1) **Interning:** We can manually add a string to the Pool. It creates an exact copy of the heap string object in the String Constant Pool.

String s4 = new String("Hello").intern(); // Ensures use of the string pool

**2) Why String objects are immutable in Java?**

Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

**1. ClassLoader:**

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

**2. Thread Safe:**

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

**3. Security:**

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

**4. Heap Space:**

The immutability of String helps to minimize the usage in the heap memory.

**3) Why String class is Final in Java?**

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.

4) **String compare by compareTo() Method**

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

**5) How to create Immutable Class in Java?**

We can also create an immutable class by creating a final class that has final data members, no setter method. the final variable will be initialized during object creation.

6) **How to optimize string creation in Java?**

 String creation using literals take less time.

**7)** **Anagram String**

Two strings are called anagrams if they contain same set of characters but in different order.

**8)What is exception propagation in Java?**

The process of passing an exception from one method to its calling method

**9) Can a finally block contain a return statement?**

Yes, but using a return statement inside the finally block can override any return statements in the try or catch blocks, which may lead to unexpected behavior.

**10. What happens if an exception occurs inside the finally block?**

In Java, if an **exception occurs inside the finally block**, it behaves like any other exception — **it will be thrown**, and if not caught, it will **propagate** up the call stack.

**11. Why Use throw in Java?**

* Validate inputs (for example, throwing an exception if user input is invalid).
* Handle specific conditions (for example, throwing exceptions if a requested resource is not found).
* Create more meaningful error messages that are specific to the application's context.
* Improve debugging by providing custom error information.

**Note**: If we throw an unchecked exception from a method, it is not required to handle the exception or declare it in a throws clause. However, for checked exceptions, handling or declaration in the throws clause is mandatory."

**Note**: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

**12. Create a Custom Exception**

You can extend either:

* Exception → for **checked exceptions**
* RuntimeException → for **unchecked exceptions**

**Note**: By default, Checked Exceptions are not forwarded in calling chain (propagated).

You must **explicitly declare** the propagation using the throws clause.  
Otherwise, the compiler throws an error.

**13. Which exception should be declared?**

**Ans:** Checked exception only, because:

* Unchecked exceptions are under our control so that we can correct our code.

14.**Can we rethrow an exception?**

* **Ans:** Yes, by throwing the same exception in the catch block.

**15. What happens if a method declares an exception using throws but does not handle it?**

* **Ans:** The calling method must either handle the exception using a try-catch block or propagate it further using throws.

**16. Can a method use both throw and throws keywords in its definition?**

No. A method can either use the throw keyword to throw an exception or the throws keyword to declare that it may throw an exception, but not both simultaneously.

**17. Exception Handling with Method Overriding**

* **If the superclass method does not declare an exception**
  + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* **If the superclass method declares an exception**
  + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

18. **What happens if we do not call super() inside the custom exception constructor?**

It will compile and run, but the exception message will be null. i.e. getMessage() Method

**19. what is the purpose of calling super(str) in Custom Exception class constructor?**

It passes the error message to the constructor of the Exception class. which allows the message to be retrieved later using getMessage().

**20.** **Multithreading**

we use multithreading rather than multiprocessing because threads use a shared memory area.

Note: At least one process is required for each thread.

**21. Thread States**

* **New:** A thread that has been created but not yet started. A thread is in this state after it's created with new Thread(), but before calling start().
* **Runnable or Running:** After calling start(), the thread moves to RUNNABLE. It is **ready to run**, but the OS scheduler decides **when** to run it.

When the thread gets the CPU, it moves from the runnable to the running state.

**Waiting:** A thread that is waiting indefinitely for another thread to perform a task.

**Terminated:** A thread that has completed its task.

22. **BLOCKED State**

The thread is waiting to acquire a lock.  When a thread has acquired a lock on an object, another thread attempting to acquire the same lock will enter the BLOCKED state until the lock is released.

23. **Interrupt()**

The interrupt() method requests a thread to stop its execution and **do something else**. but the thread can safely ignore it if it is not designed to handle interruptions.

**24. Thread Scheduler**

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.

**25. Working of the Java Thread Scheduler**

* The **thread scheduler** picks the thread with the **highest priority** to run first.
* If a **high-priority thread becomes ready** while a lower-priority thread is running, the lower-priority thread **gets paused**, and the high-priority one runs instead.

When two threads have the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

**26. Thread.sleep()**

The method sleep() is being used to halt the working of a thread for a given amount of time.

If executing threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments.

**27. Daemon Thread in Java**

**A daemon thread in Java** is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

**Note**: If you want to make a user thread as Daemon, it must not be started otherwise it will throw an IllegalThreadStateException.

**27. ExecutorService :** The ExecutorService helps maintain a pool of threads and assigns them tasks.

It also provides the facility to queue up tasks until there is a free thread available if the number of tasks is more than the threads available**.**

|  |  |
| --- | --- |
| **Method** | **What it does** |
| submit(Runnable task) | Run task without returning a result. |
| submit(Callable<T> task) | Run a task that returns a result |
| invokeAll(Collection<Callable<T>> tasks) | Executes multiple Callables and waits for **all** to finish. Returns a List. |
| invokeAny(Collection<Callable<T>> tasks) | Executes multiple Callables and returns result of **any one that completes first**. |

28. How to Sort HashMap by Value

**28. Thread Pool**

A Thread Pool in Java is a collection of pre-initialized reusable threads that can execute tasks concurrently. It helps manage system resources efficiently by reusing threads instead of creating new ones for each task.

Java provides the **ExecutorService** interface and utility methods in **Executors** class to create a Threadpool.

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.execute(() -> {})

**Types of Thread Pools (via Executors)**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Executors.newFixedThreadPool(n) | Fixed number of threads |
| Executors.newCachedThreadPool() | Dynamically adjusts number of threads (good for many short tasks) |
| Executors.newSingleThreadExecutor() | One thread only, tasks executed sequentially |
| Executors.newScheduledThreadPool(n) | Supports scheduled and periodic task execution.Ex:-  1) To send a reminder email **5 seconds after user login**  2) Or to **check server health every 10 seconds** |

1. **shutdown(){Recommended)** – *graceful shutdown* (waits for tasks to finish).

It stops accepting new tasks. Let existing tasks finish

2. **shutdownNow()** – *immediate shutdown* (tries to stop all tasks)

**How Thread Pool Works**

1. A task is submitted via .execute() or .submit(). The task is placed in a **queue**.
2. A thread from the pool picks up the task and executes it. Once done, the thread returns to the pool for reuse.

**Risks Involved in Thread Pools**

1. Thread Exhaustion (Too Many Tasks, Not Enough Threads)
2. Task Exceptions Can Kill Threads Silently
3. Deadlock
4. Improper Shutdown = Resource Leaks
5. Too Few or Too Many Threads

**ThreadGroup**

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

But in **modern Java**, it's **rarely used** and considered **legacy**, because:

* It's not thread-safe, It doesn’t offer full control
* Better alternatives exist (ExecutorService, ForkJoinPool)

**ForkJoinPool**

ForkJoinPool is a **specialized thread pool** in Java designed for **parallelism**. It breaks a big task into **smaller subtasks**, runs them **in parallel**, and then **combines the results**.

Think of it like **divide-and-conquer** logic, executed concurrently. Ex:- Recursive File Search.

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| ForkJoinPool | The pool that manages and executes tasks |
| RecursiveTask<V> | For tasks that **return a result** |
| RecursiveAction | For tasks that **do not return a result** |

**Shutdown Hook**

A Shutdown Hook in Java is a special thread that executes when the JVM is shutting down.

Ex. Ctrl+C, System.Exit() etc

It is used to **clean up resources** (like closing files, database connections, saving logs, etc.) before the application stops.

Runtime.getRuntime().addShutdownHook(new Thread(() -> { }));

**Finalize() alternatives**: It is removed in Java 18.

|  |  |
| --- | --- |
| **Use Case** | **Preferred Approach** |
| File/DB/socket cleanup | try-with-resources |
| Custom cleanup logic | Implement AutoCloseable |
| Non-resource object cleanup. Ex. Cleaning up off-heap memory | Cleaner (Java 9+) |
| Complex GC-dependent cleanup | PhantomReference (advanced) |

**Cleaner:** java.lang.ref.Cleaner is a utility that:

* Registers an object for cleanup when it becomes unreachable (i.e., ready for garbage collection).
* Automatically runs the cleanup logic in the background after GC determines the object is unreachable.

**How It Works**

1. create a Cleaner instance.( **Cleaner cleaner = Cleaner.create();)**
2. Register an object with cleanup logic

Cleaner.Cleanable cleanable = cleaner.register(myObject, () -> { print("cleaning up obj")};

1. When the object becomes **unreachable**, the cleaner runs the cleanup logic

**Runtime class:**

The Runtime class in Java allows your program to interact with the **JVM** while it is running.

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| exec(String command) | Runs a system command |
| totalMemory() | Returns total memory JVM is using |
| freeMemory() | Returns free memory in JVM |
| maxMemory() | Returns max memory JVM can use |
| gc() | Requests garbage collection |
| addShutdownHook(Thread hook) | Registers a shutdown hook |
| exit(int status) | Terminates the JVM |

**Synchronization**

It is the capability to control the access of multiple threads to any shared resource. Synchronization is a better option where we want to allow only one thread to access the shared resource.

* **Synchronized Method:** The whole method is locked per object.
* **Synchronized Block:** Locks only the code block, not the entire method.
* **Static Synchronized Method:** Locks on the class, not the instance.
* **Synchronized on a Custom Lock:** You can use any object as a lock for synchronization, not just this

Object lock = new Object();

synchronized (lock) { //code}

**Reentrant Lock:** A reentrant lock is a lock that a thread can acquire multiple times without getting blocked. That means:

* If a thread already holds the lock, it can re-enter the locked code (just like synchronized).

You can create a **fair lock** that gives threads access in **first-come-first-served** order. Provide true in the constructor.

private final **ReentrantLock** lock = new ReentrantLock(//true); int count = 0;

public void increment() {

lock.lock(); **// acquire the lock**

try { //code } finally {

lock.unlock(); // **always release the lock**  }}

|  |  |
| --- | --- |
| **Method** | **Description** |
| lock() | Acquires the lock (waits if necessary) |
| unlock() | Releases the lock |
| tryLock() | Attempts to get the lock, returns true if successful |
| tryLock(timeout, unit) | Waits for specified time to get the lock |
| lockInterruptibly() | Waits for the lock, but can be interrupted |
| isHeldByCurrentThread() | Checks if current thread holds the lock |
| isLocked() | Checks if lock is held by any thread |

|  |  |  |
| --- | --- | --- |
| **Feature** | **synchronized (keyword)** | **ReentrantLock (class)** |
| **Basic Use** | synchronized block or method | Explicit lock() and unlock() |
| **Try to acquire without blocking** | ❌ No | ✅ tryLock() |
| **Timeout support** | ❌ No | ✅ tryLock(timeout, unit) |
| **Fairness (FIFO)** | ❌ No | ✅ Yes (via constructor new ReentrantLock(true)) |
| **Explicit unlock required** | ❌ No (automatic at block end) | ✅ Yes (must call unlock()) |
| **Condition support (wait/notify)** | ✅ Yes (via wait(), notify()) | ✅ Yes (newCondition()) |
| **Performance** | Better for simple use cases | More scalable under contention |
| **Deadlock risk** | Medium | Lower (if tryLock() or lockInterruptibly() used properly) |

# **Deadlock in Java**

Deadlock can occur in a situation when a thread is waiting for an object lock that is acquired by another thread, and a second thread is waiting for an object lock that is acquired by the first thread. Since both threads are waiting for each other to release the lock, the condition is called a deadlock.

**Conditions for Deadlock**  
 **Mutual Exclusion** – Only one thread can hold a lock at a time

 **Hold and Wait** – Thread holds one lock and waits for another

 **No Preemption** – Locks cannot be forcibly taken

 **Circular Wait** – Thread A waits for B, B waits for A

**How to Prevent Deadlock**

1. **Lock Ordering:** Always acquire locks in the **same order** across all threads.
2. **Try-Lock with Timeout (ReentrantLock)**
3. **Use Higher-Level Concurrency Utilities: ExecutorService, Semaphore, ConcurrentHashMap, Atomic class**
4. **Avoid Nested Locks**:
5. **Avoid Unnecessary Locks**:
6. **Using Thread Join**:

**Semaphore:** A **Semaphore** in Java is a **concurrency control tool** that limits the number of threads accessing a shared resource **at the same time**.

* Threads **acquire** a permit before accessing a resource.
* After use, threads **release** the permit.
* If no permits are available, the thread **waits (blocks)** until one is released.

Think of it as a **traffic signal** that allows only a limited number of cars (threads) to go at once.

Semaphore fairSemaphore = new Semaphore(3, true); // **FIFO order**

|  |  |
| --- | --- |
| **Method** | **Description** |
| acquire() | Requests a permit (waits if none available) |
| release() | Releases a permit |
| tryAcquire() | Attempts to acquire, returns immediately |
| availablePermits() | Shows how many permits are left |

* If a thread calls **acquire**() and no permit is available, it **waits (blocks)**.
* **release**() must be called to give the permit back.
* Semaphores are **not reentrant** – the same thread must call release() only once for every acquire()

**Atomic class: Atomic classes** in Java provide a way to **safely perform operations on variables in a multithreaded environment**, **without using synchronized or locks**.

**Advantages of Atomic Classes**

* Thread-safe without locking
* Faster than using synchronized
* Lock-free and non-blocking
* Efficient on multicore processors

**Limitations:** Only works for **simple atomic operations.** Not suitable for **compound actions** (like check-then-act)

|  |  |
| --- | --- |
| **Method** | **Description** |
| get() | Returns current value |
| set(int newValue) | Sets new value |
| getAndSet(int newValue) | Atomically sets and returns old value |
| incrementAndGet() | ++i (atomic) |
| getAndIncrement() | i++ (atomic) |
| addAndGet(int delta) | i += delta |
| compareAndSet(expected, update) | **CAS** – updates only if value is what you expect |

**Inter-thread Communication**

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

* **wait(), notify(), notifyAll()**

### **Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?**

It is because they are related to lock and object has a lock.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| The wait() method releases the lock. | The sleep() method doesn't release the lock. |
| It is a method of Object class | It is a method of Thread class |
| It is the non-static method | It is the static method |
| It should be notified by notify() or notifyAll() methods | After the specified amount of time, sleep is completed. |

**Interrupting a Thread:** In Java, interrupting a thread is a way to politely request a thread to stop what it’s doing, especially if it’s blocked or running a long task.

Every thread has an interrupted status flag. When thread.interrupt() is called, this flag is set to true.

* If the thread is sleeping, waiting, or blocked, it will throw an InterruptedException.
* If it's running normally, you have to check the flag manually using Thread.interrupted() or isInterrupted().

|  |  |
| --- | --- |
| **Method** | **What it does** |
| thread.interrupt() | Sets the thread’s interrupt flag |
| Thread.interrupted() | Static; checks and clears the flag |
| thread.isInterrupted() | Checks the flag without clearing it |

**Reentrant Monitor :** A thread can acquire the **same lock multiple times** without getting blocked or deadlocked.

**Advantage**: It eliminates the possibility of single thread deadlocking

**Reentrancy:** A thread that already holds the lock can acquire it again without blocking.

**Volatile Keyword:** The volatile keyword in Java is used to ensure visibility of changes to a variable across threads.

In multithreading, each thread may **cache variables** locally for performance.  
**means**: One thread might not **see** the latest value updated by another thread.

When a variable is declared as volatile:

* **Any write** to it is **immediately visible to all threads**.
* **Threads always read the latest value** directly from **main memory**, not their local cache.

|  |  |
| --- | --- |
| **Volatile Keyword** | **Synchronization Keyword** |
| Volatile keyword is a field modifier. | Synchronized keyword modifies code blocks and methods. |
| The thread cannot be blocked for waiting in case of volatile. | Threads can be blocked for waiting in case of synchronized. |
| It improves thread performance. | Synchronized methods degrade the thread performance. |
| It synchronizes the value of **one variable** at a time between thread memory and main memory. | It synchronizes the value of **all variables** between thread memory and main memory. |
| Volatile fields are not subject to compiler optimization. | Synchronize is subject to compiler optimization. |

**Java Optional Class orElseThrow()**

Optional<T> is a **container object** that may or may not contain a non-null value.  
It helps **avoid NullPointerException** and encourages **null-safe code**.

**Throw an exception** if the value inside the Optional is **absent (i.e., null)**.

|  |  |
| --- | --- |
| **Method** | **Behavior** |
| orElse(value) | Returns the value or fallback if absent |
| orElseGet(Supplier) | Lazy version of orElse |
| orElseThrow() | Throws NoSuchElementException if empty |
| orElseThrow(Supplier) | Throws your custom exception |

**Java Preconditions**

**Preconditions** in Java are **checks you perform before executing a method or block of code** to make sure that all necessary input conditions are valid. They help you:

* Catch programming errors early. Improve code robustness and readability.
* Avoid unexpected behavior (e.g., NullPointerException)

You can manually write them or **Google Guava's** Preconditions

|  |  |
| --- | --- |
| **Method** | **Use** |
| checkNotNull(obj) | Throws NullPointerException if obj is null |
| checkArgument(condition) | Throws IllegalArgumentException if false |
| checkState(condition) | Verifies object state (not method arg) |