**Java Multi-Threading**

**Process vs Thread**

* **Process:** A process runs independently of other processes and does not share memory with other processes. Each process has its own memory space including heap memory, call stack and method area. Inter-process communication is required to share data between two processes.
* **Thread:** A thread is a light weight unit of process. Threads share memory (i.e. heap memory and method area), but each thread has its own call stack. Also the context switching between threads is faster as compared to processes, as only the stack needs to be switched.

**Concurrency vs Parallelism**

* **Concurrency:** Concurrency is about managing multiple tasks at once, but not necessarily running them at the same instance. Tasks can start, run and complete in overlapping time periods. This can work on a single core processor with the help of context switching.
* **Parallelism:** Parallelism is about executing multiple tasks at the exact same time (simultaneously), typically on multiple core/processors.

**Java supports both Concurrency and Parallelism. Java enables concurrency by design and parallelism is a runtime operation.**

**The moment we create multiple threads in Java, we are working with concurrency. If the system hardware has multiple cores and if the JVM can utilizes them, we can achieve parallelism.**

**Concurrency Issues:**

1. **Visibility problem**

* A visibility problem occurs when one thread updates a shared variable, but other threads continue to see a stale/cached value because they are reading from their local caches (e.g., CPU cache or thread-local copy) instead of main memory.
* This happens because the Java Memory Model allows threads to cache values for efficiency, leading to inconsistent views of shared data.
* The visibility problem is typically resolved using volatile keyword or with synchronization as it establishes a happens-before relationship.

1. **Access problem**

* An access problem (or race condition) occurs when multiple threads simultaneously read, write, or modify shared data without proper synchronization, causing unpredictable interleaving of operations and leading to incorrect or inconsistent results.
* This happens because compound actions like read-modify-write are not atomic unless protected using synchronized, Lock, or atomic classes like AtomicInteger.

**Memory Visibility Guarantees of Synchronization in Java:**

In Java concurrency, when a thread enters a synchronized block or acquires a lock, it flushes its local caches, it is required to read the fresh values of shared variables from main memory, thereby discarding any cached or stale values. Similarly, when a thread exits a synchronized block or releases a lock, it must flush (write back) all changes made to shared variables to main memory so that they become visible to other threads. This behavior forms a ‘happens-before’ relationship as defined by the Java Memory Model, ensuring correct visibility and ordering of memory operations across threads.

**Different ways of creating a Thread in Java**

**1. Using Thread Class:** We can create a thread by extending the Thread class and overriding its run() method.

**Example:**

class MyThread extends Thread {

@Override

public void run() {

System.out.println("Thread running via Thread class");

}

}

public class Test {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Start the thread

}

}

**2. Using Runnable Interface:** We can define thread logic in a class that implements Runnable and pass it to a Thread object.

**Example:**

class MyRunnable implements Runnable {

@Override

public void run() {

System.out.println("Thread running via Runnable interface");

}

}

public class Test {

public static void main(String[] args) {

MyRunnable r = new MyRunnable();

Thread t1 = new Thread(r);

t1.start();

}

}

**3. Using Anonymous Inner Class:** We can create threads without defining a separate class.

**Example:**

public class Test {

public static void main(String[] args) {

// Runnable anonymous class

Thread t1 = new Thread(new Runnable() {

@Override

public void run() {

System.out.println("Thread via anonymous Runnable");

}

});

t1.start();

// Thread anonymous class

Thread t2 = new Thread() {

@Override

public void run() {

System.out.println("Thread via anonymous Thread");

}

};

t2.start();

}

}

**4. Using Lambda Functions:** With Java 8+, we can use lambdas with Runnable for cleaner code.

**Example:**

public class Test {

public static void main(String[] args) {

Thread t1 = new Thread(() -> {

System.out.println("Thread via lambda Runnable");

});

t1.start();

}

}

**5. Executor Service with Runnable Interface:** We can use the Executor framework for better thread management (thread pools, resource reuse).

**Example:**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test {

public static void main(String[] args) {

ExecutorService executor = Executors.newSingleThreadExecutor();

executor.submit(() -> {

System.out.println("Thread via ExecutorService");

});

executor.shutdown(); // Always shutdown the executor

}

}

**6. Executor Service with Callable Interface:** Unlike Runnable, Callable can return a result or throw exceptions.

**Example:**

import java.util.concurrent.\*;

public class Test {

public static void main(String[] args) throws Exception {

ExecutorService executor = Executors.newSingleThreadExecutor();

Future<String> future = executor.submit(() -> {

return "Thread via Callable with result";

});

System.out.println(future.get()); // Get the result

executor.shutdown();

}

}

**Runnable vs Callable**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Runnable** | **Callable** |
| **Package** | java.lang | java.util.concurrent |
| **Return type** | No return value (void run()) | Returns a result (V call()) |
| **Exception Handling** | Cannot throw checked exceptions (only unchecked/runtime exceptions) | Can throw checked exceptions (e.g., IOException) |
| **Method to implement** | public void run() | public V call() throws Exception |

**When would you prefer using ExecutorService over traditional thread creation (like using Thread or Runnable)?**

1. Manage a pool of threads efficiently : Instead of creating a new thread for each task (which is expensive in terms of system resources), ExecutorService allows reuse of threads from a pool, reducing overhead and improving performance.
2. Better control over task execution : ExecutorService provides methods like submit(), invokeAll(), and invokeAny() that give more flexibility compared to simply starting a thread.
3. Handle a large number of concurrent tasks : With ExecutorService, we can easily queue up tasks and let the executor manage scheduling, thread reuse, and resource management.
4. Get results or handle exceptions from tasks : Unlike Runnable, ExecutorService works with Callable, so we can submit tasks that return values (via Future) or throw checked exceptions, allowing for more complex task coordination.

**Java Thread Class**

**Constructors:**

* **Thread()**
* **Thread(String name)**
* **Thread(Runnable target)**
* **Thread(Runnable target, String name)**

**Methods:**

1. **public void run()**

* Provides the execution logic for a thread. Called by start().
* If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. Subclasses of Thread should override this method.

1. **public void start()**

* Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
* If you call run() directly instead of start(), it runs on the current thread, not a new one.
* Throws IllegalThreadStateException, if the thread was already started or if it has completed its execution or is dead.

1. **public final void setName​(String name), public final String getName()**

* Get or set the thread's name.

1. **public final void setPriority​(int newPriority), public final int getPriority()**

* Get or set the thread's priority.

1. **public final boolean isAlive(), public boolean isInterrupted()**

* isAlive() checks if this thread has started and is still running.
* isInterrupted() checks if the thread has been interrupted. The interrupted status of the thread is unaffected by this method.

1. **public final void setDaemon​(boolean on), public final boolean isDaemon()**

* Marks a thread as a daemon (runs in background). Must be set before start() is called.
* Checks if this thread is a daemon thread.

1. **public static void yield()**

* It is a way to implement cooperative multitasking.
* A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint.

1. **public static void sleep​(long millis), public static void sleep​(long millis, int nanos)**

* Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
* The thread does not lose ownership of any monitors.
* Throws InterruptedException

1. **public final void join(), public final void join​(long millis), public final void join​(long millis, int nanos)**

* Makes the calling thread wait until this thread finishes.
* The timed version of join() method implements a max waiting time.
* If the thread has already finished its execution, join() returns immediately.
* Throws InterruptedException

1. **public void interrupt()**

* Used to signal a thread to stop its execution. It does not immediately stop the threads execution, but sets up an interrupted flag.
* The executing thread can either catch the InterruptedException and handle it or can setup a Thread.currentThread().isInterrupted() check and act accordingly.

1. **public static boolean interrupted()**

* Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method.