**Exercise 1: Basic Thread Creation**

Question: Create a Java program that defines a class **MyThread** extending **Thread**. Override the **run()** method to print "Thread is running". Create an instance of **MyThread** and start it.

class MyThread extends Thread {

public void run() {

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

}

}

public class Main {

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start();

}

}

**Explanation:** This exercise demonstrates basic thread creation by extending the **Thread** class and overriding its **run()** method. When the **start()** method is called on the thread object, it invokes the **run()** method asynchronously.

**Exercise 2: Runnable Interface**

Question: Develop a Java program that implements the **Runnable** interface using a class named **MyRunnable**. In the **run()** method, print "Runnable is running". Create a thread using an instance of **MyRunnable** and start it.

.

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable is running");

}

}

public class Main {

public static void main(String[] args) {

MyRunnable runnable = new MyRunnable();

Thread thread = new Thread(runnable);

thread.start();

}

}

**Explanation:** This exercise uses the **Runnable** interface to create a thread. The **run()** method's code is placed in the **run()** method of the **MyRunnable** class, and then the **Thread** class is used to start the thread.

**Exercise 3: Thread Synchronization**

Question: Create a Java program that simulates a counter using a class **Counter**. The **Counter** class should have an **increment()** method that increments a private **count** variable. Use synchronization to prevent race conditions when two threads increment the counter concurrently.

**class Counter {**

**private int count = 0;**

**public synchronized void increment() {**

**count++;**

**}**

**public int getCount() {**

**return count;**

**}**

**}**

**public class Main {**

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

**Counter counter = new Counter();**

**Thread thread1 = new Thread(() -> {**

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

**counter.increment();**

**}**

**});**

**Thread thread2 = new Thread(() -> {**

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

**counter.increment();**

**}**

**});**

**thread1.start();**

**thread2.start();**

**thread1.join();**

**thread2.join();**

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

**}**

**}**

**Explanation: This exercise demonstrates thread synchronization using the synchronized keyword. The increment() method of the Counter class is synchronized to ensure that only one thread can modify the count variable at a time, preventing race conditions.**

**Exercise 4: Thread Priority**

Question: Write a Java program that creates two threads, **thread1** and **thread2**. Make **thread1** print even numbers and **thread2** print odd numbers. Set the priority of **thread1** to maximum and **thread2** to minimum.

**public class Main {**

**public static void main(String[] args) {**

**Thread thread1 = new Thread(() -> {**

**for (int i = 0; i < 5; i++) {**

**System.out.println("Thread 1: " + i);**

**}**

**});**

**Thread thread2 = new Thread(() -> {**

**for (int i = 0; i < 5; i++) {**

**System.out.println("Thread 2: " + i);**

**}**

**});**

**thread1.setPriority(Thread.MAX\_PRIORITY);**

**thread2.setPriority(Thread.MIN\_PRIORITY);**

**thread1.start();**

**thread2.start();**

**}**

**}**

**Explanation: This exercise sets thread priorities using the setPriority() method. Thread priorities range from Thread.MIN\_PRIORITY (1) to Thread.MAX\_PRIORITY (10). However, the actual behavior might vary depending on the underlying operating system's thread scheduling**

**Exercise 5: Thread Join**

Question: Develop a Java program that creates two threads, **thread1** and **thread2**. Ensure that **thread2** doesn't start until **thread1** has finished executing.

public class Main {

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

Thread thread1 = new Thread(() -> {

for (int i = 0; i < 5; i++) {

System.out.println("Thread 1: " + i);

}

});

Thread thread2 = new Thread(() -> {

for (int i = 0; i < 5; i++) {

System.out.println("Thread 2: " + i);

}

});

thread1.start();

thread1.join(); // Wait for thread1 to finish before starting thread2

thread2.start();

}

}

**Explanation:** This exercise uses the **join()** method to make sure that **thread2** doesn't start until **thread1** has completed its execution.

**Exercise 6: Thread Deadlock**

Question: Create a Java program that causes a deadlock scenario using two threads and two resources. Illustrate how a cyclic dependency of locks can lead to a deadlock.

public class Main {

public static void main(String[] args) {

final Object resource1 = "resource1";

final Object resource2 = "resource2";

Thread thread1 = new Thread(() -> {

synchronized (resource1) {

System.out.println("Thread 1: Holding resource 1...");

try { Thread.sleep(100); } catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for resource 2...");

synchronized (resource2) {

System.out.println("Thread 1: Acquired resource 2.");

}

}

});

Thread thread2 = new Thread(() -> {

synchronized (resource2) {

System.out.println("Thread 2: Holding resource 2...");

try { Thread.sleep(100); } catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for resource 1...");

synchronized (resource1) {

System.out.println("Thread 2: Acquired resource 1.");

}

}

});

thread1.start();

thread2.start();

}

}

**Explanation:** This exercise demonstrates a thread deadlock scenario where two threads are each holding a resource that the other thread wants, causing both threads to wait indefinitely. To avoid deadlocks, careful resource acquisition ordering should be followed.

**Exercise 7: Thread Interruption**

Question: Write a Java program that starts a thread and interrupts it after a short delay. Inside the thread, continuously print "Thread is running..." until it's interrupted, then print "Thread interrupted."

public class Main {

public static void main(String[] args) {

Thread thread = new Thread(() -> {

while (!Thread.currentThread().isInterrupted()) {

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

}

System.out.println("Thread interrupted.");

});

thread.start();

try { Thread.sleep(100); } catch (InterruptedException e) {}

thread.interrupt();

}

}

**Explanation:** This exercise demonstrates how to interrupt a thread using the **interrupt()** method and how to respond to thread interruption by checking the thread's interrupted status using **isInterrupted()**.

**Exercise 8: Thread Local**

Question: Develop a Java program that uses the **ThreadLocal** class to store and access an integer value separately for each thread. Print the thread-specific value in each thread's context.

public class Main {

private static ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);

public static void main(String[] args) {

Thread thread1 = new Thread(() -> {

threadLocal.set(42);

System.out.println("Thread 1: " + threadLocal.get());

});

Thread thread2 = new Thread(() -> {

threadLocal.set(99);

System.out.println("Thread 2: " + threadLocal.get());

});

thread1.start();

thread2.start();

}

}

**Explanation:** This exercise uses **ThreadLocal** to create thread-local variables. Each thread accesses its own instance of the variable, avoiding conflicts between threads.

**Exercise 9: Volatile Keyword**

Question: Write a Java program that declares a **volatile** boolean flag. Create two threads: one that continuously checks the flag and waits until it becomes **true**, and another that changes the flag to **true** after a short delay.

public class Main {

private static volatile boolean flag = false;

public static void main(String[] args) {

Thread thread1 = new Thread(() -> {

while (!flag) {

System.out.println("Thread 1 is waiting...");

}

System.out.println("Thread 1: Flag is now true.");

});

Thread thread2 = new Thread(() -> {

try { Thread.sleep(100); } catch (InterruptedException e) {}

System.out.println("Thread 2: Changing flag to true.");

flag = true;

});

thread1.start();

thread2.start();

}

}

**Explanation:** This exercise demonstrates the use of the **volatile** keyword to ensure that changes to a variable are immediately visible to all threads, avoiding issues with thread-caching.

**Exercise 10: Thread Pool**

Question: Develop a Java program that utilizes a thread pool to execute ten tasks concurrently. Each task should print a message indicating its task number and the thread that executed it. import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Main {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3);

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

int taskNumber = i;

executor.submit(() -> {

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

});

}

executor.shutdown();

}

}

**Explanation:** This exercise demonstrates using a thread pool with the **ExecutorService** interface. A fixed thread pool with three threads is created, and tasks are submitted for execution. The thread pool manages the threads' lifecycle, reusing them for multiple tasks.