**Day-7 Assignment**

**Wrapper classes**

**1.**Check if character is a Digit.

public class DigitCheck {

public static void main(String[] args) {

char ch1 = '5';

char ch2 = 'A';

System.out.println(ch1 + " is digit? " + Character.isDigit(ch1));

System.out.println(ch2 + " is digit? " + Character.isDigit(ch2));

}

}

2. Compare two Strings.

public class StringCompare {

public static void main(String[] args) {

String s1 = "Hello";

String s2 = "World";

String s3 = "Hello";

System.out.println(s1.equals(s2));

System.out.println(s1.equals(s3));

}

}

3. Convert using valueof method.

public class ValueOfDemo {

public static void main(String[] args) {

int num = 123;

String str = String.valueOf(num);

System.out.println(str);

}

}

4. Create Boolean Wrapper usage.

public class BooleanWrapperDemo {

public static void main(String[] args) {

Boolean bool1 = Boolean.valueOf(true);

Boolean bool2 = Boolean.valueOf("false");

System.out.println(bool1);

System.out.println(bool2);

}

}

5. Convert null to wrapper classes.  
public class NullToWrapper {

public static void main(String[] args) {

Integer num = null;

Double dbl = null;

Boolean bool = null;

System.out.println(num);

System.out.println(dbl);

System.out.println(bool);

}

}

**Pass by value and pass by reference**

1. Write a program where a method accepts an integer parameter and tries to change its value. Print the value before and after the method call.

public class PassByValueDemo {

public static void changeValue(int num) {

num = 50;

}

public static void main(String[] args) {

int x = 10;

System.out.println("Before method call: " + x);

changeValue(x);

System.out.println("After method call: " + x);

}

}

1. Create a method that takes two integer values and swaps them. Show that the original values remain unchanged after the method call.

public class SwapDemo {

public static void swap(int a, int b) {

int temp = a;

a = b;

b = temp;

}

public static void main(String[] args) {

int x = 5, y = 10;

System.out.println("Before swap: x = " + x + ", y = " + y);

swap(x, y);

System.out.println("After swap: x = " + x + ", y = " + y);

}

}

1. Write a Java program to pass primitive data types to a method and observe whether changes inside the method affect the original variables.

public class PrimitivePassDemo {

public static void modifyValues(int num, double val) {

num += 10;

val \*= 2;

}

public static void main(String[] args) {

int number = 5;

double value = 3.5;

System.out.println("Before method call: number = " + number + ", value = " + value);

modifyValues(number, value);

System.out.println("After method call: number = " + number + ", value = " + value);

}

}

**Call by Reference (Using Objects)**

1. Create a class Box with a variable length. Write a method that modifies the value of length by passing the Box object. Show that the original object is modified.  
   class Box {

int length;

}

public class CallByReferenceDemo {

public static void modifyLength(Box b) {

b.length += 5;

}

public static void main(String[] args) {

Box box = new Box();

box.length = 10;

System.out.println("Before method call: length = " + box.length);

modifyLength(box);

System.out.println("After method call: length = " + box.length);

}

}

1. Write a Java program to pass an object to a method and modify its internal fields. Verify that the changes reflect outside the method.

class Person {

String name;

int age;

}

public class ModifyObjectDemo {

public static void updatePerson(Person p) {

p.name = "John";

p.age = 30;

}

public static void main(String[] args) {

Person person = new Person();

person.name = "Alice";

person.age = 25;

System.out.println("Before method call: " + person.name + ", " + person.age);

updatePerson(person);

System.out.println("After method call: " + person.name + ", " + person.age);

}

}

1. Create a class Student with name and marks. Write a method to update the marks of a student. Demonstrate the changes in the original object.

class Student {

String name;

int marks;

}

public class UpdateMarksDemo {

public static void updateMarks(Student s, int newMarks) {

s.marks = newMarks;

}

public static void main(String[] args) {

Student student = new Student();

student.name = "Narasimha";

student.marks = 85;

System.out.println("Before update: " + student.name + " - " + student.marks);

updateMarks(student, 95);

System.out.println("After update: " + student.name + " - " + student.marks);

}

}

1. Create a program to show that Java is strictly "call by value" even when passing objects (object references are passed by value).

class Person {

String name;

}

public class CallByValueObjects {

public static void changeReference(Person p) {

p = new Person();

p.name = "New Name";

}

public static void main(String[] args) {

Person person = new Person();

person.name = "Original Name";

System.out.println("Before changeReference: " + person.name);

changeReference(person);

System.out.println("After changeReference: " + person.name);

}

}

1. Write a program where you assign a new object to a reference passed into a method. Show that the original reference does not change.  
   class Car {

String model;

}

public class ReferenceAssignmentDemo {

public static void reassign(Car c) {

c = new Car();

c.model = "Tesla";

}

public static void main(String[] args) {

Car car = new Car();

car.model = "BMW";

System.out.println("Before reassign: " + car.model);

reassign(car);

System.out.println("After reassign: " + car.model);

}

}

1. Explain the difference between passing primitive and non-primitive types to methods in Java with examples.  
   public class PrimitiveDemo {

public static void changeValue(int x) {

x = 50;

}

public static void main(String[] args) {

int num = 10;

System.out.println("Before: " + num);

changeValue(num);

System.out.println("After: " + num);

}

}

1. Can you simulate call by reference in Java using a wrapper class or array? Justify with a program.

USING WRAPPER CLASS  
class IntWrapper {

int value;

}

public class CallByReferenceSimulation {

public static void modify(IntWrapper num) {

num.value = 100; // modifies the same object

}

public static void main(String[] args) {

IntWrapper number = new IntWrapper();

number.value = 10;

System.out.println("Before: " + number.value);

modify(number);

System.out.println("After: " + number.value);

}

}

USING AN ARRAY

public class ArrayReferenceDemo {

public static void modifyArray(int[] arr) {

arr[0] = 999; // modifies original array

}

public static void main(String[] args) {

int[] numbers = {1, 2, 3};

System.out.println("Before: " + numbers[0]);

modifyArray(numbers);

System.out.println("After: " + numbers[0]);

}

}

**MultiThreading**

1. Write a program to create a thread by extending the Thread class and print numbers from 1 to 5.

class MyThread extends Thread {

public void run() {

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

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

try {

Thread.sleep(500);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

public class ThreadExample1 {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

}

}

1. Create a thread by implementing the Runnable interface that prints the current thread name.  
   class MyRunnable implements Runnable {

public void run() {

System.out.println("Current Thread: " + Thread.currentThread().getName());

}

}

public class ThreadExample2 {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable());

t1.start();

}

}

1. Write a program to create two threads, each printing a different message 5 times.

class ThreadOne extends Thread {

public void run() {

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

System.out.println("Thread One Message");

}

}

}

class ThreadTwo extends Thread {

public void run() {

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

System.out.println("Thread Two Message");

}

}

}

public class TwoThreadsDemo {

public static void main(String[] args) {

new ThreadOne().start();

new ThreadTwo().start();

}

}

1. Demonstrate the use of Thread.sleep() by pausing execution between numbers from 1 to 3.

public class SleepDemo extends Thread {

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(i);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

public static void main(String[] args) {

new SleepDemo().start();

}

}

1. Create a thread and use Thread.yield() to pause and give chance to another thread.

public class YieldDemo extends Thread {

public void run() {

for (int i = 1; i <= 3; i++) {

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

Thread.yield();

}

}

public static void main(String[] args) {

YieldDemo t1 = new YieldDemo();

YieldDemo t2 = new YieldDemo();

t1.start();

t2.start();

}

}

1. Implement a program where two threads print even and odd numbers respectively.

class EvenThread extends Thread {

public void run() {

for (int i = 2; i <= 10; i += 2) {

System.out.println("Even: " + i);

}

}

}

class OddThread extends Thread {

public void run() {

for (int i = 1; i <= 9; i += 2) {

System.out.println("Odd: " + i);

}

}

}

public class EvenOddThreads {

public static void main(String[] args) {

new EvenThread().start();

new OddThread().start();

}

}

1. Create a program that starts three threads and sets different priorities for them.

class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

public void run() {

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

System.out.println(getName() + " - " + i);

}

}

}

public class ThreadPriorityExample {

public static void main(String[] args) {

MyThread t1 = new MyThread("Low Priority Thread");

MyThread t2 = new MyThread("Medium Priority Thread");

MyThread t3 = new MyThread("High Priority Thread");

t1.setPriority(Thread.MIN\_PRIORITY);

t2.setPriority(Thread.NORM\_PRIORITY);

t3.setPriority(Thread.MAX\_PRIORITY);

t1.start();

t2.start();

t3.start();

}

}

1. Write a program to demonstrate Thread.join() – wait for a thread to finish before proceeding.  
   class MyThread extends Thread {

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(getName() + " - " + i);

}

}

}

public class ThreadJoinExample {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

try {

t1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Main thread resumes after t1 finishes.");

}

}

1. Show how to stop a thread using a boolean flag.

class MyThread extends Thread {

private boolean running = true;

public void run() {

int i = 1;

while (running) {

System.out.println("Count: " + i++);

try {

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

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

}

public void stopThread() {

running = false;

}

}

public class StopThreadFlag {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start();

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

t.stopThread();

}

}

1. Create a program with multiple threads that access a shared counter without synchronization. Show the race condition.

class Counter {

int count = 0;

public void increment() {

count++;

}

}

class MyThread extends Thread {

Counter counter;

MyThread(Counter counter) {

this.counter = counter;

}

public void run() {

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

counter.increment();

}

}

}

public class RaceConditionDemo {

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

Counter counter = new Counter();

MyThread t1 = new MyThread(counter);

MyThread t2 = new MyThread(counter);

t1.start();

t2.start();

t1.join();

t2.join();

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

}

}

1. Solve the above problem using synchronized keyword to prevent race condition.

class Counter {

int count = 0;

public synchronized void increment() {

count++;

}

}

class MyThread extends Thread {

Counter counter;

MyThread(Counter counter) {

this.counter = counter;

}

public void run() {

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

counter.increment();

}

}

}

public class SynchronizedDemo {

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

Counter counter = new Counter();

MyThread t1 = new MyThread(counter);

MyThread t2 = new MyThread(counter);

t1.start();

t2.start();

t1.join();

t2.join();

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

}

}

1. Write a Java program using synchronized block to ensure mutual exclusion.

class Counter {

int count = 0;

public void increment() {

// Only this block is synchronized

synchronized (this) {

count++;

}

}

}

class MyThread extends Thread {

Counter counter;

MyThread(Counter counter) {

this.counter = counter;

}

public void run() {

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

counter.increment();

}

}

}

public class SynchronizedBlockDemo {

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

Counter counter = new Counter();

MyThread t1 = new MyThread(counter);

MyThread t2 = new MyThread(counter);

t1.start();

t2.start();

t1.join();

t2.join();

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

}

}

1. Implement a BankAccount class accessed by multiple threads to deposit and withdraw money. Use synchronization.

class BankAccount {

private int balance;

BankAccount(int initialBalance) {

this.balance = initialBalance;

}

// Synchronized deposit method

public synchronized void deposit(int amount) {

balance += amount;

System.out.println(Thread.currentThread().getName() +

" deposited " + amount + " | Balance: " + balance);

}

// Synchronized withdraw method

public synchronized void withdraw(int amount) {

if (balance >= amount) {

balance -= amount;

System.out.println(Thread.currentThread().getName() +

" withdrew " + amount + " | Balance: " + balance);

} else {

System.out.println(Thread.currentThread().getName() +

" tried to withdraw " + amount + " but insufficient funds! | Balance: " + balance);

}

}

public int getBalance() {

return balance;

}

}

class DepositTask implements Runnable {

private BankAccount account;

DepositTask(BankAccount account) {

this.account = account;

}

public void run() {

for (int i = 0; i < 3; i++) {

account.deposit(100);

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

}

}

}

class WithdrawTask implements Runnable {

private BankAccount account;

WithdrawTask(BankAccount account) {

this.account = account;

}

public void run() {

for (int i = 0; i < 3; i++) {

account.withdraw(100);

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

}

}

}

public class BankAccountDemo {

public static void main(String[] args) {

BankAccount account = new BankAccount(500);

Thread t1 = new Thread(new DepositTask(account), "Depositor");

Thread t2 = new Thread(new WithdrawTask(account), "Withdrawer");

t1.start();

t2.start();

try {

t1.join();

t2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Final Balance: " + account.getBalance());

}

}

1. Create a Producer-Consumer problem using wait() and notify().  
   import java.util.LinkedList;

import java.util.Queue;

class Drop {

private Queue<Integer> buffer = new LinkedList<>();

private int capacity;

Drop(int capacity) {

this.capacity = capacity;

}

public synchronized void put(int value) throws InterruptedException {

while (buffer.size() == capacity) {

wait();

}

buffer.add(value);

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

notifyAll();

}

public synchronized int take() throws InterruptedException {

while (buffer.isEmpty()) {

wait();

}

int value = buffer.poll();

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

notifyAll();

return value;

}

}

class Producer implements Runnable {

private Drop drop;

Producer(Drop drop) {

this.drop = drop;

}

public void run() {

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

try {

drop.put(i);

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable {

private Drop drop;

Consumer(Drop drop) {

this.drop = drop;

}

public void run() {

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

try {

drop.take();

Thread.sleep(150);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ProducerConsumerDemo {

public static void main(String[] args) {

Drop drop = new Drop(2);

Thread producer = new Thread(new Producer(drop));

Thread consumer = new Thread(new Consumer(drop));

producer.start();

consumer.start();

}

}

1. Create a program where one thread prints A-Z and another prints 1-26 alternately.

class Printer {

private boolean letterTurn = true;

public synchronized void printLetter(char letter) throws InterruptedException {

while (!letterTurn) {

wait();

}

System.out.print(letter + " ");

letterTurn = false;

notify();

}

public synchronized void printNumber(int number) throws InterruptedException {

while (letterTurn) {

wait();

}

System.out.print(number + " ");

letterTurn = true;

notify();

}

}

class LetterThread extends Thread {

private Printer printer;

LetterThread(Printer printer) {

this.printer = printer;

}

public void run() {

try {

for (char c = 'A'; c <= 'Z'; c++) {

printer.printLetter(c);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

class NumberThread extends Thread {

private Printer printer;

NumberThread(Printer printer) {

this.printer = printer;

}

public void run() {

try {

for (int i = 1; i <= 26; i++) {

printer.printNumber(i);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class AlternatingPrint {

public static void main(String[] args) {

Printer printer = new Printer();

new LetterThread(printer).start();

new NumberThread(printer).start();

}

}