**Java Assignment**

|  |
| --- |
|  |

**Task 3: Synchronization and Inter-thread Communication**Implement a producer-consumer problem using wait() and notify() methods to handle the correct processing sequence between threads.

import java.util.LinkedList;

class Buffer {

private final int maxSize;

private final LinkedList<Integer> queue = new LinkedList<>();

public Buffer(int size) {

this.maxSize = size;

}

public synchronized void put(int value) throws InterruptedException {

while (queue.size() == maxSize) {

wait();

}

queue.add(value);

notifyAll();

}

public synchronized int take() throws InterruptedException {

while (queue.isEmpty()) {

wait();

}

int value = queue.removeFirst();

notifyAll();

return value;

}

}

class Consumer implements Runnable {

private final Buffer buffer;

public Consumer(Buffer buffer) {

this.buffer = buffer;

}

@Override

public void run() {

while (true) {

try {

int value = buffer.take();

System.out.println("Consumed: " + value);

Thread.sleep(500); // Simulate time taken to consume an item

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

break;

}

}

}

}

public class ProducerConsumerExample {

public static void main(String[] args) {

Buffer buffer = new Buffer(10);

Thread producerThread = new Thread(new Producer(buffer));

Thread consumerThread = new Thread(new Consumer(buffer));

producerThread.start();

consumerThread.start();

}

}  
 **Task 4: Synchronized Blocks and Methods**Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public synchronized void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.println(Thread.currentThread().getName() + " deposited " + amount + ", new balance: " + balance);

}

}

public synchronized void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ", new balance: " + balance);

} else {

System.out.println(Thread.currentThread().getName() + " failed to withdraw " + amount + ", insufficient funds.");

}

}

public double getBalance() {

return balance;

}

}

class DepositRunnable implements Runnable {

private final BankAccount account;

private final double amount;

public DepositRunnable(BankAccount account, double amount) {

this.account = account;

this.amount = amount;

}

@Override

public void run() {

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

account.deposit(amount);

try {

Thread.sleep(100); // Simulate time taken for each deposit

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

class WithdrawRunnable implements Runnable {

private final BankAccount account;

private final double amount;

public WithdrawRunnable(BankAccount account, double amount) {

this.account = account;

this.amount = amount;

}

@Override

public void run() {

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

account.withdraw(amount);

try {

Thread.sleep(100); // Simulate time taken for each withdrawal

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

}

public class BankAccountSimulation {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000); // Initial balance of 1000

Thread depositThread1 = new Thread(new DepositRunnable(account, 200), "DepositThread-1");

Thread depositThread2 = new Thread(new DepositRunnable(account, 300), "DepositThread-2");

Thread withdrawThread1 = new Thread(new WithdrawRunnable(account, 150), "WithdrawThread-1");

Thread withdrawThread2 = new Thread(new WithdrawRunnable(account, 250), "WithdrawThread-2");

depositThread1.start();

depositThread2.start();

withdrawThread1.start();

withdrawThread2.start();

}

}  
 **Task 5: Thread Pools and Concurrency Utilities**Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

import java.util.concurrent.Callable;

class ComplexTask implements Callable<String> {

private final int taskId;

public ComplexTask(int taskId) {

this.taskId = taskId;

}

@Override

public String call() throws Exception {

// Simulate a complex calculation or I/O operation

long duration = (long) (Math.random() \* 5000);

System.out.println("Task " + taskId + " is starting, will take " + duration + " milliseconds.");

Thread.sleep(duration);

return "Task " + taskId + " completed";

}

}

import java.util.concurrent.\*;

public class ThreadPoolExample {

public static void main(String[] args) {

int poolSize = 4; // Fixed size thread pool

int numTasks = 10; // Number of tasks to submit

ExecutorService executorService = Executors.newFixedThreadPool(poolSize);

Future<String>[] futures = new Future[numTasks];

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

ComplexTask task = new ComplexTask(i);

futures[i] = executorService.submit(task);

}

// Collect the results

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

try {

System.out.println(futures[i].get());

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

}

executorService.shutdown();

}

}

**Task 7: Writing Thread-Safe Code, Immutable Objects**  
Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

class Counter {

private int count;

public Counter(int initialCount) {

this.count = initialCount;

}

public synchronized void increment() {

count++;

System.out.println(Thread.currentThread().getName() + " incremented to " + count);

}

public synchronized void decrement() {

count--;

System.out.println(Thread.currentThread().getName() + " decremented to " + count);

}

public synchronized int getCount() {

return count;

}

}

final class ImmutableData {

private final String data;

public ImmutableData(String data) {

this.data = data;

}

public String getData() {

return data;

}

}

class CounterTask implements Runnable {

private final Counter counter;

public CounterTask(Counter counter) {

this.counter = counter;

}

@Override

public void run() {

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

counter.increment();

try {

Thread.sleep((int) (Math.random() \* 100));

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

counter.decrement();

}

}

} public class ThreadSafeDemo {

public static void main(String[] args) {

// Thread-safe Counter demonstration

Counter counter = new Counter(0);

Thread t1 = new Thread(new CounterTask(counter), "Thread-1");

Thread t2 = new Thread(new CounterTask(counter), "Thread-2");

t1.start();

t2.start()

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

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

// Immutable class demonstration

ImmutableData data = new ImmutableData("Shared Data");

System.out.println("ImmutableData: " + data.getData());

Thread t3 = new Thread(() -> {

System.out.println(Thread.currentThread().getName() + " reads ImmutableData: " + data.getData());

}, "Thread-3");

Thread t4 = new Thread(() -> {

System.out.println(Thread.currentThread().getName() + " reads ImmutableData: " + data.getData());

}, "Thread-4");

t3.start();

t4.start();

}

}

**Day 6:  
Task 1: Generics and Type Safety**  
Create a generic **Pair** class that holds two objects of different types, and write a method to return a reversed version of the pair.  
public class Pair<T, U> {

private T first;

private U second;

// Constructor

public Pair(T first, U second) {

this.first = first;

this.second = second;

}

// Getters

public T getFirst() {

return first;

}

public U getSecond() {

return second;

}

// Method to reverse the pair

public Pair<U, T> reverse() {

return new Pair<>(second, first);

}

// toString method for better visualization

@Override

public String toString() {

return "(" + first + ", " + second + ")";

}

public static void main(String[] args) {

Pair<String, Integer> pair = new Pair<>("Hello", 42);

System.out.println("Original pair: " + pair);

Pair<Integer, String> reversedPair = pair.reverse();

System.out.println("Reversed pair: " + reversedPair);

}

}  
**Task 2: Generic Classes and Methods**Implement a generic method that swaps the positions of two elements in an array, regardless of their type, and demonstrate its usage with different object types.

import java.util.Arrays;

public class ArrayUtils {

// Generic method to swap two elements in an array

public static <T> void swap(T[] array, int index1, int index2) {

if (array == null || index1 < 0 || index1 >= array.length || index2 < 0 || index2 >= array.length) {

throw new IllegalArgumentException("Invalid index or array");

}

T temp = array[index1];

array[index1] = array[index2];

array[index2] = temp;

}

public static void main(String[] args) {

// Example with Integer array

Integer[] intArray = {1, 2, 3, 4, 5};

System.out.println("Original Integer array: " + Arrays.toString(intArray));

swap(intArray, 1, 3);

System.out.println("Swapped Integer array: " + Arrays.toString(intArray));

// Example with String array

String[] strArray = {"a", "b", "c", "d"};

System.out.println("Original String array: " + Arrays.toString(strArray));

swap(strArray, 0, 2);

System.out.println("Swapped String array: " + Arrays.toString(strArray));

// Example with Double array

Double[] doubleArray = {1.1, 2.2, 3.3, 4.4};

System.out.println("Original Double array: " + Arrays.toString(doubleArray));

swap(doubleArray, 2, 3);

System.out.println("Swapped Double array: " + Arrays.toString(doubleArray));

}

}

**Task 3: Reflection API**Use reflection to inspect a class's methods, fields, and constructors, and modify the access level of a private field, setting its value during runtime

public class Person {

private String name;

private int age;

public Person() {

// Default constructor

}

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

private void printPersonInfo() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class ReflectionExample {

public static void main(String[] args) {

try {

// Get the Class object associated with the Person class

Class<?> personClass = Person.class;

// Inspecting constructors

System.out.println("Constructors:");

Constructor<?>[] constructors = personClass.getDeclaredConstructors();

for (Constructor<?> constructor : constructors) {

System.out.println(constructor);

}

// Inspecting fields

System.out.println("\nFields:");

Field[] fields = personClass.getDeclaredFields();

for (Field field : fields) {

System.out.println(field);

}

// Inspecting methods

System.out.println("\nMethods:");

Method[] methods = personClass.getDeclaredMethods();

for (Method method : methods) {

System.out.println(method);

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

import java.lang.reflect.Field;

public class ModifyPrivateField {

public static void main(String[] args) {

try {

// Create an instance of the Person class

Person person = new Person();

// Get the Class object associated with the Person class

Class<?> personClass = person.getClass();

// Get the private field 'name'

Field nameField = personClass.getDeclaredField("name");

// Make the private field accessible

nameField.setAccessible(true);

// Set the value of the private field 'name'

nameField.set(person, "John Doe");

// Verify the change

System.out.println("Name: " + person.getName());

// Repeat for the 'age' field

Field ageField = personClass.getDeclaredField("age");

ageField.setAccessible(true);

ageField.setInt(person, 30);

System.out.println("Age: " + person.getAge());

} catch (Exception e) {

e.printStackTrace();

}

}

}

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class ReflectionAndModificationExample {

public static void main(String[] args) {

try {

// Get the Class object associated with the Person class

Class<?> personClass = Person.class;

// Inspecting constructors

System.out.println("Constructors:");

Constructor<?>[] constructors = personClass.getDeclaredConstructors();

for (Constructor<?> constructor : constructors) {

System.out.println(constructor);

}

// Inspecting fields

System.out.println("\nFields:");

Field[] fields = personClass.getDeclaredFields();

for (Field field : fields) {

System.out.println(field);

}

// Inspecting methods

System.out.println("\nMethods:");

Method[] methods = personClass.getDeclaredMethods();

for (Method method : methods) {

System.out.println(method);

}

// Create an instance of the Person class

Person person = new Person();

// Modify the private field 'name'

Field nameField = personClass.getDeclaredField("name");

nameField.setAccessible(true);

nameField.set(person, "Jane Doe");

// Modify the private field 'age'

Field ageField = personClass.getDeclaredField("age");

ageField.setAccessible(true);

ageField.setInt(person, 25);

// Verify the changes

System.out.println("\nModified values:");

System.out.println("Name: " + person.getName());

System.out.println("Age: " + person.getAge());

} catch (Exception e) {

e.printStackTrace();

}

}

}

**Task 4: Lambda Expressions**Implement a Comparator for a Person class using a lambda expression, and sort a list of Person objects by their age.

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class LambdaComparatorExample {

public static void main(String[] args) {

// Create a list of Person objects

List<Person> people = new ArrayList<>();

people.add(new Person("Alice", 30));

people.add(new Person("Bob", 25));

people.add(new Person("Charlie", 35));

people.add(new Person("Diana", 20));

// Sort the list by age using a lambda expression

Collections.sort(people, (p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));

// Print the sorted list

people.forEach(System.out::println);

}

}