1.A Creating and Managing Threads

class NumberPrinter extends Thread {

public void run() {

for (int i = 1; i <= 10; i++) {

System.out.println(i);

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class Main {

public static void main(String[] args) {

NumberPrinter thread1 = new NumberPrinter();

NumberPrinter thread2 = new NumberPrinter();

thread1.start();

thread2.start();

try {

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Both threads have finished.");

}

}

2.A States and Transactions

public class ThreadLifecycleDemo {

public static void main(String[] args) {

Thread thread = new Thread(() -> {

try {

System.out.println("Thread is in the NEW state.");

Thread.sleep(1000);

System.out.println("Thread is in the RUNNABLE state.");

synchronized (ThreadLifecycleDemo.class) {

ThreadLifecycleDemo.class.wait(1000);

}

System.out.println("Thread is in the WAITING state.");

// Simulate timed waiting

Thread.sleep(2000);

System.out.println("Thread is in the TIMED\_WAITING state.");

synchronized (ThreadLifecycleDemo.class) {

Thread anotherThread = new Thread(() -> {

synchronized (ThreadLifecycleDemo.class) {

// Simulate some work

System.out.println("Another thread is doing some work.");

}

});

anotherThread.start();

anotherThread.join();

}

System.out.println("Thread is in the BLOCKED state.");

Thread.sleep(1000);

System.out.println("Thread is in the TERMINATED state.");

} catch (InterruptedException e) {

e.printStackTrace();

}

});

thread.start();

}

}

2.A State and Transitions

public class ThreadLifecycleDemo {

public static void main(String[] args) {

Thread thread = new Thread(() -> {

try {

// NEW state

System.out.println("Thread is in NEW state");

// Simulate some work

Thread.sleep(1000);

// RUNNABLE state

System.out.println("Thread is in RUNNABLE state");

// Simulate waiting

synchronized (ThreadLifecycleDemo.class) {

ThreadLifecycleDemo.class.wait(1000);

}

System.out.println("Thread is in WAITING state");

Thread.sleep(1000);

System.out.println("Thread is in TIMED\_WAITING state");

synchronized (ThreadLifecycleDemo.class) {

ThreadLifecycleDemo.class.wait();

}

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

} catch (InterruptedException e) {

e.printStackTrace();

}

});

thread.start();

try {

thread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

3.A Synchronisation and Inter Thread Communication

import java.util.ArrayList;

import java.util.Random;

public class ProducerConsumerDemo {

private static final ArrayList<Integer> list = new ArrayList<>();

private static final Object lock = new Object();

public static void main(String[] args) {

Thread producerThread = new Thread(ProducerConsumerDemo::producer);

Thread consumerThread = new Thread(ProducerConsumerDemo::consumer);

producerThread.start();

consumerThread.start();

}

public static void producer() {

while (true) {

try {

Thread.sleep(500);

synchronized (lock) {

if (list.size() == 10) {

System.out.println("Queue full. Waiting to add...");

lock.wait();

} else {

int value = new Random().nextInt(100);

list.add(value);

System.out.println("Added element: " + value);

lock.notify();

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void consumer() {

while (true) {

try {

Thread.sleep(500);

synchronized (lock) {

if (list.isEmpty()) {

System.out.println("Queue is empty. Waiting to remove...");

lock.wait();

} else {

int removedValue = list.remove(0);

System.out.println("Removed element: " + removedValue);

lock.notify();

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

4.A Synchronised Blocks and Methods

class BankAccount {

private int balance;

public BankAccount(int initialBalance) {

this.balance = initialBalance;

}

public synchronized void deposit(String name, int amount) {

balance += amount;

System.out.println(name + " deposited $" + amount + ". New balance: $" + balance);

}

public synchronized void withdraw(String name, int amount) {

if (balance >= amount) {

balance -= amount;

System.out.println(name + " withdrew $" + amount + ". New balance: $" + balance);

} else {

System.out.println(name + " cannot withdraw $" + amount + ". Insufficient balance: $" + balance);

}

}

}

public class BankDemo {

public static void main(String[] args) {

BankAccount account = new BankAccount(100);

Thread thread1 = new Thread(() -> account.deposit("User1", 50));

Thread thread2 = new Thread(() -> account.withdraw("User2", 30));

thread1.start();

thread2.start();

}

}

5.A Thread pools and Concurrency Utilities

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPoolDemo {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(3);

for (int i = 1; i <= 5; i++) {

final int taskId = i;

executor.submit(() -> {

System.out.println("Task " + taskId + " executed by thread: " + Thread.currentThread().getName());

});

}

executor.shutdown();

}

}

6.A Executors Concurrent Collections and CompletableFuture

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class PrimeNumberCalculator {

public static void main(String[] args) {

int maxNumber = 1000; // Set your desired maximum number

ExecutorService executor = Executors.newFixedThreadPool(4); // Create a thread pool

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

return calculatePrimeNumbers(maxNumber);

});

try {

String result = primeResult.get();

System.out.println("Prime numbers: " + result);

} catch (Exception e) {

e.printStackTrace();

} finally {

executor.shutdown();

}

}

private static String calculatePrimeNumbers(int maxNumber) {

return "";

}

}

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.util.concurrent.CompletableFuture;

public class PrimeNumberFileWriter {

public static void main(String[] args) {

String primeNumbers = "2, 3, 5, 7, ...";

CompletableFuture<Void> writeToFile = CompletableFuture.runAsync(() -> {

try (BufferedWriter writer = new BufferedWriter(new FileWriter("prime\_numbers.txt"))) {

writer.write(primeNumbers);

} catch (Exception e) {

e.printStackTrace();

}

});

writeToFile.join();

System.out.println("Prime numbers written to file asynchronously.");

}

}

7.A Wriritng Thread safe Code Immutable objects

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {

private AtomicInteger counter = new AtomicInteger(0);

public int increment() {

return counter.incrementAndGet();

}

public int decrement() {

return counter.decrementAndGet();

}

public int getValue() {

return counter.get();

}

public static void main(String[] args) {

AtomicCounter counter = new AtomicCounter();

Thread incrementThread = new Thread(() -> {

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

counter.increment();

}

});

Thread decrementThread = new Thread(() -> {

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

counter.decrement();

}

});

incrementThread.start();

decrementThread.start();

try {

incrementThread.join();

decrementThread.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final counter value: " + counter.getValue());

}

}

public final class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;s

}

public int getY() {

return y;

}

public static void main(String[] args) {

Point point = new Point(10, 20);

System.out.println("Point coordinates: x=" + point.getX() + ", y=" + point.getY());

}

}