**Wrapper classes**

1. Check if character is a Digit
2. Compare two Strings
3. Convert using valueof method
4. Create Boolean Wrapper usage
5. Convert null to wrapper classes

Code:

package Practice\_assign;

public class WrapperDemo {

public static void main(String[] args) {

// 1. Check if character is a Digit

char ch = '5';

if (Character.isDigit(ch)) {

System.out.println(ch + " is a digit.");

} else {

System.out.println(ch + " is not a digit.");

}

// 2. Compare two Strings

String str1 = "Hello";

String str2 = "World";

int cmp = str1.compareTo(str2);

if (cmp == 0) {

System.out.println("Strings are equal.");

} else if (cmp < 0) {

System.out.println(str1 + " comes before " + str2);

} else {

System.out.println(str1 + " comes after " + str2);

}

// 3. Convert using valueOf method

int num = 100;

String numStr = String.valueOf(num);

System.out.println("Converted int to String: " + numStr);

// 4. Create Boolean Wrapper usage

Boolean boolObj = Boolean.valueOf("true");

if (boolObj) {

System.out.println("Boolean object is true");

}

// 5. Convert null to wrapper classes

try {

String nullStr = null;

Integer intObj = Integer.valueOf(nullStr); // This will throw NumberFormatException

System.out.println("Converted null to Integer: " + intObj);

} catch (NumberFormatException e) {

System.out.println("Cannot convert null String to Integer: " + e.getMessage());

}

}

}

Output:

1) 5 is a digit.

2) Hello comes before World

3) Converted int to String: 100

4) Boolean object is true

5) Cannot convert null String to Integer: null

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.

Code:

public class PassByValueDemo {

// Method that tries to change the value

public static void changeValue(int num) {

System.out.println("Inside method before change: " + num);

num = num + 10; // changing local copy

System.out.println("Inside method after change: " + num);

}

public static void main(String[] args) {

int original = 50;

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

changeValue(original);

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

}

}

Output:

Before method call: 50

Inside method before change: 50

Inside method after change: 60

After method call: 50

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

Code:

package Practice\_assign;

public class SwapDemo {

// Method that tries to swap values

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

System.out.println("Inside method before swap: a = " + a + ", b = " + b);

int temp = a;

a = b;

b = temp;

System.out.println("Inside method after swap: a = " + a + ", b = " + b);

}

public static void main(String[] args) {

int x = 10;

int y = 20;

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

swap(x, y);

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

}

}

Output:

Before method call: x = 10, y = 20

Inside method before swap: a = 10, b = 20

Inside method after swap: a = 20, b = 10

After method call: x = 10, y = 20

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

Code:

package Practice\_assign;

public class PrimitivePassDemo {

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

System.out.println("Inside method before change: num = " + num + ", val = " + val + ", ch = " + ch);

num = num + 5;

val = val \* 2;

ch = (char) (ch + 1);

System.out.println("Inside method after change: num = " + num + ", val = " + val + ", ch = " + ch);

}

public static void main(String[] args) {

int number = 10;

double value = 3.5;

char letter = 'A';

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

modifyValues(number, value, letter);

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

}

}

Output:

Before method call: num = 10, val = 3.5, ch = A

Inside method before change: num = 10, val = 3.5, ch = A

Inside method after change: num = 15, val = 7.0, ch = B

After method call: num = 10, val = 3.5, ch = A

**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.

Code:

package Practice\_assign;

class Box {

int length;

Box(int length) {

this.length = length;

}

}

public class BoxDemo {

// Method to modify Box length

public static void modifyBox(Box b) {

System.out.println("Inside method before change: length = " + b.length);

b.length = b.length + 10; // Modifies the same object

System.out.println("Inside method after change: length = " + b.length);

}

public static void main(String[] args) {

Box myBox = new Box(20);

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

modifyBox(myBox);

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

}

}

Output:

Before method call: length = 20

Inside method before change: length = 20

Inside method after change: length = 30

After method call: length = 30

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.

Code:

package Practice\_assign;

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class ObjectPassDemo {

// Method to modify the object's fields

public static void updatePerson(Person p) {

System.out.println("Inside method before change: name = " + p.name + ", age = " + p.age);

// Modify fields

p.name = "Updated " + p.name;

p.age += 5;

System.out.println("Inside method after change: name = " + p.name + ", age = " + p.age);

}

public static void main(String[] args) {

Person person = new Person("John", 25);

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

updatePerson(person);

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

}

}

Output:

Before method call: name = John, age = 25

Inside method before change: name = John, age = 25

Inside method after change: name = Updated John, age = 30

After method call: name = Updated John, age = 30

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.

Code:

package Practice\_assign;

class Student {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

}

public class StudentDemo {

// Method to update marks

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

System.out.println("Inside method before update: " + s.name + " - " + s.marks);

s.marks = newMarks; // Updating the marks

System.out.println("Inside method after update: " + s.name + " - " + s.marks);

}

public static void main(String[] args) {

Student stu = new Student("Alice", 85);

System.out.println("Before method call: " + stu.name + " - " + stu.marks);

updateMarks(stu, 95);

System.out.println("After method call: " + stu.name + " - " + stu.marks);

}

}

Output:

Before method call: Alice - 85

Inside method before update: Alice - 85

Inside method after update: Alice - 95

After method call: Alice - 95

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

Code:

package Practice\_assign;

class Demo {

int data;

Demo(int data) {

this.data = data;

}

}

public class CallByValueDemo {

// This method tries to reassign the reference

public static void changeReference(Demo obj) {

System.out.println("Inside method before change: data = " + obj.data);

// Modifying field (this will affect original object)

obj.data = obj.data + 10;

// Reassigning obj to a new object (this will NOT affect original reference in main)

obj = new Demo(999);

System.out.println("Inside method after reassignment: data = " + obj.data);

}

public static void main(String[] args) {

Demo d = new Demo(50);

System.out.println("Before method call: data = " + d.data);

changeReference(d);

System.out.println("After method call: data = " + d.data);

}

}

Code:

Before method call: data = 50

Inside method before change: data = 50

Inside method after reassignment: data = 999

After method call: data = 60

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.

Code:

package Practice\_assign;

class Car {

String model;

Car(String model) {

this.model = model;

}

}

public class ReferenceReassignDemo {

// Method tries to reassign the passed reference

public static void changeCar(Car c) {

System.out.println("Inside method before reassignment: model = " + c.model);

// Assigning a new object to the local reference

c = new Car("Tesla");

System.out.println("Inside method after reassignment: model = " + c.model);

}

public static void main(String[] args) {

Car myCar = new Car("Honda");

System.out.println("Before method call: model = " + myCar.model);

changeCar(myCar);

System.out.println("After method call: model = " + myCar.model);

}

}

Output:

Before method call: model = Honda

Inside method before reassignment: model = Honda

Inside method after reassignment: model = Tesla

After method call: model = Honda

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

Answer:

**Primitive types** (like int, double, char) are passed by **copy of value**.

* When you pass a primitive variable, the method gets its own copy.
* Changing it inside the method **does not change** the original variable outside.
* **Non-primitive types** (objects like String, arrays, or custom classes) are passed by **copy of reference**.
  + The method gets a copy of the reference (address) pointing to the same object.
  + If you change the object's data inside the method, the original object **will be changed**.
  + But if you make the reference point to a new object, the original reference **won’t change**.

Examples:

Primitive:

void change(int x) {

x = 10; // changes only local copy

}

int num = 5;

change(num);

// num is still 5 here

Non primitive:

class Box { int length; }

void change(Box b) {

b.length = 10; // changes the actual object

}

Box box = new Box();

box.length = 5;

change(box);

// box.length is now 10 here

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

Code:

package Practice\_assign;

class IntWrapper {

int value;

IntWrapper(int value) {

this.value = value;

}

}

public class CallByReferenceDemo {

public static void modify(IntWrapper num) {

num.value = 100; // changes the original object's field

}

public static void main(String[] args) {

IntWrapper myNum = new IntWrapper(50);

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

modify(myNum);

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

}

}

Output:

Before modify: 50

After modify: 100

**MultiThreading**

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

Code:

package Practice;

class member extends Thread{

public void run()

{

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

{

System.***out***.println(i);

try {

Thread.*sleep*(500);

}

catch(InterruptedException e) {

}

}

}

}

public class Assign\_1 {

public static void main(String[] args) {

member t1=new member();

t1.start();

}

}

Output:

1

2

3

4

5

2 Create a thread by implementing the Runnable interface that prints the current thread name.

Code:

package Practice;

class MyRun implements Runnable{

public void run()

{

System.***out***.println("current thread name: "+Thread.*currentThread*().getName()); ;

}

}

public class Assign\_2 {

public static void main(String[] args) {

MyRun t1=new MyRun();

Thread t=new Thread(t1);

t.start();

}

}

Output:

current thread name: Thread-0

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

Code:

package Practice;

class messages implements Runnable{

public void run()

{

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

{

System.***out***.println("Hello");

try {

Thread.*sleep*(200);

}

catch(InterruptedException e)

{

}

}

}

}

class greets implements Runnable{

public void run()

{

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

{

System.***out***.println("Im Bharadwaj");

try {

Thread.*sleep*(500);

}

catch(InterruptedException e)

{

}

}

}

}

public class Assign\_3 {

public static void main(String[] args) {

Runnable r=new messages();

Runnable s=new greets();

Thread t1=new Thread(r);

Thread t2=new Thread(s);

t1.start();

t2.start();

}

}

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

package Practice;

class Ab implements Runnable{

public void run()

{

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

{

System.***out***.println(i);

try {

Thread.*sleep*(300);

}

catch(InterruptedException e)

{

}

}

}

}

public class Assign\_4 {

public static void main(String[] args) {

Runnable r=new Ab();

Thread t=new Thread(r);

t.start();

}

}

Output:

1

2

3

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

Code:

package Practice;

class sms implements Runnable{

public void run()

{

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

{

System.***out***.println("Hello");

Thread.*yield*();

}

}

}

class gun implements Runnable{

public void run()

{

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

{

System.***out***.println("Im Bharadwaj");

}

}

}

public class Assign\_5 {

public static void main(String[] args) {

Runnable r=new sms();

Runnable s=new gun();

Thread t1=new Thread(r);

Thread t2=new Thread(s);

t1.start();

t2.start();

}

}

Output:

Hello

Hello

Hello

Hello

Hello

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

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

Code:

package Practice;

class EvenThread extends Thread {

public void run() {

for (int i = 0; i <= 10; i += 2) {

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

}

}

}

class OddThread extends Thread {

public void run() {

for (int i = 1; i <= 10; i += 2) {

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

}

}

}

public class EvenOdd {

public static void main(String[] args) {

EvenThread t1 = new EvenThread();

OddThread t2 = new OddThread();

t1.start();

t2.start();

}

}

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

Code:

package Practice;

class Ab implements Runnable{

public void run()

{

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

{

System.***out***.println("Thread A");

//t1.setPriority(Thread.MIN\_PRIORITY);

}

}

}

class Ac implements Runnable{

public void run()

{

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

{

System.***out***.println("Thread B");

//t2.setPriority(Thread.NORM\_PRIORITY);

}

}

}

class Ad implements Runnable{

public void run()

{

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

{

System.***out***.println("Thread C");

//t3.setPriority(Thread.MAX\_PRIORITY);

}

}

}

public class Assign\_7 {

public static void main(String[] args) {

Runnable p=new Ab();

Runnable q=new Ac();

Runnable r=new Ad();

Thread t1=new Thread(p);

Thread t2=new Thread(q);

Thread t3=new Thread(r);

t1.setPriority(Thread.***MIN\_PRIORITY***);

t2.setPriority(Thread.***NORM\_PRIORITY***);

t3.setPriority(Thread.***MAX\_PRIORITY***);

t1.start();

t2.start();

t3.start();

}

}

Output:

Thread C

Thread C

Thread B

Thread B

Thread B

Thread B

Thread B

Thread B

1

Thread C

Thread C

Thread C

Thread C

2

3

8 Write a program to demonstrate Thread.join() – wait for a thread to finish before proceeding.

Code:

package Practice;

class Fa implements Runnable{

public void run()

{

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

{

System.***out***.println("Hello");

try {

Thread.*sleep*(200);

}

catch(InterruptedException e)

{

}

}

}

}

class Fb implements Runnable{

public void run()

{

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

{

System.***out***.println("Im Bharadwaj");

try {

Thread.*sleep*(500);

}

catch(InterruptedException e)

{

}

}

}

}

public class Assign\_8 {

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

Runnable r=new Fa();

Runnable s=new Fb();

Thread t1=new Thread(r);

Thread t2=new Thread(s);

System.***out***.println("Starting t1");

t1.start();

t1.join();

System.***out***.println("t1 finished, starting t2");

t2.start();

t2.join();

System.***out***.println("t2 finished");

}

}

Output:

Starting t1

Hello

Hello

Hello

Hello

Hello

t1 finished, starting t2

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

Im Bharadwaj

t2 finished

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

Code:

package Practice;

class Ga implements Runnable{

Boolean running=false;

static int *ret*=0;

public void run()

{

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

{

if(running==true)

{

*ret*=1;

break;

}

}

}

}

public class Assign\_9 {

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

Runnable r=new Ga();

Thread t=new Thread(r);

t.start();

t.join();

if(Ga.*ret*==1)

{

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

}

else

{

System.***out***.println("working");

}

}

}

Output:

working

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

Code:

package Practice;

class Counter{

static int *count*=0;

public static synchronized void increment()

{

*count*++;

}

}

class counterTask implements Runnable{

public void run()

{

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

{

Counter.*increment*();

}

}

}

public class Assign\_10 {

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

counterTask c=new counterTask();

counterTask d=new counterTask();

Thread t1=new Thread(c);

Thread t2=new Thread(d);

t1.start();

t2.start();

t1.join();

t2.join();

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

}

}

Output:

Final counter value: 2000

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

Code:

class Printer {

private int number = 1;

private int max;

Printer(int max) {

this.max = max;

}

public synchronized void printOdd() {

while (number <= max) {

if (number % 2 == 0) { // wait if number is even

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

} else {

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

number++;

notify();

}

}

}

public synchronized void printEven() {

while (number <= max) {

if (number % 2 == 1) { // wait if number is odd

try {

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

} else {

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

number++;

notify();

}

}

}

}

class OddThread extends Thread {

Printer printer;

OddThread(Printer p) {

this.printer = p;

}

public void run() {

printer.printOdd();

}

}

class EvenThread extends Thread {

Printer printer;

EvenThread(Printer p) {

this.printer = p;

}

public void run() {

printer.printEven();

}

}

public class EvenOddSync {

public static void main(String[] args) {

Printer printer = new Printer(10);

Thread t1 = new OddThread(printer);

Thread t2 = new EvenThread(printer);

t1.start();

t2.start();

}

}

Output:

Odd: 1

Even: 2

Odd: 3

Even: 4

...

Even: 10

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

Code:

class Counter {

int count = 0;

public void increment() {

// synchronized block to allow only one thread at a time

synchronized(this) {

count++;

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

}

}

}

public class SyncBlockDemo {

public static void main(String[] args) {

Counter counter = new Counter();

// Thread 1

Thread t1 = new Thread(() -> {

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

counter.increment();

try { Thread.sleep(50); } catch (InterruptedException e) {}

}

}, "Thread-1");

// Thread 2

Thread t2 = new Thread(() -> {

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

counter.increment();

try { Thread.sleep(50); } catch (InterruptedException e) {}

}

}, "Thread-2");

t1.start();

t2.start();

}

}

Output:

Thread-1 count: 1

Thread-2 count: 2

Thread-1 count: 3

Thread-2 count: 4

Thread-1 count: 5

Thread-2 count: 6

Thread-1 count: 7

Thread-2 count: 8

Thread-1 count: 9

Thread-2 count: 10

13 Implement a Bank Account class accessed by multiple threads to deposit and withdraw money. Use synchronization.

Code:

class BankAccount {

private int balance;

public BankAccount(int initialBalance) {

balance = initialBalance;

}

// Deposit method synchronized to prevent race conditions

public synchronized void deposit(int amount) {

balance += amount;

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

notifyAll(); // Notify waiting threads in case they are waiting to withdraw

}

// Withdraw method synchronized to ensure safe withdrawal

public synchronized void withdraw(int amount) {

// Wait if balance is insufficient

while (balance < amount) {

try {

System.out.println(Thread.currentThread().getName() + " waiting to withdraw " + amount + ", balance: " + balance);

wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

balance -= amount;

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

}

public synchronized int getBalance() {

return balance;

}

}

public class BankDemo {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000);

// Thread to deposit money

Thread depositor = new Thread(() -> {

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

account.deposit(500);

try { Thread.sleep(200); } catch (InterruptedException e) {}

}

}, "Depositor");

// Thread to withdraw money

Thread withdrawer = new Thread(() -> {

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

account.withdraw(700);

try { Thread.sleep(300); } catch (InterruptedException e) {}

}

}, "Withdrawer");

depositor.start();

withdrawer.start();

}

}

Output:

Withdrawer waiting to withdraw 700, balance: 1000

Withdrawer withdrew 700, balance: 300

Depositor deposited 500, balance: 800

Withdrawer waiting to withdraw 700, balance: 800

Depositor deposited 500, balance: 1300

Withdrawer withdrew 700, balance: 600

Depositor deposited 500, balance: 1100

Withdrawer withdrew 700, balance: 400

Depositor deposited 500, balance: 900

Withdrawer withdrew 700, balance: 200

Depositor deposited 500, balance: 700

Withdrawer withdrew 700, balance: 0

Depositor deposited 500, balance: 500

14 Create a Producer-Consumer problem using wait() and notify().

Code:

class Buffer {

private int data;

private boolean available = false;

public synchronized void produce(int value) {

while (available) {

try {

wait();

} catch (InterruptedException e) {}

}

data = value;

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

available = true;

notify();

}

public synchronized int consume() {

while (!available) {

try {

wait();

} catch (InterruptedException e) {}

}

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

available = false;

notify();

return data;

}

}

class Producer extends Thread {

Buffer buffer;

Producer(Buffer buffer) {

this.buffer = buffer;

}

public void run() {

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

buffer.produce(i);

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

}

}

}

class Consumer extends Thread {

Buffer buffer;

Consumer(Buffer buffer) {

this.buffer = buffer;

}

public void run() {

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

buffer.consume();

try { Thread.sleep(150); } catch (InterruptedException e) {}

}

}

}

public class ProducerConsumerDemo {

public static void main(String[] args) {

Buffer buffer = new Buffer();

Producer producer = new Producer(buffer);

Consumer consumer = new Consumer(buffer);

producer.start();

consumer.start();

}

}

Output:

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

Produced: 3

Consumed: 3

Produced: 4

Consumed: 4

Produced: 5

Consumed: 5

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

Code:

class Printer {

private boolean letterTurn = true; // true means letter thread's turn

public synchronized void printLetter(char c) {

while (!letterTurn) {

try {

wait();

} catch (InterruptedException e) {}

}

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

letterTurn = false;

notify();

}

public synchronized void printNumber(int n) {

while (letterTurn) {

try {

wait();

} catch (InterruptedException e) {}

}

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

letterTurn = true;

notify();

}

}

class LetterThread extends Thread {

Printer printer;

LetterThread(Printer printer) {

this.printer = printer;

}

public void run() {

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

printer.printLetter(c);

}

}

}

class NumberThread extends Thread {

Printer printer;

NumberThread(Printer printer) {

this.printer = printer;

}

public void run() {

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

printer.printNumber(i);

}

}

}

public class AlternatePrint {

public static void main(String[] args) {

Printer printer = new Printer();

Thread t1 = new LetterThread(printer);

Thread t2 = new NumberThread(printer);

t1.start();

t2.start();

}

}

Output:

A 1 B 2 C 3 D 4 E 5 F 6 G 7 H 8 I 9 J 10 K 11 L 12 M 13 N 14 O 15 P 16 Q 17 R 18 S 19 T 20 U 21 V 22 W 23 X 24 Y 25 Z 26

16 Write a program that demonstrates inter-thread communication using wait() and notifyAll().

Code:

class Message {

private String msg;

private boolean hasMessage = false;

public synchronized void putMessage(String m) {

while (hasMessage) {

try {

wait();

} catch (InterruptedException e) {}

}

msg = m;

hasMessage = true;

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

notifyAll();

}

public synchronized String takeMessage() {

while (!hasMessage) {

try {

wait();

} catch (InterruptedException e) {}

}

hasMessage = false;

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

notifyAll();

return msg;

}

}

class Producer extends Thread {

Message message;

Producer(Message message) {

this.message = message;

}

public void run() {

String[] messages = {"Hello", "Java", "Threads", "wait()", "notifyAll()"};

for (String m : messages) {

message.putMessage(m);

try { Thread.sleep(200); } catch (InterruptedException e) {}

}

}

}

class Consumer extends Thread {

Message message;

Consumer(Message message) {

this.message = message;

}

public void run() {

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

message.takeMessage();

try { Thread.sleep(300); } catch (InterruptedException e) {}

}

}

}

public class InterThreadCommunicationDemo {

public static void main(String[] args) {

Message message = new Message();

Producer producer = new Producer(message);

Consumer consumer = new Consumer(message);

producer.start();

consumer.start();

}

}

Output:

Produced: Hello

Consumed: Hello

Produced: Java

Consumed: Java

Produced: Threads

Consumed: Threads

Produced: wait()

Consumed: wait()

Produced: notifyAll()

Consumed: notifyAll()

17 Create a daemon thread that runs in background and prints time every second.

Code:

package Practice;

import java.time.LocalTime;

class TimePrinter implements Runnable {

public void run() {

while (true) {

System.out.println("Current time: " + LocalTime.now());

try {

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

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class DaemonThreadDemo {

public static void main(String[] args) {

Thread daemonThread = new Thread(new TimePrinter());

daemonThread.setDaemon(true); // Mark as daemon thread

daemonThread.start();

try {

Thread.sleep(5000); // Main thread sleeps for 5 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Main thread ending, daemon thread will stop too.");

}

}

Output:

Current time: 14:25:30.123

Current time: 14:25:31.123

Current time: 14:25:32.123

Current time: 14:25:33.123

Current time: 14:25:34.123

Main thread ending, daemon thread will stop too.

18 Demonstrate the use of Thread.isAlive() to check thread status.

Code:

package Practice;

class MyThread extends Thread {

public void run() {

try {

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

Thread.sleep(2000); // simulate work for 2 seconds

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

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class ThreadStatusDemo {

public static void main(String[] args) {

MyThread t = new MyThread();

System.out.println("Before start, isAlive: " + t.isAlive());

t.start();

System.out.println("Just after start, isAlive: " + t.isAlive());

try {

Thread.sleep(1000); // wait 1 second while thread runs

System.out.println("After 1 second, isAlive: " + t.isAlive());

t.join(); // wait for thread to finish

System.out.println("After join, isAlive: " + t.isAlive());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Output:

Before start, isAlive: false

Thread started.

Just after start, isAlive: true

After 1 second, isAlive: true

Thread finished.

After join, isAlive: false

19 Write a program to demonstrate thread group creation and management.

Code:

package Practice;

class MyThread extends Thread {

public MyThread(ThreadGroup group, String name) {

super(group, name);

}

public void run() {

System.out.println(getName() + " is running in group " + getThreadGroup().getName());

try {

Thread.sleep(1000); // simulate some work

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class ThreadGroupDemo {

public static void main(String[] args) {

// Create a thread group

ThreadGroup group1 = new ThreadGroup("Group1");

ThreadGroup group2 = new ThreadGroup("Group2");

// Create threads in group1

MyThread t1 = new MyThread(group1, "Thread-1");

MyThread t2 = new MyThread(group1, "Thread-2");

// Create threads in group2

MyThread t3 = new MyThread(group2, "Thread-3");

MyThread t4 = new MyThread(group2, "Thread-4");

// Start all threads

t1.start();

t2.start();

t3.start();

t4.start();

// Print active thread count in groups

System.out.println("Active threads in " + group1.getName() + ": " + group1.activeCount());

System.out.println("Active threads in " + group2.getName() + ": " + group2.activeCount());

// List all threads in group1

group1.list();

// List all threads in group2

group2.list();

}

}

Output:

Thread-1 is running in group Group1

Thread-2 is running in group Group1

Thread-3 is running in group Group2

Thread-4 is running in group Group2

Active threads in Group1: 2

Active threads in Group2: 2

java.lang.ThreadGroup[name=Group1,maxpri=10]

Thread[Thread-1,5,Group1]

Thread[Thread-2,5,Group1]

java.lang.ThreadGroup[name=Group2,maxpri=10]

Thread[Thread-3,5,Group2]

Thread[Thread-4,5,Group2]

20 Create a thread that performs a simple task (like multiplication) and returns result using Callable and Future.

Code:

package Practice;

import java.util.concurrent.Callable;

import java.util.concurrent.FutureTask;

import java.util.concurrent.ExecutionException;

class MultiplyTask implements Callable<Integer> {

private int a, b;

public MultiplyTask(int a, int b) {

this.a = a;

this.b = b;

}

public Integer call() {

return a \* b;

}

}

public class CallableFutureDemo {

public static void main(String[] args) {

MultiplyTask task = new MultiplyTask(5, 7);

FutureTask<Integer> future = new FutureTask<>(task);

Thread t = new Thread(future);

t.start();

try {

Integer result = future.get(); // waits for task to complete and gets result

System.out.println("Multiplication result: " + result);

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}

}

}

Output:

Multiplication result: 35