**Java Threading Assignment-3 Theory Questions and Answers**

**1.What is a thread in Java, and how does it differ from a process?**

In Java, a thread represents a single execution flow of control within a program. Threads allow concurrent execution of tasks, enabling programs to perform multiple operations simultaneously. Threads within the same process share memory and resources, which facilitates efficient communication and coordination.

Differences from a process:

* + **Thread**: Lightweight unit of execution, shares memory space with other threads of the same process.
  + **Process**: Independent execution unit, each process has its own memory space, resources, and state.

**2.How do you create a thread in Java using the Thread class and the Runnable interface?**

Thread Class:

In Java, the Thread class is a built-in class that represents a thread of execution.

To create a thread using the Thread class, you typically subclass it and override its run() method with the code you want to execute in the thread.

Example:

class MyThread extends Thread {

public void run() {

System.out.println("Thread using Thread class is running.");

}

}

MyThread thread = new MyThread();

thread.start(); // Starts the thread, which calls run() method internally

**3.Describe the lifecycle of a thread in Java.**

The lifecycle of a thread in Java consists of several states:

* + **New**: Created but not yet started.
  + **Runnable**: Ready to run, waiting for CPU time.
  + **Blocked/Waiting**: Waiting for a monitor lock or waiting for another thread's signal.
  + **Timed Waiting**: Waiting for a specified time period.
  + **Terminated**: Finished execution or terminated abruptly.

1. **What are the different states a thread can be in during its lifecycle? How does a thread transition from one state to another?**
   * **New**: Thread is instantiated.
   * **Runnable**: Thread transitions to runnable state using start() method.
   * **Blocked/Waiting**: Enters when waiting for I/O, synchronization, or timeout.
   * **Terminated**: Thread completes execution or is terminated forcefully.
2. **Explain the purpose of the start(), run(), and join() methods in the context of Java threads.**
   * **start()**: Initiates a new thread, calls its run() method.
   * **run()**: Contains code executed by the thread.
   * **join()**: Waits for the thread to terminate.
3. **What is the difference between calling the start() method and calling the run() method directly on a thread object?**
   * **start()**: Initiates a new thread of execution and calls its run() method in a new stack frame.
   * **run()**: Executes the thread's run() method in the current thread's stack, not creating a new thread.

**7.Why is thread synchronization necessary in a multi-threaded environment?**

Thread synchronization ensures that shared resources are accessed in a coordinated manner, preventing race conditions and maintaining data integrity.

**8.What is a race condition, and how can it be avoided using the synchronized keyword in Java?**

A race condition occurs when multiple threads access and modify shared data concurrently, leading to inconsistent results. Using the synchronized keyword ensures that only one thread can execute a synchronized block or method at a time, preventing simultaneous access and modifying shared data safely.

// Synchronized method example

public synchronized void increment() {

count++;

}

// Synchronized block example

synchronized (lockObject) {

// Critical section

}

**9.What mechanisms are available in Java for thread communication?**

Java provides methods like wait(), notify(), and notifyAll() for inter-thread communication using object monitors.

**10.Explain the use of wait(), notify(), and notifyAll() methods. Provide an example demonstrating their usage.**

* + **wait()**: Pauses the current thread until another thread invokes notify() or notifyAll() on the same object.
  + **notify()**: Wakes up a single thread that is waiting on the object's monitor.
  + **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

Example:

synchronized (sharedObject) {

while (!condition) {

sharedObject.wait(); // Releases the lock and waits

}

sharedObject.notify(); // Notifies one waiting thread

sharedObject.notifyAll(); // Notifies all waiting threads

}

**11.How can you set the priority of a thread in Java? What are the possible priority levels?**

Thread priorities in Java are set using the setPriority() method, which accepts integer values from Thread.MIN\_PRIORITY (1) to Thread.MAX\_PRIORITY (10), with Thread.NORM\_PRIORITY (5) being the default.

Thread thread = new Thread();

thread.setPriority(Thread.MAX\_PRIORITY); // Setting thread priority to maximum

**12.Does setting a thread's priority guarantee the order of execution? Explain why or why not.**

Setting a thread's priority does not guarantee the order of execution because thread scheduling is platform-dependent and varies between JVM implementations. Higher-priority threads have a better chance of being executed earlier, but it's not deterministic.

**13.What is the ExecutorService in Java, and how does it differ from managing threads manually?**

ExecutorService is a higher-level concurrency utility that manages a pool of threads, allowing efficient execution of tasks. It abstracts thread creation, management, and termination, providing features like task scheduling, thread pooling, and thread reuse. Compared to managing threads manually, ExecutorService offers better resource management, scalability, and simplifies concurrent programming.

**14.Explain the purpose of the Callable interface and Future class. Provide an example demonstrating their usage.**

* + **Callable Interface**: Represents a task that returns a result and can throw an exception. Used with ExecutorService to submit tasks that return a Future.
  + **Future Class**: Represents the result of an asynchronous computation. Allows checking if the computation is complete, cancelling the task, and retrieving the result.

Example:

import java.util.concurrent.\*;

public class CallableExample {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService executor = Executors.newFixedThreadPool(1);

Callable<Integer> task = () -> {

// Perform some computation

Thread.sleep(2000);

return 123;

};

Future<Integer> future = executor.submit(task);

System.out.println("Future result: " + future.get()); // Waits for task completion

executor.shutdown();

}

}

**15.What is a deadlock, and how can it be avoided in a multi-threaded program?**

Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. It can be avoided by:

* + Ensuring a single lock acquisition order.
  + Using tryLock() with timeout.
  + Releasing locks in a timely manner.

**16.Describe what is meant by a livelock and how it differs from a deadlock.**

Livelock occurs when threads continuously change their state in response to the actions of other threads, without making progress. Unlike deadlock, threads in livelock are not blocked but are unable to proceed due to repeated unsuccessful attempts to resolve a conflict.

**17.What are thread pools, and why are they useful in a multi-threaded application?**

Thread pools are a collection of pre-initialized threads that can be used to execute tasks concurrently. They improve performance by reducing thread creation overhead and managing resources effectively. Thread pools allow better control over the number of concurrent threads and facilitate reuse of threads, enhancing scalability and responsiveness of multi-threaded applications.

**18.What is thread safety, and why is it important?**

Thread safety ensures that shared data is accessed in a manner that guarantees correct behavior in a multi-threaded environment. It prevents race conditions, data corruption, and inconsistencies, ensuring predictable and reliable program execution.

**19.How can you ensure that a class is thread-safe? Provide examples of thread-safe and non-thread-safe classes.**

* + **Thread-safe**: Use synchronization (synchronized keyword), atomic classes (AtomicInteger), or thread-safe collections (ConcurrentHashMap).
  + **Non-thread-safe**: Classes without synchronization where shared data can be accessed concurrently without proper coordination.

Example of thread-safe class:

public class Counter {

private int count;

public synchronized void increment() {

count++;

}

public synchronized int getCount() {

return count;

}

}

Example of non-thread-safe class:

java

Copy code

public class UnsafeCounter {

private int count;

public void increment() {

count++;

}

public int getCount() {

return count;

}

}

**20.Modify a given program to use the Runnable interface instead of extending the Thread class.**

* + Runnable Interface:
  1. In Java, the Runnable interface is used to define a task that can be executed concurrently by a thread.
  2. Implementing Runnable allows for better separation of concerns by separating the task (runnable) from the execution (thread).
* Advantages of Runnable:
  1. Promotes better object-oriented design by separating the task logic from thread management.
  2. Allows for reusability of the task logic, as the same Runnable instance can be executed by multiple threads.
  3. Facilitates easier synchronization and resource management compared to extending the Thread class directly.

1. **Implement a simple program that demonstrates thread synchronization using the synchronized keyword.**

* Thread Synchronization:

1.In multi-threaded environments, thread synchronization ensures that only one thread at a time can execute a specific block of code or method associated with a shared resource.

2.Without synchronization, concurrent threads accessing shared resources can lead to data inconsistency and unexpected behavior.

* + synchronized Keyword:

1.Java provides the synchronized keyword to achieve thread synchronization.

2.It can be used with methods and code blocks to ensure that only one thread can access the synchronized resource at any given time.

1. **Create a multi-threaded program that uses ExecutorService to manage a pool of threads.**
   * ExecutorService:

1.ExecutorService in Java provides a high-level API for managing and controlling threads.

2.It abstracts away the complexity of thread creation, management, and lifecycle handling, making it easier to execute tasks asynchronously.

* + Thread Pool:

1.A thread pool is a collection of threads managed by ExecutorService that can be reused to execute multiple tasks.

2.It improves performance by reducing the overhead of thread creation and destruction.