□ Task 1: Thread by Extending Thread Class

Problem:

Create a class that extends `Thread` and prints “Hello from Thread!” 5 times with 1-second delay.

Solution:

```
class MyPrinterThread extends Thread {
    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println("Hello from Thread!");
            try {
                Thread.sleep(1000);  // 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        MyPrinterThread t = new MyPrinterThread();
        t.start();
    }
}
```

□ Task 2: Thread by Implementing Runnable

Problem:

Create a thread using `Runnable` interface and print numbers from 1 to 10.

Solution:

```
class RunnableCounter implements Runnable {
    public void run() {
        for(int i = 1; i <= 10; i++) {
            System.out.println("Count: " + i);
        }
    }

    public static void main(String[] args) {
        RunnableCounter task = new RunnableCounter();
        Thread thread = new Thread(task);
        thread.start();
    }
}
```

□ Task 3: Sleep and Join

Problem:

Use `sleep()` for delay and `join()` to make main thread wait.

Solution:

```
class CountThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Counting: " + i);
            try {
                Thread.sleep(1000);
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        CountThread t = new CountThread();
        t.start();
        t.join();  // main thread waits here
        System.out.println("Main thread done.");
    }
}
```

□ Task 4: isAlive() Demo

Solution:

```
public class ThreadAlive extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadAlive t = new ThreadAlive();

        System.out.println("Is Alive before start: " + t.isAlive());
        t.start();
        System.out.println("Is Alive after start: " + t.isAlive());
        t.join();
        System.out.println("Is Alive after join: " + t.isAlive());
    }
}
```

□ Task 5: Thread Priority

Solution:

```
class PriorityDemo extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Priority: " +
Thread.currentThread().getPriority());
    }

    public static void main(String[] args) {
        PriorityDemo t1 = new PriorityDemo();
        PriorityDemo t2 = new PriorityDemo();
        PriorityDemo t3 = new PriorityDemo();

        t1.setPriority(Thread.MIN_PRIORITY);   // 1
        t2.setPriority(Thread.NORM_PRIORITY);  // 5
        t3.setPriority(Thread.MAX_PRIORITY);   // 10

        t1.setName("Low");
        t2.setName("Medium");
        t3.setName("High");

        t1.start(); t2.start(); t3.start();
    }
}
```

□ Task 6: Counter Without Synchronization

Solution:

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}

public class NoSync {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) c.increment();
        });

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

```
        t1.join(); t2.join();

        System.out.println("Final Count (without sync): " + c.count);  //
likely < 2000
    }
}
```

□ Task 7: Counter With Synchronization

Solution:

```
class SafeCounter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}

public class WithSync {
    public static void main(String[] args) throws InterruptedException {
        SafeCounter c = new SafeCounter();

        Thread t1 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) c.increment();
        });

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Final Count (with sync): " + c.count);  // always
2000
    }
}
```

□ Task 8: Bank Account Withdrawal

Solution:

```
class BankAccount {
    private int balance = 1000;

    synchronized void withdraw(int amount, String user) {
        if (balance >= amount) {
            System.out.println(user + " is withdrawing $" + amount);
            balance -= amount;
            System.out.println("Remaining Balance: $" + balance);
        } else {
            System.out.println(user + " tried to withdraw but insufficient
funds.");
        }
    }
}

public class BankApp {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Runnable user1 = () -> account.withdraw(700, "User1");
        Runnable user2 = () -> account.withdraw(500, "User2");

        new Thread(user1).start();
        new Thread(user2).start();
    }
}
```

□ Task 9: Producer-Consumer using wait/notify

Solution:

```
class Buffer {
    private int data;
    private boolean available = false;

    synchronized void produce(int value) {
        while(available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        data = value;
        System.out.println("Produced: " + data);
        available = true;
        notify();
    }

    synchronized void consume() {
        while(!available) {
            try { wait(); } catch (InterruptedException e) {}
```

```
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify();
    }
}

public class ProducerConsumer {
    public static void main(String[] args) {
        Buffer b = new Buffer();

        Thread producer = new Thread(() -> {
            for(int i = 1; i <= 5; i++) b.produce(i);
        });

        Thread consumer = new Thread(() -> {
            for(int i = 1; i <= 5; i++) b.consume();
        });

        producer.start(); consumer.start();
    }
}
```

□ Task 10: Chat Simulation (Ping-Pong Communication)

Solution:

```
class Chat {
    boolean turn = true;

    synchronized void sayHi(String msg) {
        while(!turn) {
            try { wait(); } catch(Exception e) {}
        }
        System.out.println("Hi: " + msg);
        turn = false;
        notify();
    }

    synchronized void sayHello(String msg) {
        while(turn) {
            try { wait(); } catch(Exception e) {}
        }
        System.out.println("Hello: " + msg);
        turn = true;
        notify();
    }
}
```

```
public class ChatApp {
    public static void main(String[] args) {
        Chat chat = new Chat();

        new Thread(() -> {
            chat.sayHi("How are you?");
            chat.sayHi("What’s up?");
        }).start();

        new Thread(() -> {
            chat.sayHello("I’m good!");
            chat.sayHello("Working on Java.");
        }).start();
    }
}
```

□ Task 11: Thread Race Simulation

Solution:

```
class AnimalRace extends Thread {
    public AnimalRace(String name) {
        super(name);
    }

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(getName() + " running " + i);
            try {
                Thread.sleep((int)(Math.random() * 1000));
            } catch(Exception e) {}
        }
    }

    public static void main(String[] args) {
        new AnimalRace("Tortoise").start();
        new AnimalRace("Rabbit").start();
        new AnimalRace("Dog").start();
    }
}
```

□ Task 12: Countdown Timer

Solution:

```
public class CountdownTimer extends Thread {
    public void run() {
        for(int i = 10; i >= 1; i--) {
            System.out.println("Countdown: " + i);
            try { Thread.sleep(1000); } catch(Exception e) {}
        }
        System.out.println("Time’s up!");
    }

    public static void main(String[] args) {
        new CountdownTimer().start();
    }
}
```