Wrapper classes

1. Check if character is a Digit

A.
```
package day7;
public class CheckDigit {
    public static void main(String[] args) {
        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

A.
```
package day7;
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "Hello";
        System.out.println("Using == : " + (str1 == str2));
        System.out.println("Using equals() : " + str1.equals(str2));
    }
}
```

3. Convert using valueof method

A.
```
package day7;
public class ValueOfConversion {
    public static void main(String[] args) {
```

```
        int num = 50;
        String str = String.valueOf(num);
        System.out.println("String: " + str);
    }
}
```

4. Create Boolean Wrapper usage

A.
```
package day7;
public class BooleanWrapper {
    public static void main(String[] args) {
        Boolean bool = Boolean.valueOf("true");
        System.out.println("Boolean value: " + bool);
    }
}
```

5. Convert null to wrapper classes

A.
```
package day7;
public class NullToWrapper {
    public static void main(String[] args) {
        Integer num = null;
        try {
            int value = num;
            System.out.println(value);
        } catch (NullPointerException e) {
            System.out.println("Cannot unbox null to primitive: " + e);
        }
    }
}
```

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.

A.
```
package day7;
public class PassByValueDemo {
    public static void main(String[] args) {
        int num = 10;
        System.out.println("Before: " + num);
        changeValue(num);
        System.out.println("After: " + num);
    }
    static void changeValue(int x) {
        x = 20;
    }
}
```

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

A.
```
package day7;
public class SwapIntegers {
    public static void main(String[] args) {
        int a = 5, b = 10;
        System.out.println("Before swap: a=" + a + ", b=" + b);
        swap(a, b);
        System.out.println("After swap: a=" + a + ", b=" + b);
    }
    static void swap(int x, int y) {
        int temp = x;
        x = y;
```

```
        y = temp;
    }
}
```

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

A.
```
package day7;
public class PrimitivePassDemo {
    public static void main(String[] args) {
        int val = 100;
        modify(val);
        System.out.println("Original value: " + val);
    }
    static void modify(int x) {
        x = x + 10;
        System.out.println("Modified inside method: " + x);
    }
}
```

Call by Reference (Using Objects)

4. 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.

A.
```
package day7;
class Box {
    int length;
}
public class BoxReferenceDemo {
    public static void main(String[] args) {
        Box b = new Box();
```

```
        b.length = 10;
        changeLength(b);
        System.out.println("Updated length: " + b.length);
    }
    static void changeLength(Box b) {
        b.length = 20;
    }
}
```

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

A.
```
package day7;
class Item {
    String name = "OldItem";
}
public class ObjectFieldModify {
    public static void main(String[] args) {
        Item item = new Item();
        changeName(item);
        System.out.println("Updated name: " + item.name);
    }
    static void changeName(Item obj) {
        obj.name = "NewItem";
    }
}
```

6. 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.

A.
```
package day7;
```

```
class Student {
    String name;
    int marks;
}
public class StudentMarksUpdate {
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "John";
        s.marks = 70;
        updateMarks(s);
        System.out.println(s.name + " has updated marks: " + s.marks);
    }
    static void updateMarks(Student student) {
        student.marks = 90;
    }
}
```

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

A.
```
package day7;
class Demo {
    int value = 5;
}
public class ObjectAssignmentDemo {
    public static void main(String[] args) {
        Demo d = new Demo();
        assignNewObject(d);
        System.out.println("Value after method: " + d.value);
    }
```

```
    static void assignNewObject(Demo obj) {
        obj = new Demo();
        obj.value = 100;
    }
}
```

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

A.
```
package day7;
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}
public class ObjectAssignmentDemo {
    public static void main(String[] args) {
        Person person = new Person("Suma");
        System.out.println("Before method: " + person.name);
        reassignObject(person);
        System.out.println("After method: " + person.name);
    }
    static void reassignObject(Person p) {
        p = new Person("Bob");
        System.out.println("Inside method: " + p.name);
    }
}
```

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

A. package day7;

```
public class PrimitiveVsObjectDemo {
    static void modifyPrimitive(int x) {
        x = x + 5;
        System.out.println("Inside modifyPrimitive: " + x);
    }
    static void modifyArray(int[] arr) {
        arr[0] = arr[0] + 5;
        System.out.println("Inside modifyArray: " + arr[0]);
    }
    public static void main(String[] args) {
        int a = 10;
        System.out.println("Before primitive method: " + a);
        modifyPrimitive(a);
        System.out.println("After primitive method: " + a);
        int[] nums = {10};
        System.out.println("\nBefore array method: " + nums[0]);
        modifyArray(nums);
        System.out.println("After array method: " + nums[0]);
    }
}
```

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

A. package day7;

```
class IntWrapper {
    int value;
    IntWrapper(int value) {
        this.value = value;
    }
```

```
}
public class SimulateCallByReference {
    public static void main(String[] args) {
        IntWrapper wrapper = new IntWrapper(100);
        System.out.println("Before method: " + wrapper.value);
        modify(wrapper);
        System.out.println("After method: " + wrapper.value);
    }
    static void modify(IntWrapper ref) {
        ref.value = 200;
        System.out.println("Inside method: " + ref.value);
    }
}
```

MultiThreading

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

A.
```
package day7;
public class ThreadExtendDemo extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i);
        }
    }
    public static void main(String[] args) {
        ThreadExtendDemo t = new ThreadExtendDemo();
        t.start();
    }
}
```

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

A. package day7;

```
public class RunnableThreadDemo implements Runnable {
    public void run() {
        System.out.println("Current Thread: " +
Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        Thread t = new Thread(new RunnableThreadDemo());
        t.start();
    }
}
```

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

A. package day7;

```
class MessagePrinter implements Runnable {
    private String message;
    MessagePrinter(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(message);
        }
    }
}
public class TwoThreadsDemo {
    public static void main(String[] args) {
```

```
        Thread t1 = new Thread(new MessagePrinter("Hello from Thread 1"));
        Thread t2 = new Thread(new MessagePrinter("Hello from Thread 2"));
        t1.start();
        t2.start();
    }
}
```

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

A. package day7;

```
public class ThreadSleepDemo {
    public static void main(String[] args) {
        for (int i = 1; i <= 3; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000); // pause 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

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

A. package day7;

```
class YieldDemo extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(getName() + " - " + i);
            Thread.yield();
```

```
        }
    }
}
public class ThreadYieldDemo {
    public static void main(String[] args) {
        YieldDemo t1 = new YieldDemo();
        YieldDemo t2 = new YieldDemo();
        t1.start();
        t2.start();
    }
}
```

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

A.
```
package day7;
class EvenPrinter extends Thread {
    public void run() {
        for (int i = 2; i <= 10; i += 2) {
            System.out.println("Even: " + i);
        }
    }
}
class OddPrinter 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) {
        EvenPrinter even = new EvenPrinter();
        OddPrinter odd = new OddPrinter();
        even.start();
        odd.start();
    }
}
```

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

A.
```
package day7;
class PriorityThread extends Thread {
    public void run() {
        System.out.println(getName() + " with priority " + getPriority() + " is running.");
    }
}
public class ThreadPriorityDemo {
    public static void main(String[] args) {
        PriorityThread t1 = new PriorityThread();
        PriorityThread t2 = new PriorityThread();
        PriorityThread t3 = new PriorityThread();
        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");
        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.NORM_PRIORITY); // 5
        t3.setPriority(Thread.MAX_PRIORITY); // 10
        t1.start();
```

```
        t2.start();
        t3.start();
    }
}
```

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

A.
```
package day7;
class JoinThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + ": " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {}
        }
    }
}
public class ThreadJoinDemo {
    public static void main(String[] args) {
        JoinThread t1 = new JoinThread();
        JoinThread t2 = new JoinThread();
        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t1.start();
        try {
            t1.join(); // Wait for t1 to finish
        } catch (InterruptedException e) {}
        t2.start();
    }
```

```
}
```

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

A.

```
package day7;
class StoppableThread extends Thread {
    private volatile boolean running = true;
    public void run() {
        while (running) {
            System.out.println("Thread is running...");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {}
        }
        System.out.println("Thread stopped.");
    }
    public void stopRunning() {
        running = false;
    }
}
public class StopThreadDemo {
    public static void main(String[] args) throws InterruptedException {
        StoppableThread t = new StoppableThread();
        t.start();

        Thread.sleep(2000);
        t.stopRunning();
    }
}
```

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

A. package day7;

```
class Counter {
    int count = 0;
    void increment() {
        count++;
    }
}
class CounterThread extends Thread {
    Counter counter;
    CounterThread(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();
        CounterThread t1 = new CounterThread(counter);
        CounterThread t2 = new CounterThread(counter);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final count (expected 2000): " + counter.count);
    }
```

```
}
```

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

A.

```
package Multithreading;
class Counter {
    private int count=0;
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
class CounterThread extends Thread {
    private Counter counter;
    public CounterThread(Counter c) {
        this.counter=c;
    }
    public void run() {
        for(int i=0;i<1000;i++) {
            counter.increment();
        }
    }
}
public class RaceConditionSolved {
    public static void main(String[] args) throws InterruptedException {
        Counter counter=new Counter();
        CounterThread t1=new CounterThread(counter);
        CounterThread t2=new CounterThread(counter);
```

```
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final count is: "+counter.getCount());
    }
}
```

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

A.
```
package Multithreading;
class SyncBlockDemo {
    private int count=0;
    public void increment() {
        synchronized(this) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
class SyncBlockThread extends Thread {
    private SyncBlockDemo demo;
    public SyncBlockThread(SyncBlockDemo demo) {
        this.demo=demo;
    }
    public void run() {
        for(int i=0;i<1000;i++) {
            demo.increment();
```

```
        }
    }
}
public class SyncBlockDemoMain {
    public static void main(String[] args) throws InterruptedException {
        SyncBlockDemo demo=new SyncBlockDemo();
        SyncBlockThread t1=new SyncBlockThread(demo);
        SyncBlockThread t2=new SyncBlockThread(demo);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Count: "+demo.getCount());
    }
}
```

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

A.

```
package Multithreading;
class BankAccount {
    private int balance=0;
    public synchronized void deposit(int amount) {
        balance+=amount;
    }
    public synchronized void withdraw(int amount) {
        balance-=amount;
    }
    public int getBalance() {
        return balance;
    }
```

```
}
class DepositThread extends Thread {
    private BankAccount account;
    public DepositThread(BankAccount account) {
        this.account=account;
    }
    public void run() {
        for(int i=0;i<1000;i++) {
            account.deposit(100);
        }
    }
}
class WithdrawThread extends Thread {
    private BankAccount account;
    public WithdrawThread(BankAccount account) {
        this.account=account;
    }
    public void run() {
        for(int i=0;i<1000;i++) {
            account.withdraw(100);
        }
    }
}
public class BankAccountSync {
    public static void main(String[] args) throws InterruptedException {
        BankAccount account=new BankAccount();
        DepositThread d=new DepositThread(account);
        WithdrawThread w=new WithdrawThread(account);
        d.start();
```

```
        w.start();
        d.join();
        w.join();
        System.out.println("Final balance: "+account.getBalance());
    }
}
```

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

A.
```
package Multithreading;
import java.util.LinkedList;
import java.util.Queue;
class ProducerConsumer {
    private final Queue<Integer> queue=new LinkedList<>();
    private final int LIMIT=5;
    public void produce() throws InterruptedException {
        int value=0;
        while(true) {
            synchronized(this) {
                while(queue.size()==LIMIT) wait();
                queue.offer(value++);
                notify();
            }
        }
    }
    public void consume() throws InterruptedException {
        while(true) {
            synchronized(this) {
                while(queue.isEmpty()) wait();
                int val=queue.poll();
                System.out.println("Consumed "+val);
```

```
            notify();
        }
    }
}
public class ProducerConsumerDemo {
    public static void main(String[] args) {
        ProducerConsumer pc=new ProducerConsumer();
        Thread producer=new Thread(() -> {
            try {
                pc.produce();
            } catch(InterruptedException e) {}
        });
        Thread consumer=new Thread(() -> {
            try {
                pc.consume();
            } catch(InterruptedException e) {}
        });
        producer.start();
        consumer.start();
    }
}
```

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

A.
```
package Multithreading;
public class PrintAlternately {
    private static final Object lock=new Object();
    private static boolean printLetter=true;
```

```
public static void main(String[] args) {
    Thread t1=new Thread(() -> {
        for(char c='A';c<='Z';c++) {
            synchronized(lock) {
                while(!printLetter) {
                    try {lock.wait();} catch(InterruptedException e) {}
                }
                System.out.print(c+" ");
                printLetter=false;
                lock.notify();
            }
        }
    });
    Thread t2=new Thread(() -> {
        for(int i=1;i<=26;i++) {
            synchronized(lock) {
                while(printLetter) {
                    try {lock.wait();} catch(InterruptedException e) {}
                }
                System.out.print(i+" ");
                printLetter=true;
                lock.notify();
            }
        }
    });
```

```
        t1.start();
        t2.start();
    }
}
```

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

A.
```
package Multithreading;
public class WaitNotifyAllDemo {
    private static final Object lock=new Object();
    private static boolean flag=false;
    public static void main(String[] args) {
        Thread t1=new Thread(() -> {
            synchronized(lock) {
                try {
                    while(!flag) lock.wait();
                    System.out.println("Thread1 notified");
                } catch(InterruptedException e) {}
            }
        });
        Thread t2=new Thread(() -> {
            synchronized(lock) {
                flag=true;
                lock.notifyAll();
                System.out.println("Thread2 notified all");
            }
        });

        t1.start();
        try {Thread.sleep(100);} catch(InterruptedException e) {}
```

```
        t2.start();
    }
}
```

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

A.
```
package Multithreading;
import java.time.LocalTime;
public class DaemonThreadDemo {
    public static void main(String[] args) {
        Thread daemon=new Thread(() -> {
            while(true) {
                System.out.println(LocalTime.now());
                try {Thread.sleep(1000);} catch(InterruptedException e) {}
            }
        });
        daemon.setDaemon(true);
        daemon.start();
        try {Thread.sleep(5000);} catch(InterruptedException e) {}
        System.out.println("Main thread finished");
    }
}
```

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

A.
```
package Multithreading;
public class ThreadIsAliveDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t=new Thread(() -> {
            try {Thread.sleep(2000);} catch(InterruptedException e) {}
        });
        System.out.println("Before start: "+t.isAlive());
```

```
        t.start();
        System.out.println("After start: "+t.isAlive());
        t.join();
        System.out.println("After join: "+t.isAlive());
    }
}
```

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

A.
```
package Multithreading;
public class ThreadGroupDemo {
    public static void main(String[] args) {
        ThreadGroup group=new ThreadGroup("MyGroup");
        Thread t1=new Thread(group, () -> {
            System.out.println(Thread.currentThread().getName()+" running");
        }, "Thread1");
        Thread t2=new Thread(group, () -> {
            System.out.println(Thread.currentThread().getName()+" running");
        }, "Thread2");
        t1.start();
        t2.start();
        System.out.println("Active threads in group: "+group.activeCount());
    }
}
```

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

A.
```
package Multithreading;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
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) {
        ExecutorService executor=Executors.newSingleThreadExecutor();
        MultiplyTask task=new MultiplyTask(6,7);
        Future<Integer> future=executor.submit(task);
        try {
            Integer result=future.get();
            System.out.println("Multiplication result: "+result);
        } catch(Exception e) {
            e.printStackTrace();
        }
        executor.shutdown();
    }
}
```