# **What is Multithreading?**

Multithreading is a process of executing multiple threads concurrently within a single process. A thread is the smallest unit of execution within a process. It shares the same memory space with other threads of the same process.

# **What is Multitasking?**

Multitasking is the ability of an operating system to handle multiple tasks or processes concurrently.

**Process-based multitasking:** Involves multiple independent processes running simultaneously. Each process has its own memory space.

**Thread-based multitasking**: Involves multiple threads within a single process sharing the same memory space.

# What is the need for multithreading?

* **Improved performance:** By utilizing multiple threads, a program can perform multiple tasks simultaneously, improving overall efficiency.
* **Responsiveness**: Multithreading can prevent a program from freezing while performing long-running tasks.
* **Better utilization of system resources:** Threads can share resources, leading to efficient resource usage.

# Difference between process and thread?

| Feature | Process | Thread |
| --- | --- | --- |
| Definition | Independent program with its own memory space | Lightweight sub-process within a process, sharing memory |
| Resource Allocation | High | Low |
| Creation Overhead | High | Low |
| Communication | Slower (Inter-Process Communication) | Faster (Shared memory) |
| Termination | Affects entire program | Affects only the thread |

# How many ways to implement multithreading in Java?

There are two primary ways:

* Extending the Thread class: Create a new class that extends the Thread class and override the run() method.
* Implementing the Runnable interface: Create a class that implements the Runnable interface and override the run() method.

Thread class and Runnable interface

Thread class: Provides methods for thread creation, control, and management.

Runnable interface: Defines a single method, run(), which contains the code to be executed by the thread.

# What is the run method? Why overriding of run method is mandatory?

The run() method is the entry point for a thread. It contains the code that the thread will execute. Overriding the run() method is mandatory because it defines the thread's behavior.

# Explaining thread life cycle

A thread typically goes through the following states:

New: Thread is created but not yet started.

Runnable: Thread is ready to run but waiting for CPU time.

Running: Thread is currently executing.

Blocked: Thread is waiting for some event (e.g., I/O, monitor lock).

Terminated: Thread has finished execution.

Use of sleep() and join() methods

sleep(): Pauses the current thread for a specified time.

join(): Waits for a specified thread to terminate before continuing.

# Explaining thread priority

Thread priority is a numerical value assigned to a thread to indicate its importance relative to other threads. Higher priority threads have a higher chance of being scheduled for execution.

However, it's important to note that thread scheduling is ultimately determined by the operating system and JVM, and priority is just a hint to the scheduler.

Priority range:

Thread.MIN\_PRIORITY: Lowest priority (1)

Thread.NORM\_PRIORITY: Default priority (5)

Thread.MAX\_PRIORITY: Highest priority (10)

import java.lang.Thread;

public class ThreadPriorityExample {

public static void main(String[] args) {

Thread highPriorityThread = new Thread(() -> {

System.out.println("High priority thread started");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("High priority thread ended");

});

highPriorityThread.setPriority(Thread.MAX\_PRIORITY);

Thread lowPriorityThread = new Thread(() -> {

System.out.println("Low priority thread started");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Low priority thread ended");

});

lowPriorityThread.setPriority(Thread.MIN\_PRIORITY);

highPriorityThread.start();

lowPriorityThread.start();

}

}

1. Two threads are created: highPriorityThread and lowPriorityThread.
2. The priority of highPriorityThread is set to Thread.MAX\_PRIORITY.
3. The priority of lowPriorityThread is set to Thread.MIN\_PRIORITY.
4. Both threads are started.

# What is synchronization? What is the need for it?

Synchronization is a mechanism to control access to shared resources among multiple threads to prevent data corruption and race conditions. It ensures that only one thread can access a shared resource at a time.

Synchronization is a mechanism in Java to control access to shared resources among multiple threads. It prevents data corruption and race conditions that can occur when multiple threads attempt to modify shared data simultaneously.

public class Counter {

private int count = 0;

public void increment() {

count++;

}

}

If multiple threads call the increment() method concurrently, the result might be unexpected due to race conditions. Synchronization ensures that only one thread can access and modify the shared count variable at a time.

## How to achieve synchronization?

There are two primary ways to achieve synchronization in Java:

1. **Synchronized Methods:**

Declaring a method as synchronized ensures that only one thread can execute it at a time.

public class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

}

**2. Synchronized Blocks:**

Provides more granular control over synchronization.

A synchronized block acquires a lock on a specific object.

public class Counter {

private int count = 0;

public void increment() {

synchronized(this) {

count++;

}

}

}

Explaining Synchronized method and synchronized block

Synchronized method: The entire method is synchronized.

Synchronized block: Only a specific code block is synchronized.

public class Counter {

private int count = 0;

public void increment() {

count++;

}

public synchronized void incrementWithSync() {

count++;

}

}

public class ThreadExample extends Thread {

Counter counter;

public ThreadExample(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 1000; i++) {

counter.increment();

counter.incrementWithSync();

}

}

public static void main(String[] args) {

Counter counter = new Counter();

ThreadExample thread1 = new ThreadExample(counter);

ThreadExample thread2 = new ThreadExample(counter);

thread1.start();

thread2.start();

try {

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Count: " + counter.count);

}

}

```

In this example, two threads are created and both increment the `count` variable. Without synchronization, the final value of `count` might not be 2000 as expected due to race conditions. However, when using the synchronized method `incrementWithSync()`, the result is more likely to be accurate.

Remember that synchronization can introduce overhead, so it's important to use it judiciously.

# What is object-level locking and class-level locking?

* Object-level locking: Acquires a lock on the object instance.
* Class-level locking: Acquires a lock on the class itself.

# Explaining Deadlock with suitable example

Deadlock occurs when two or more threads are blocked, each waiting for the other to release a lock. It results in a standstill situation.

class A {

public synchronized void foo(B b) {

System.out.println("Thread A locked A");

b.last();

}

public synchronized void last() {

System.out.println("Thread A locked A again");

}

}

class B {

public synchronized void bar(A a) {

System.out.println("Thread B locked B");

a.last();

}

public synchronized void last() {

System.out.println("Thread B locked B again");

}

}

# Interthread Communication (Wait, notify, notifyAll)

* wait(): Releases the lock and waits until notified.
* notify(): Wakes up a single thread waiting on the object's monitor.
* notifyAll(): Wakes up all threads waiting on the object's monitor.

# Thread Pool

A thread pool is a collection of reusable threads that can be used to execute tasks. It helps in managing thread creation and improves performance by reusing threads.

Key benefits of using thread pools:

**Improved performance**: Reusing threads reduces the overhead of thread creation and destruction.

**Resource management**: Controls the number of threads, preventing resource exhaustion.

**Simplified management**: Provides a higher-level abstraction for managing threads.

import java.util.concurrent.\*;

public class ThreadPoolExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {

int taskNumber = i + 1;

executor.execute(() -> {

System.out.println("Task " + taskNumber + " is being executed by thread " + Thread.currentThread().getName());

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Task " + taskNumber + " is complete");

});

}

executor.shutdown();

}

}

1. An ExecutorService is created using Executors.newFixedThreadPool(5), which creates a thread pool with 5 threads.
2. Ten tasks are submitted to the executor service.
3. Each task prints a message indicating its start and completion.
4. The executor.shutdown() method is called to gracefully terminate the thread pool after all tasks are completed.

Important points:-

* The ExecutorService interface provides methods for submitting tasks to the pool and managing its lifecycle.
* The Executors class offers pre-configured thread pool implementations.
* Thread pools can be customized using the ThreadPoolExecutor class for more fine-grained control.
* Thread pool size: The number of threads in the pool should be carefully chosen based on the number of available cores and the nature of the tasks.
* Task queue: The thread pool uses a queue to hold tasks that are waiting to be executed.
* Rejected execution handler: A handler can be specified to handle tasks that cannot be executed due to the pool being full.
* By using thread pools effectively, you can optimize the performance of your applications and manage threads efficiently.