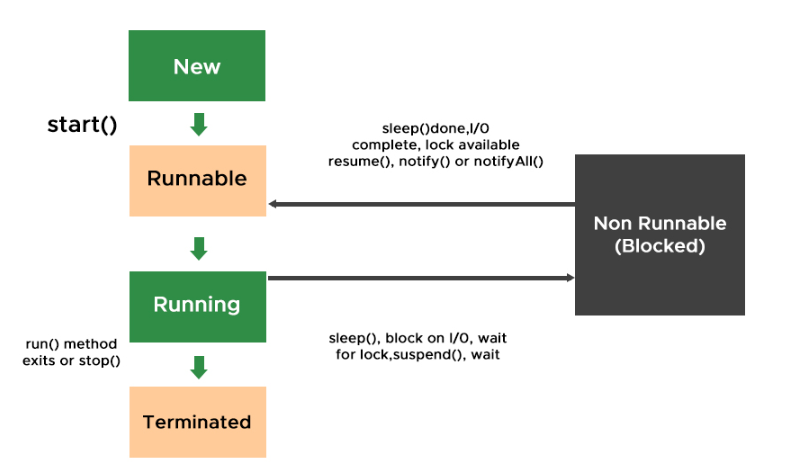
) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

**Life cycle of a Thread**

1. **New**: When a thread is just created.
2. **Runnable**: When a start() method is called over thread processed by the thread scheduler.
   * Case A: Can be a running thread
   * Case B: Can not be a running thread
3. **Running**: When it hits case 1 means the scheduler has selected it to be run the thread from runnable state to run state.
4. **Blocked**: When it hits case 2 meaning the scheduler has selected not to allow a thread to change state from runnable to run.
5. **Terminated**: When the run() method exists or stop() method is called over a thread.



**Two Ways to Implement Multithreading**

1. **Using Thread Class**

Java provides Thread class to achieve programming invoking threads thereby some major methods of thread class.

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.
23. **Using Runnable Interface**

**java.lang.Runnable**is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass Thread and implement Runnable. There is no need to subclass a Thread when a task can be done by overriding only the**run()** method of Runnable.

**Daemon Thread**

In Java, daemon threads are **low-priority threads** that run in the background to perform tasks such as garbage collection or provide services to user threads. The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.

**Thread Priority**

Java being Object-Oriented works within a Multithreading environment in which the thread scheduler assigns the processor to a thread based on the priority of the thread

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

**Functional Interfaces**

A function interface in Java is an interface with only one abstract method. Whereas interfaces from traditional descriptive conventions implement classes, functional interfaces focus on behavior, and allow developers to treat tasks as objects. They form the cornerstone of Java programming features, enabling the use of lambda syntax and style.

@FunctionalInterface annotation is used to ensure that the functional interface cannot have more than one abstract method.

Few of the Predefined-Functional Interfaces

**Supplier:** Represents a supplier of results.

**Consumer:** Represents an operation that accepts a single input argument and returns no result.

**Predicate:** Represents a predicate (boolean-valued function) of one argument.

**Function:** Represents a function that accepts one argument and produces a result.

**Lambda Expressions**

Lambda expressions in Java help to implement functional interfaces more concisely and expressively, thereby eliminating lengthy boilerplate code and strengthening readability.

**Java 8 Stream**

Introduced in Java 8, Stream API is used to process collections of objects. A stream in Java is a sequence of objects that supports various methods that can be pipelined to produce the desired result.

*Stream API is a way to express and process collections of objects.*

*Enable us to perform operations like filtering, mapping, reducing, and sorting.*

Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

1. **Intermediate Operations**

Intermediate operations transform a stream into another stream.

It enables the concept of filtering where one method filters data and passes it to another method after processing

Ex: map(), filter(),sorted(), distinct()..etc.

1. **Terminal Operations**

Terminal Operations are the type of Operations that return the result. These Operations are not processed further just return a final result value.

Ex : Collect(), forEach(), reduce(),count(),anyMatch()

**Optional Class**

Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

**Junit**

It is an *open-source testing framework* for java programmers. To perform unit testing, we need to create test cases. The unit test case is a code which ensures that the program logic works as expected.

@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**

Junit provides different methods in Assertions class for checking the expected Result. 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.

**Assumptions**

Assumptions in JUnit 5 provides a good way for checking the test cases conditionally based on preconditions and one more is if Assumptions is failed then it is marked as skipped rather then failed. Means Assumptions are used to when you want to skip a test then we use Assumptions in JUnit 5.

**assumeTrue()**this method is used to skip the test when a specified condition is not true for assumeTrue or not false for assumeFalse.

**assumingThat()** this method is allowing us, to execute a block of code based on a Boolean Assumptions, If the Assumptions is false the block is skipped.

**Parameterized Test**

This Parameterized Test is used to test a Test case with different parameters for this we use **@ParameterizedTest**annotations.

**Dynamic Tests**

JUnit 5 introduces the concept of dynamic tests, which can be generated at runtime. These are created using the **DynamicTest class**.

For creating Dynamic Tests in Run time by using **@TestFactory** annotation. This TestFactory provides a feature to create dynamic test case in the run time of the Application.

**Tagging and Filtering**

We can tag our test cases by using @tag annotation in the JUnit 5. Simply the Tags are labels for categorize the test cases. And Filter is used to filter and run the test cases by using the Tags

**Mockito Framework**

Mocking is a process of developing the objects that act as the mock or clone of the real objects. In other words, mocking is a testing technique where mock objects are used instead of real objects for testing purposes. Mock objects provide a specific (dummy) output for a particular (dummy) input passed to it.

**mocking concepts**

**Stub:** Stub objects hold predefined data and provide it to answer the calls during testing. They are referred to as a dummy object with a minimum number of methods required for a test. It also provides methods to verify other methods used to access the internal state of a stub, when necessary. Stub object is generally used for **state verification**.

**Fake:** Fake are the objects that contain working implementations but are different from the production one. Mostly it takes shortcuts and also contains the simplified version of the production code.

**Mock:** Mock objects act as a dummy or clone of the real object in testing. They are generally created by an open-source library or a mocking framework like Mockito, EasyMock, etc. Mock objects are typically used for **behavior verification**.