Com2104 Advanced Programming

WK7 LECTURE: THREAD

Objectives

- Know what is a thread and how to create one thread
- Know what is multithreading and how to realize it
- Know what is concurrency and how to implement it
- Know what is synchronization and how to realize it.

Thread

Characteristics of threads

- A thread is an execution thread in a program.
- Multiple threads of execution can be run concurrently by an application running on the JVM.
- The priority of each thread varies. Higher priority threads are executed before lower priority threads.

What is a Thread in Java?

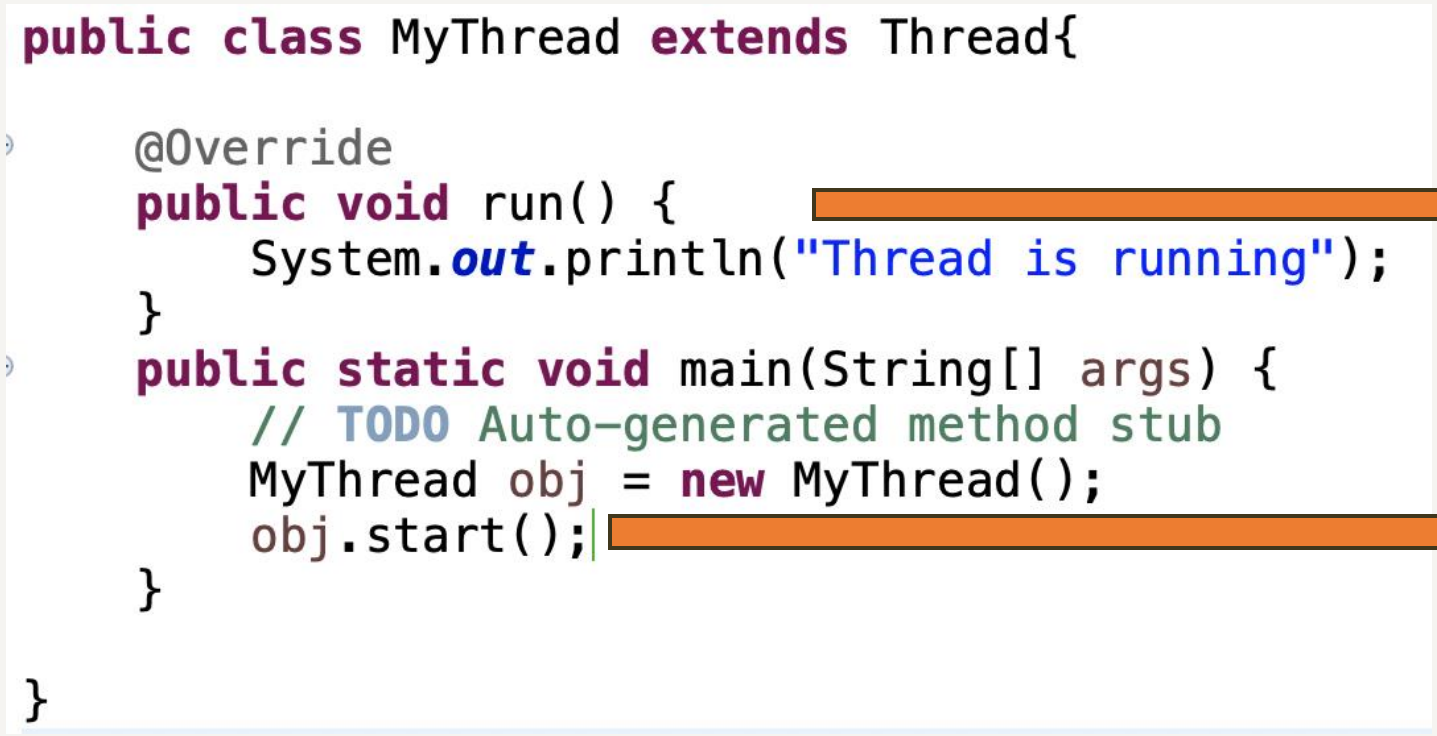
- A thread in Java is the direction or path that is taken while a program is being executed.
- Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM at the starting of the program's execution.
- When the main thread is provided, the main() method is invoked by the main thread.

Creating a Thread in Java

- A thread in Java can be created in the following two ways:
  - Extending java.lang.Thread class
    - Override run() method
  - Implementing Runnable interface
    - Override run() method

Extending java.lang.Thread class

In this case, a thread is created by a new class that extends the Thread class, creating an instance of that class. The run() method includes the functionality that is supposed to be implemented by the Thread.



```
public class MyThread extends Thread{

    @Override
    public void run() {
        System.out.println("Thread is running");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        MyThread obj = new MyThread();
        obj.start();
    }

}
```

We must have to override run() method when our class extends from Thread class.

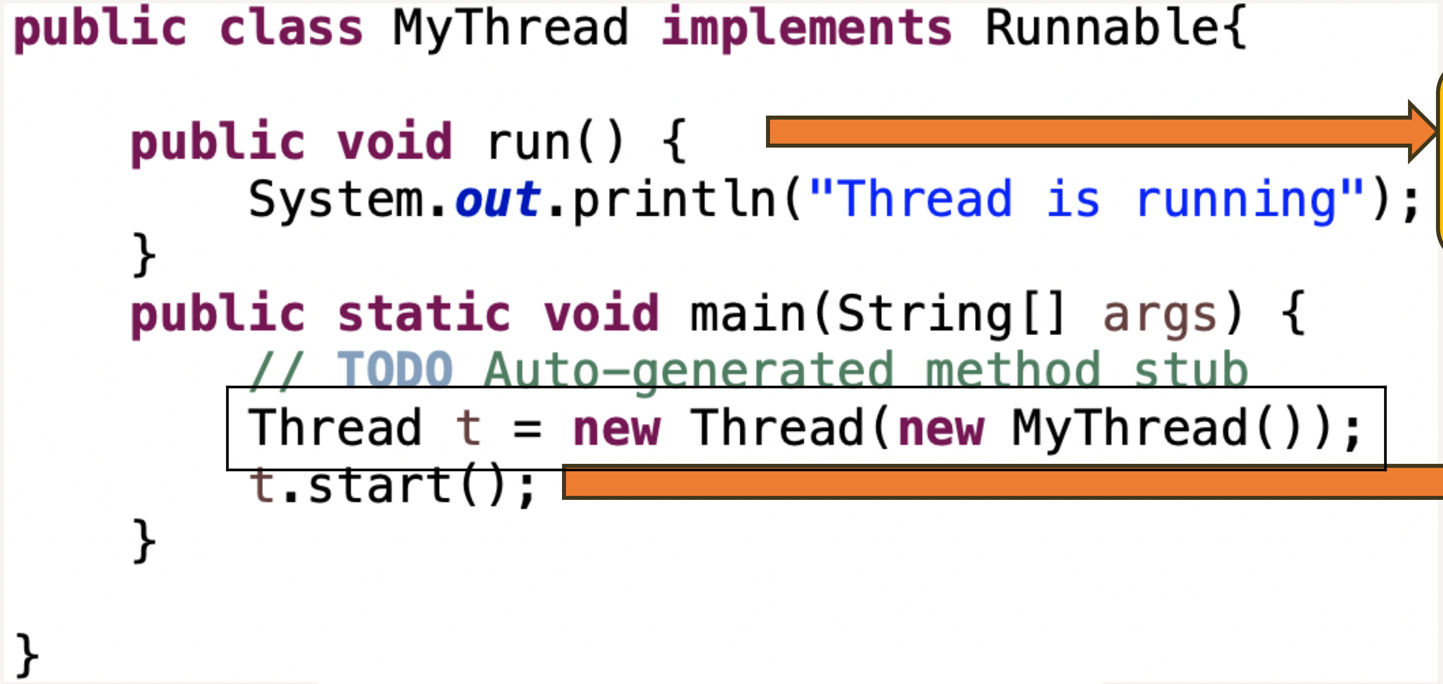
Here, start() is used to create a new thread and to make it runnable. The new thread begins inside the void run() method

Output:

```
Thread is running
```

Implementing Runnable interface

In this case, a class is created to implement the runnable interface and then the run() method

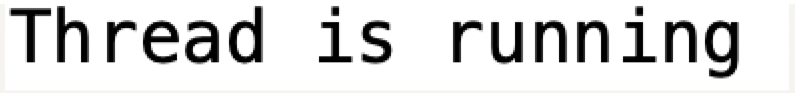


```java
public class MyThread implements Runnable{

    public void run() {
        System.out.println("Thread is running");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Thread t = new Thread(new MyThread());
        t.start();
    }

}
```

We must have to implement the run() method

The start() method is used to call the void run() method. When start() is called, a new stack is given to the thread, and run() is invoked to introduce a new thread in the program.

Output:

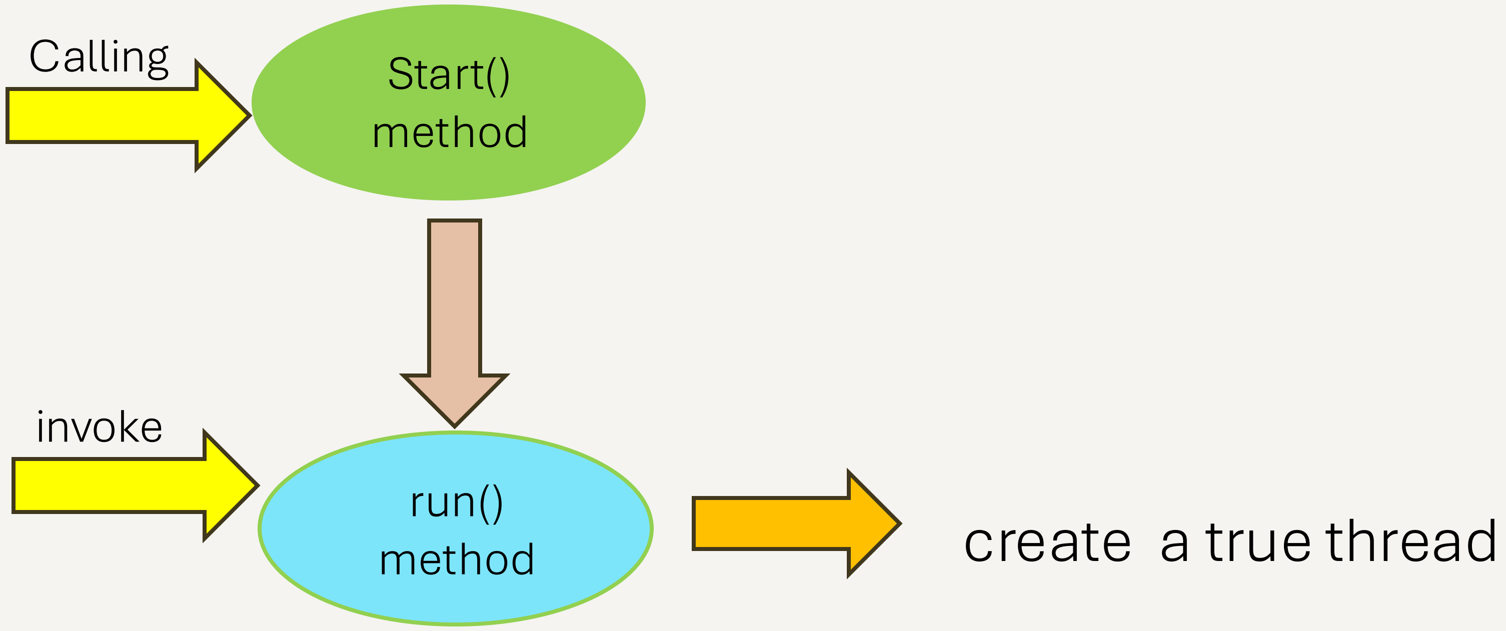


```
Thread is running
```

Be careful when creating a thread

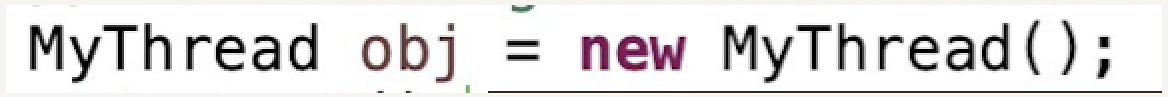
- When creating a class extending Thread class, we must have to override the run() method where you should write the functionality for the thread.
- When running the thread, create one instance of the class first, then call the start() method with the instance.

The logic of calling start() to run a thread



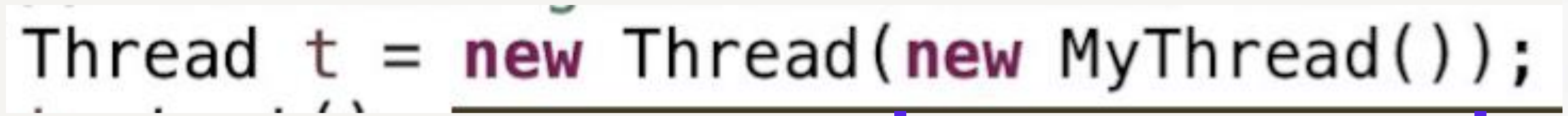
Difference of instantiation between using Thead and Runnable

Extending Thread class:



```
MyThread obj = new MyThread();
```

Implementing Runnable class:

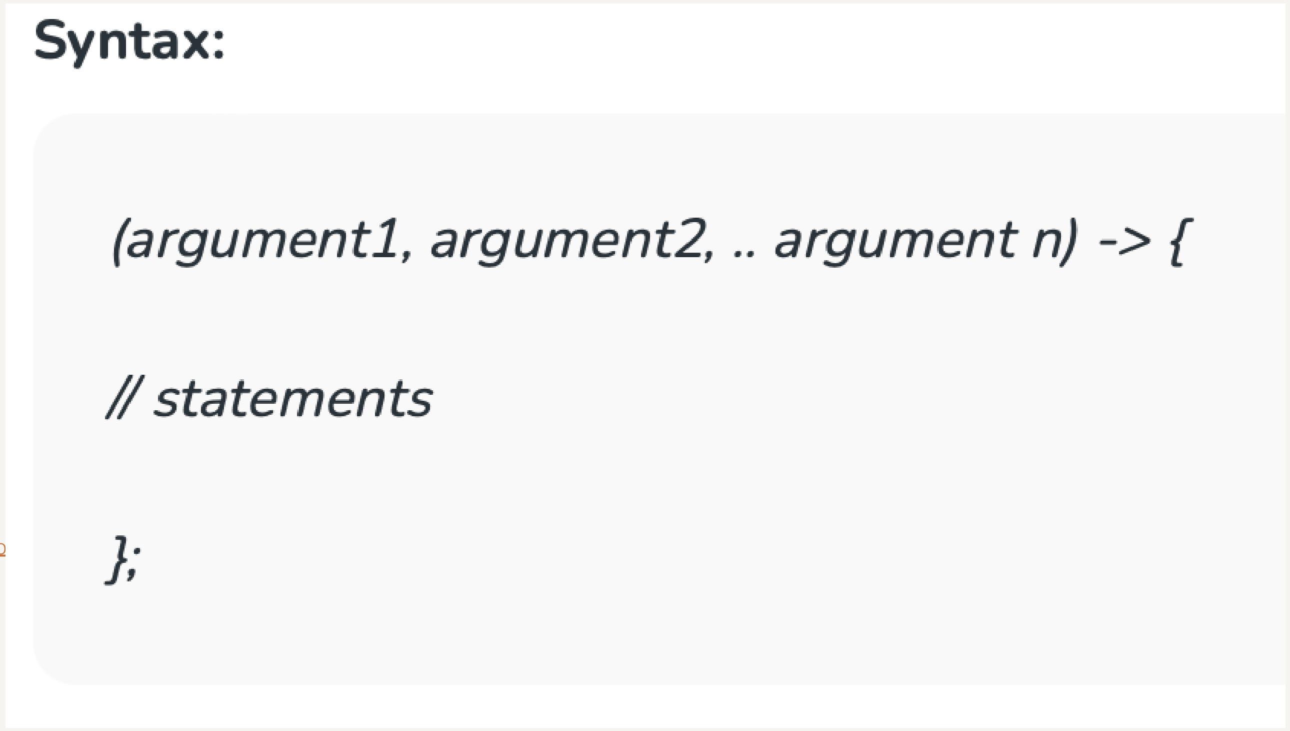


```
Thread t = new Thread(new MyThread());
```

Main difference

Create Thread using Lambda Expressions

- Here we make use of the **Runnable Interface.** As it is a Functional Interface, Lambda expressions can be used. The syntax is shown below:



Syntax:

```
(argument1, argument2, .. argument n) -> {

// statements

};
```

- https://www.simplilearn.com/tuto

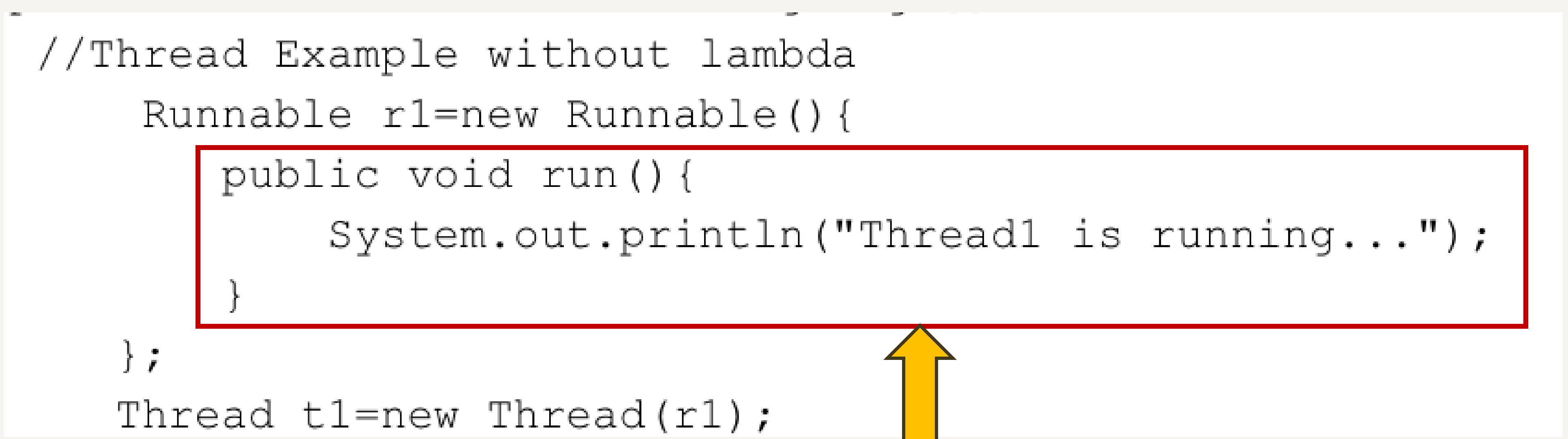
Create Thread using Lambda Expressions: Example

```
public class LambdaExpressionExample {
    public static void main(String args[]){
     //Thread Example without lambda
         Runnable r1=new Runnable(){
            public void run(){
                System.out.println("Thread1 is running...");
            }
        };
        Thread t1=new Thread(r1);
        t1.start();
        //Thread Example with lambda
        Runnable r2=()->{
                System.out.println("Thread2 is running...");
        };
        Thread t2=new Thread(r2);
        t2.start();
    }
}
```

Output

```
Thread1 is running..
Thread2 is running..
```

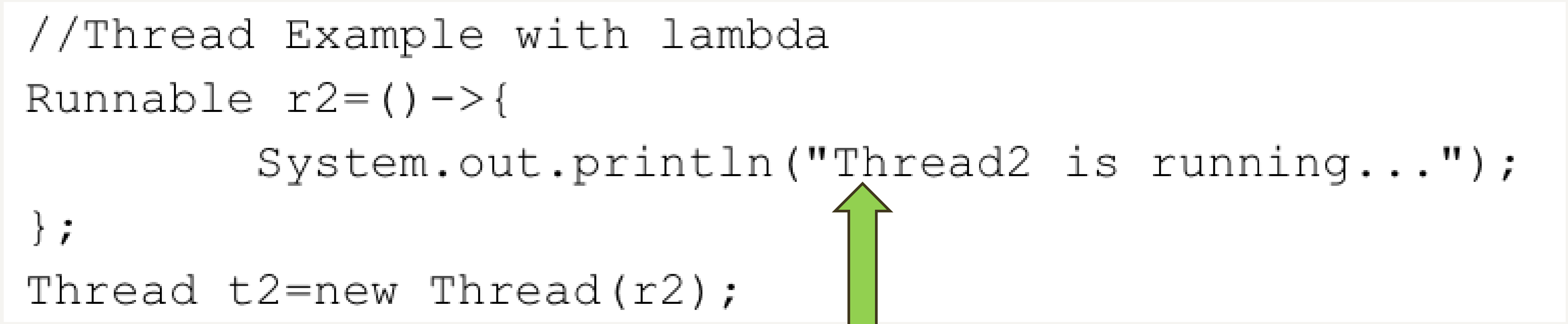
Further explanation for the above example



```
//Thread Example without lambda
    Runnable r1=new Runnable(){
      public void run(){
          System.out.println("Thread1 is running...");
      }
   };
   Thread t1=new Thread(r1);
```

Here, we implement the run() method

Further explanation for the above example



```
//Thread Example with lambda
Runnable r2=()->{
        System.out.println("Thread2 is running...");
};
Thread t2=new Thread(r2);
```

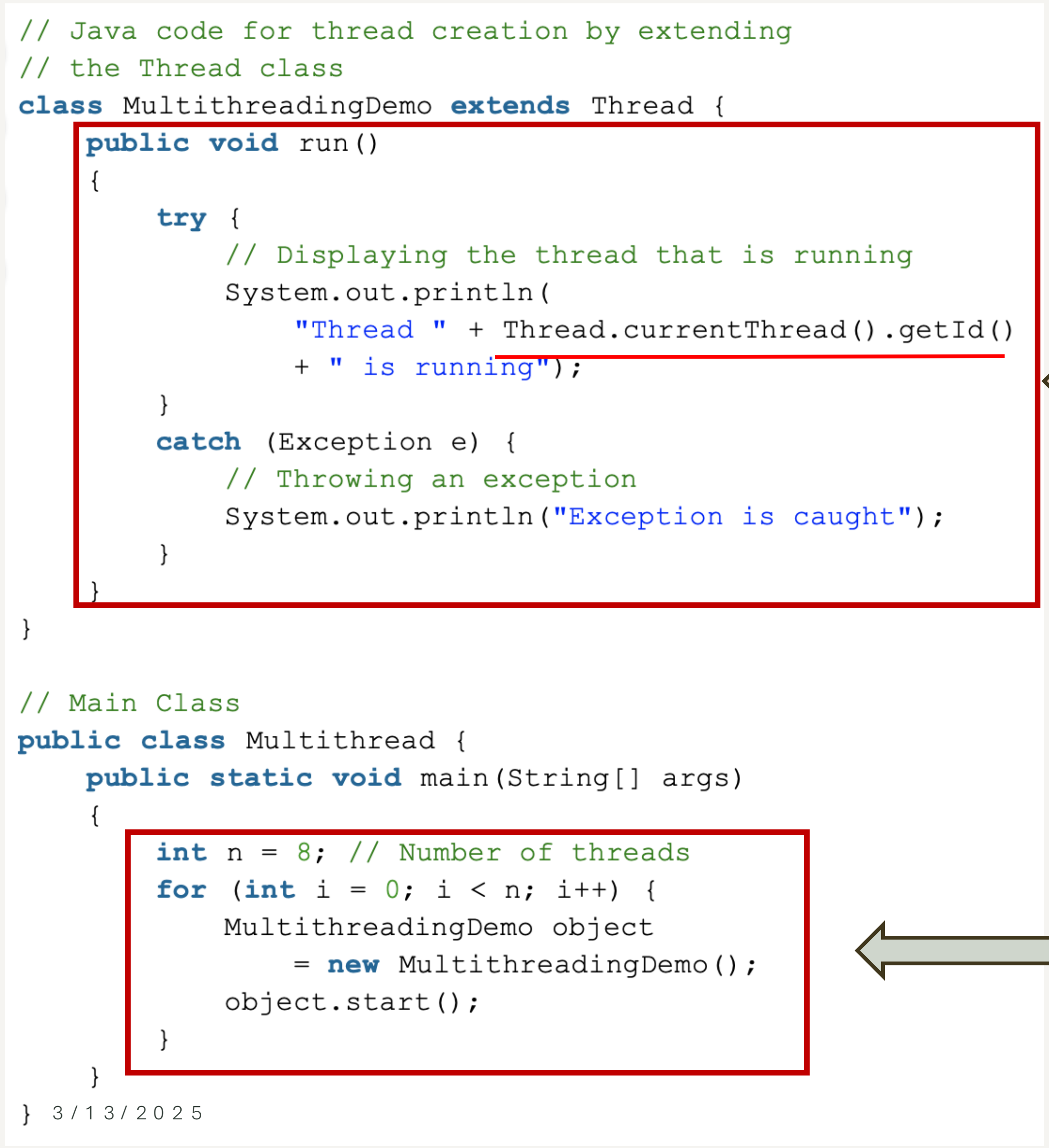
Here, we use lambda to specify the functionality of run() method.

Multithreading

Multithreading in Java

- We still use the following two methods to realize multithread:
  - Extending the Thread class
  - Implementing the Runnable Interface

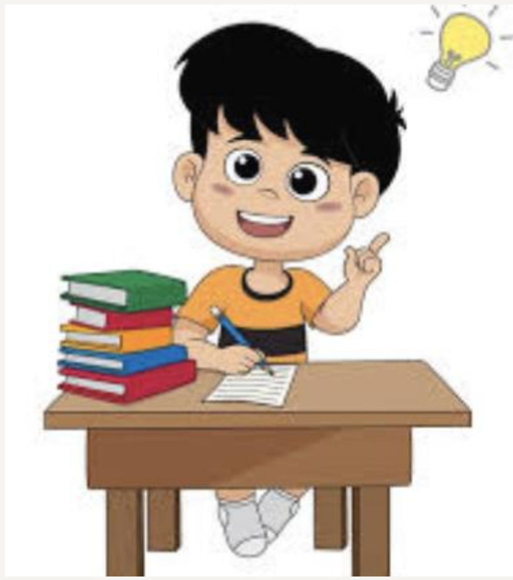
Multithreading by Extending the Thread class



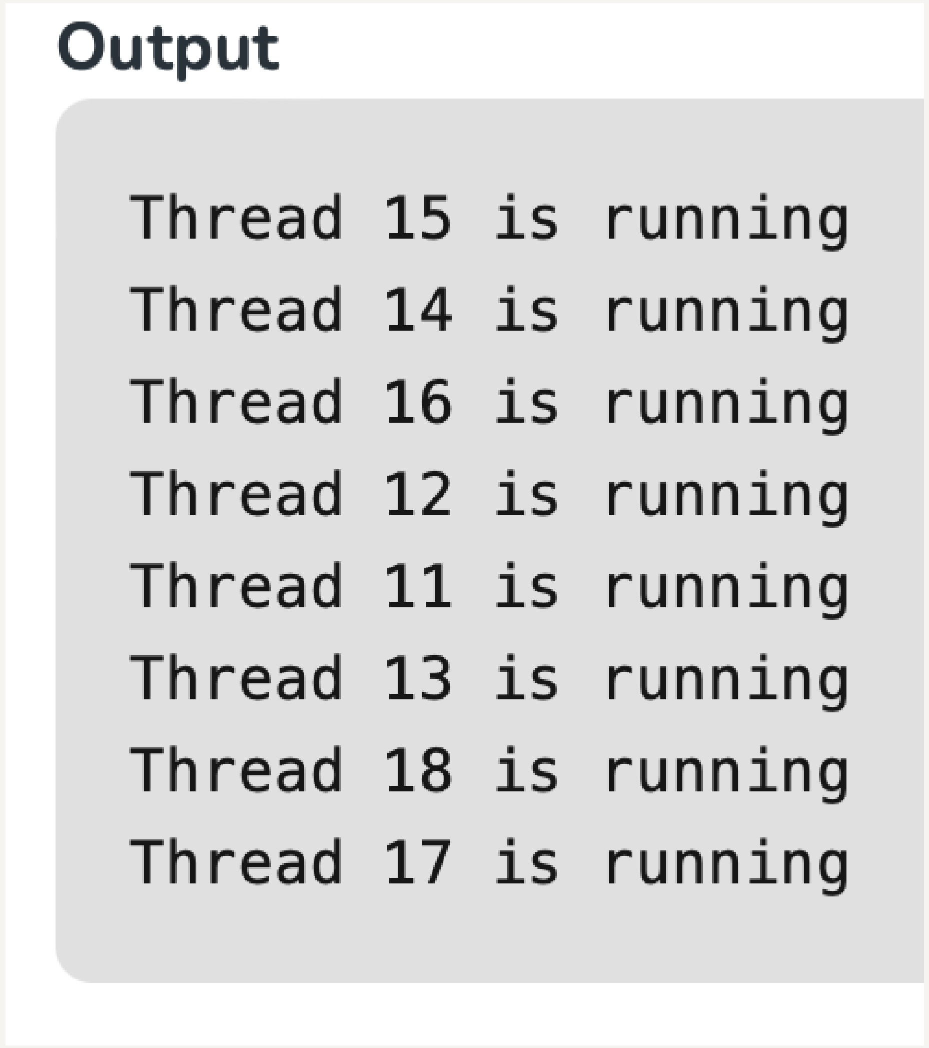
```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Using try-catch to capture the potential exceptions.

Create one more threads.



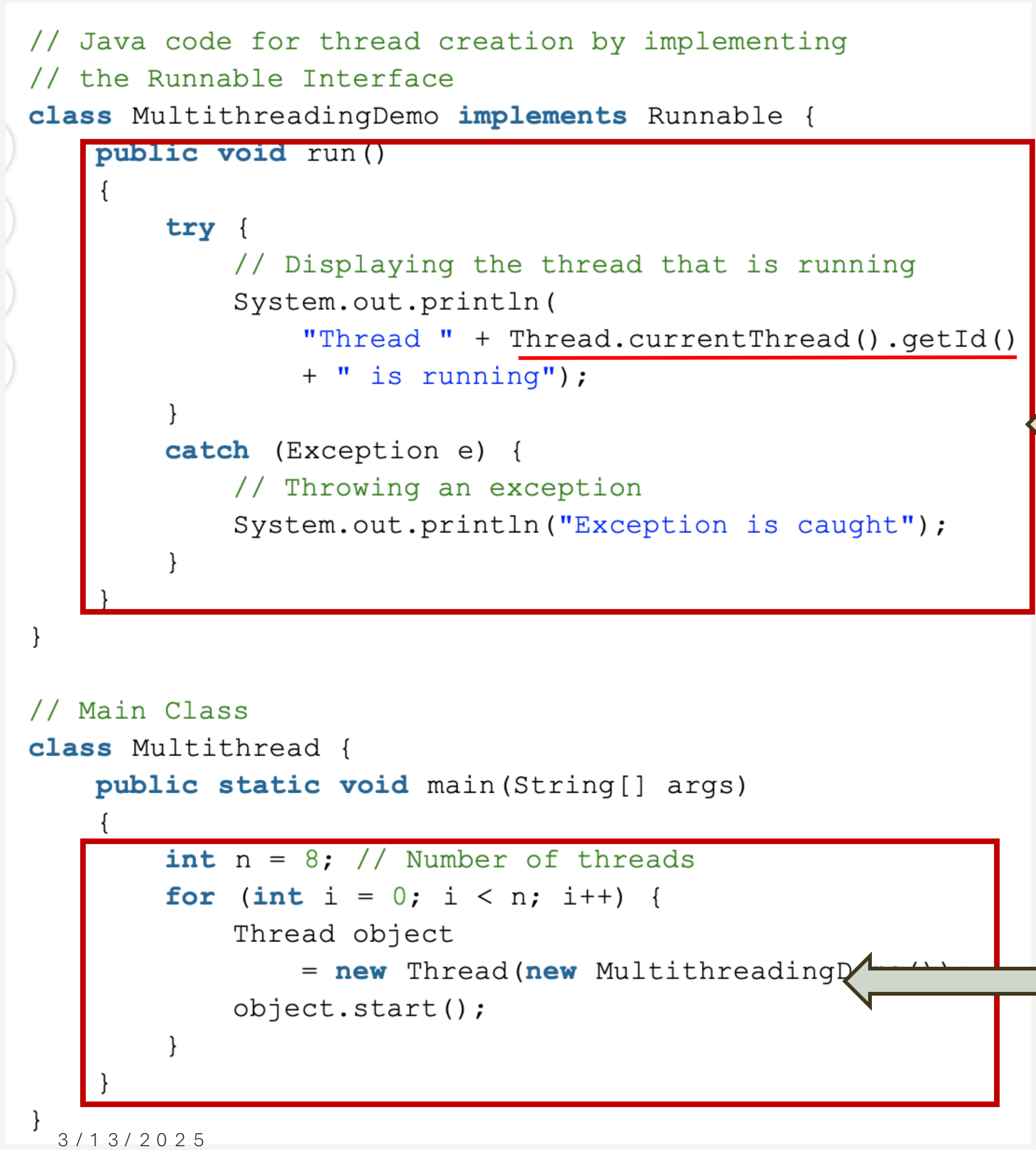
Output for the above example



Output

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Multithreading by Implementing the Runnable Interface

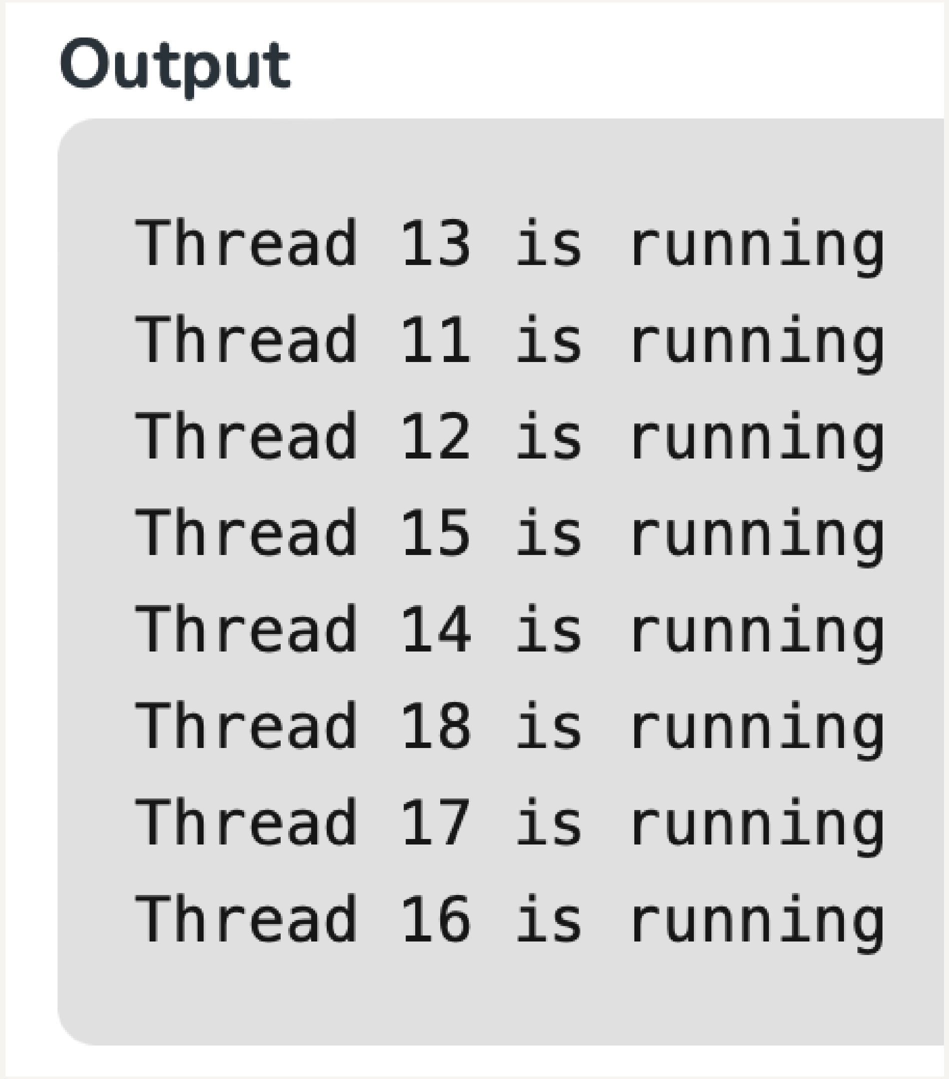


```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Using try-catch to capture the potential exceptions.

Create one more threads.

Output for the above example


Output

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Method getId()

```
Thread.currentThread().getId()
```

$\underbrace{\texttt{Thread.currentThread()}}_{\text{Give us current thread}}$

Give us current thread

Give us the ID of current thread

Java Concurrency

Two Methods about Concurrency

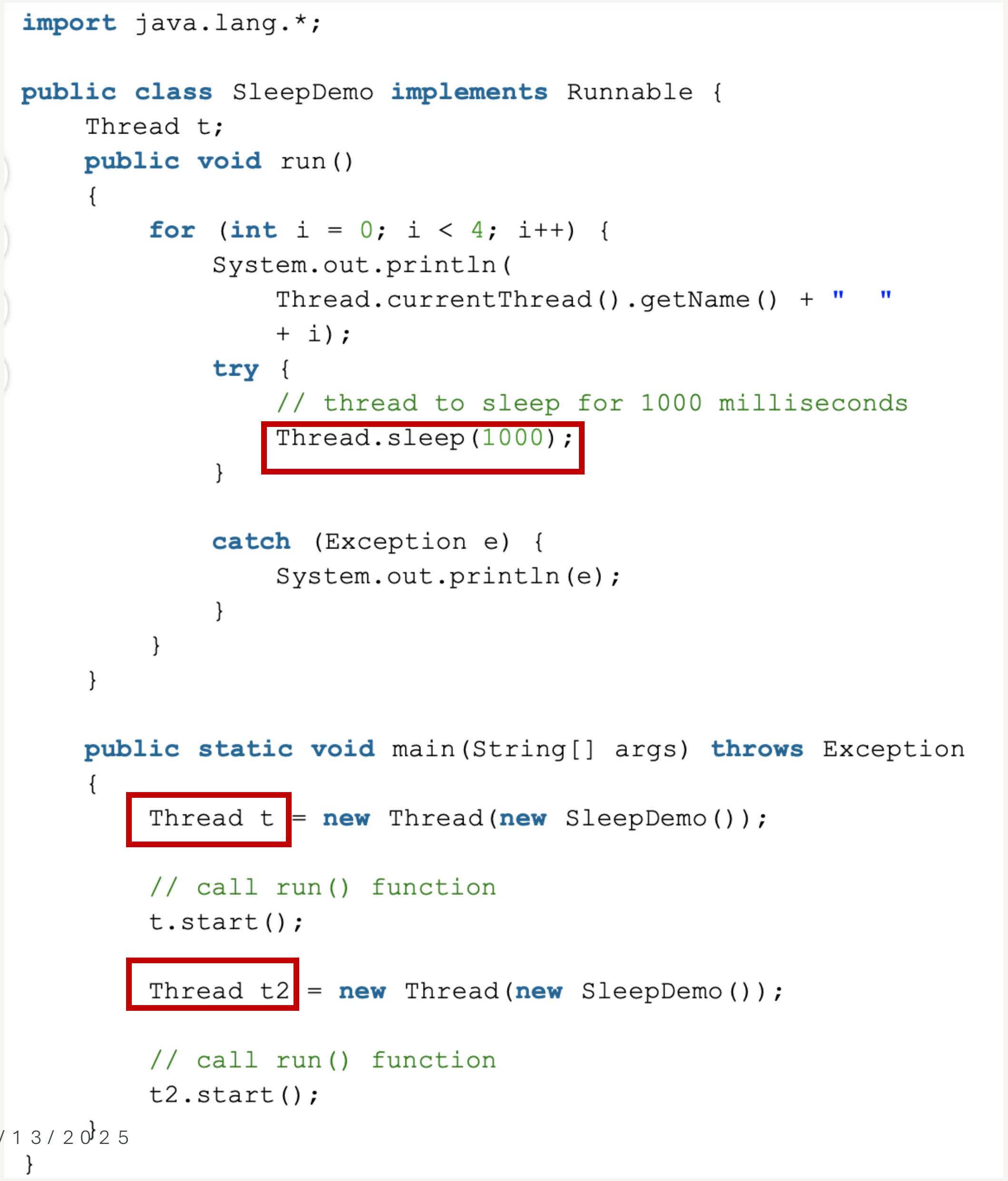
sleep()

join()

sleep() method

- This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- Usage: Thread.sleep(*milliseconds*)

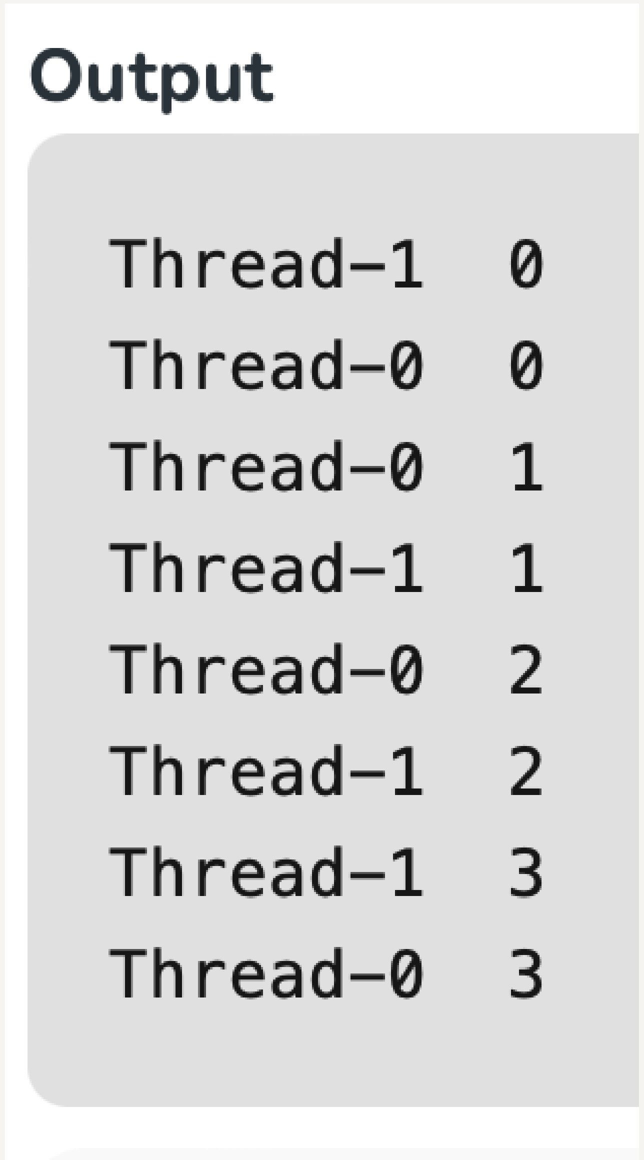
One example



```
import java.lang.*;

public class SleepDemo implements Runnable {
    Thread t;
    public void run()
    {
        for (int i = 0; i < 4; i++) {
            System.out.println(
                Thread.currentThread().getName() + "  "
                + i);
            try {
                // thread to sleep for 1000 milliseconds
                Thread.sleep(1000);
            }

            catch (Exception e) {
                System.out.println(e);
            }
        }
    }

    public static void main(String[] args) throws Exception
    {
        Thread t = new Thread(new SleepDemo());

        // call run() function
        t.start();

        Thread t2 = new Thread(new SleepDemo());

        // call run() function
        t2.start();
    }
}
```

In this case, we have two threads, and the sleep method will cause one of them to sleep for a while and then continue executing that thread.

Output for the above example



Output

```
Thread-1  0
Thread-0  0
Thread-0  1
Thread-1  1
Thread-0  2
Thread-1  2
Thread-1  3
Thread-0  3
```

Note about using sleep method

- Based on the requirement we can make a thread to be in a sleeping state for a specified period of time
- sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power-saving mode)

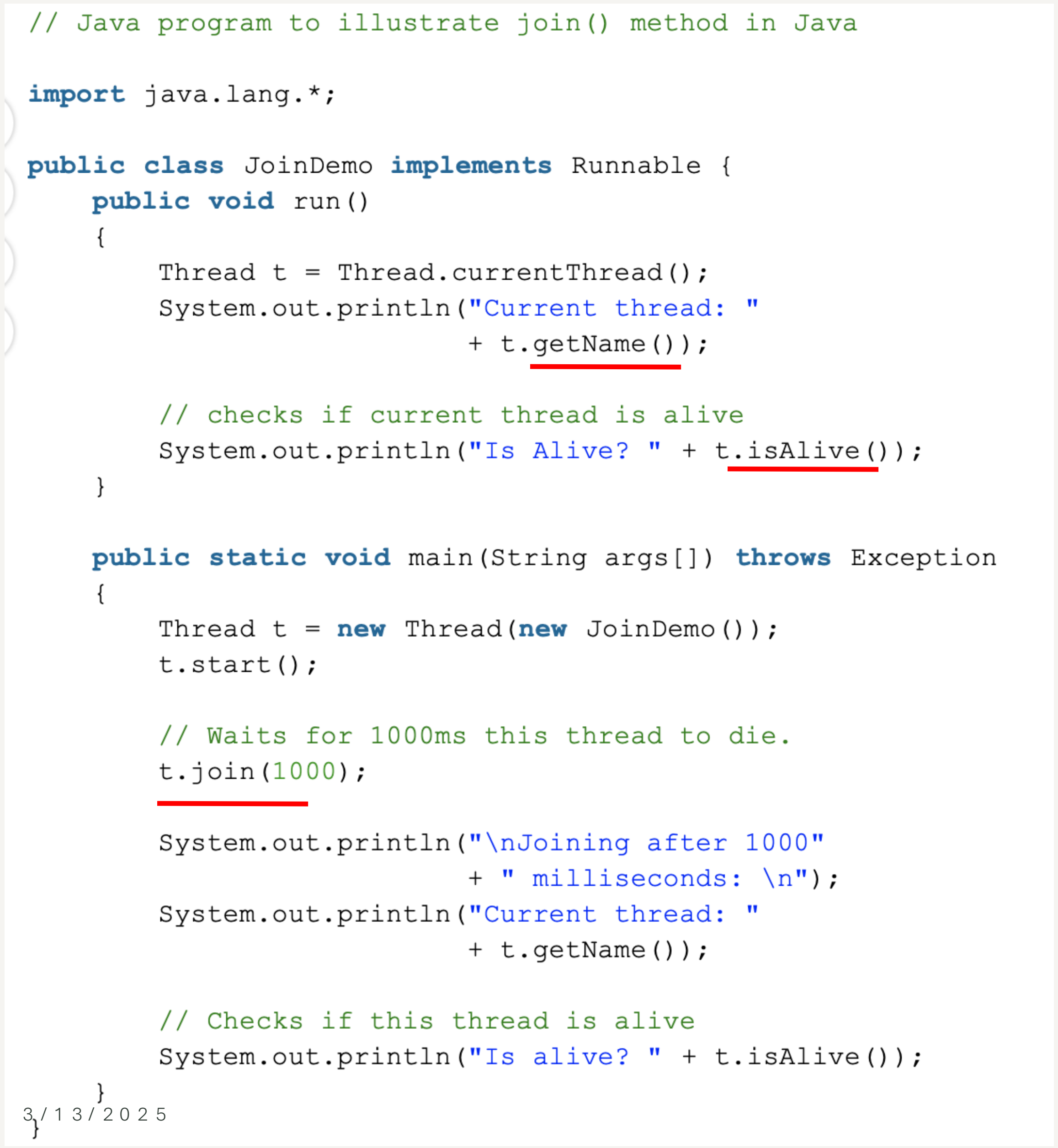
join() method

- The join() method of a Thread instance is used to join the start of a thread's execution to the end of another thread's execution such that a thread does not start running until another thread ends.
- If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.

Block means stopping execution

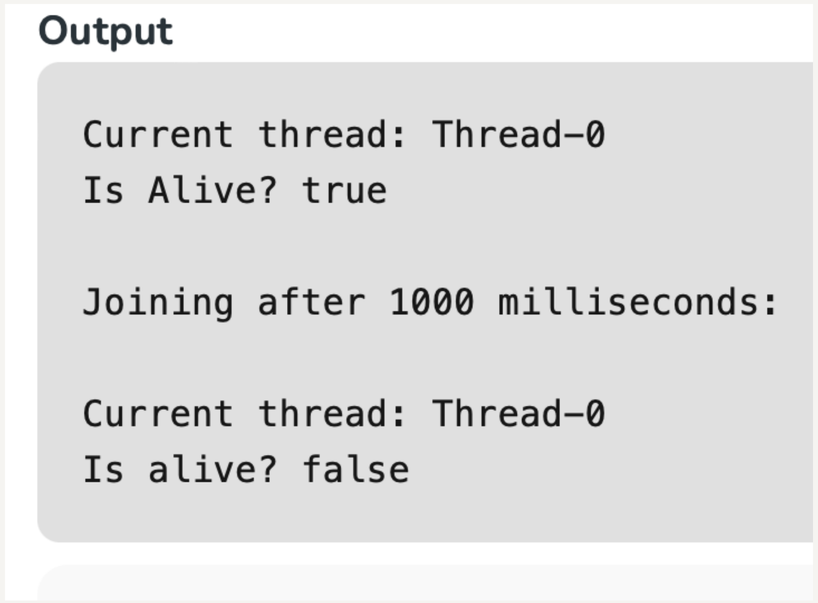
- The join() method waits at most this many milliseconds for this thread to die.

One example



```java
// Java program to illustrate join() method in Java

import java.lang.*;

public class JoinDemo implements Runnable {
    public void run()
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: "
                           + t.getName());

        // checks if current thread is alive
        System.out.println("Is Alive? " + t.isAlive());
    }

    public static void main(String args[]) throws Exception
    {
        Thread t = new Thread(new JoinDemo());
        t.start();

        // Waits for 1000ms this thread to die.
        t.join(1000);

        System.out.println("\nJoining after 1000"
                           + " milliseconds: \n");
        System.out.println("Current thread: "
                           + t.getName());

        // Checks if this thread is alive
        System.out.println("Is alive? " + t.isAlive());
    }
}
```

The Join() method causes the current thread to wait 1000 milliseconds before dying.



Output

```
Current thread: Thread-0
Is Alive? true

Joining after 1000 milliseconds:

Current thread: Thread-0
Is alive? false
```

getName() and isAlive() methods

- Thread.currentThread.getName()
  - Get the name of current thread
- Thread.currentThread.isAlive()
  - Check the alive status of current thread

Synchronization

Synchronization in Java

- In Multithreading, **Synchronization** is crucial for ensuring that multiple threads operate safely on shared resources.
- Without **Synchronization**, data inconsistency or corruption can occur when multiple threads try to access and modify shared variables simultaneously.
- In Java, it is a mechanism that ensures that **only one thread** can access a resource at any given time.
- This process helps prevent issues such as data inconsistency when multiple threads interact with shared resources.

Syntax about using Synchronization

- Adding synchronized keyword to the header of one method.

```java
// Synchronized method to increment counter
public synchronized void inc() {
     c++;
}
```

```java
// Synchronized method to get counter value
public synchronized int get() {
     return c;
}
```

One example

```java
// Java Program to demonstrate synchronization in Java
class Counter {
    private int c = 0; // Shared variable

    // Synchronized method to increment counter
    public synchronized void inc() {
        c++;
    }

    // Synchronized method to get counter value
    public synchronized int get() {
        return c;
    }
}
```

int() method add 1 to shared variable c per time. And get() method will return the value of c. Adding synchronized keyword to both methods will avoid data inconsistency.

Output

```
Counter: 2000
```

lambda

```java
public class Geeks {
    public static void main(String[] args) {
        Counter cnt = new Counter(); // Shared resource

        // Thread 1 to increment counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                cnt.inc();
            }
        });

        // Thread 2 to increment counter
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                cnt.inc();
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // Wait for threads to finish
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print final counter value
        System.out.println("Counter: " + cnt.get());
    }
}
```

Explanation for the above example

- Two threads, t1 and t2, increment the shared counter variable concurrently.
- The inc() and get() methods are synchronized, meaning only one thread can execute these methods at a time, preventing race conditions.
- The program ensures that the final value of the counter is consistent and correctly updated by both threads.

Synchronized block

```java
// Method with synchronization block
public void inc() {
    synchronized(this) { // Synchronize only this block
        c++;
    }
}
```

We could put synchronized keyword inside one function

```java
// Synchronized method to increment counter
public synchronized void inc() {
    c++;
}
```

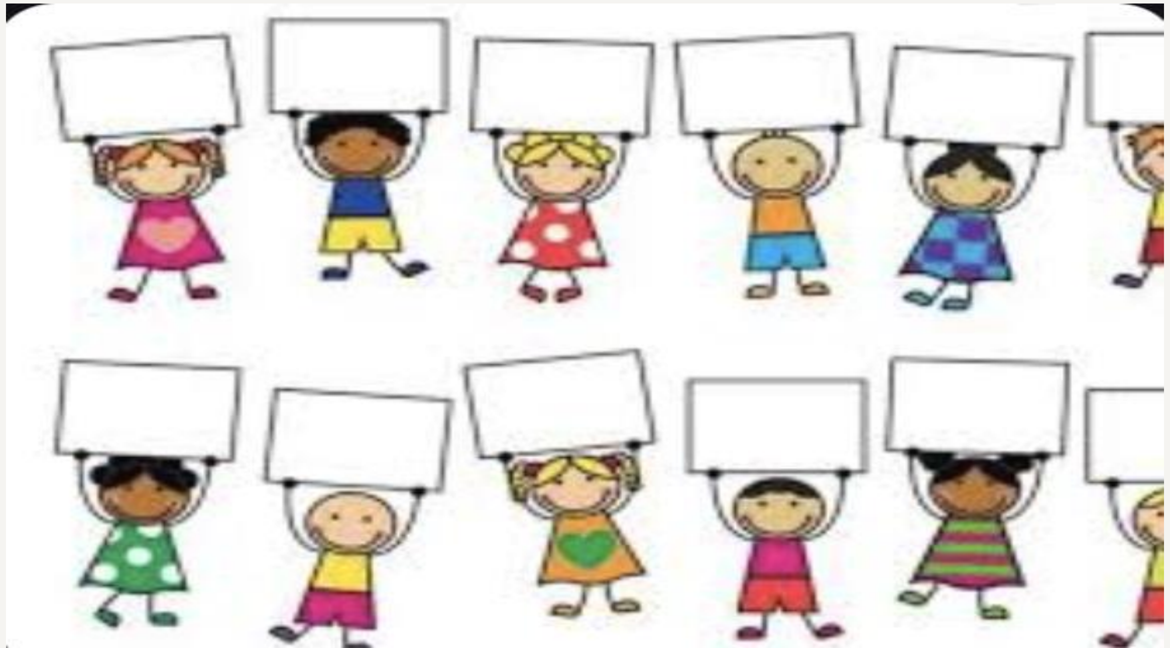
Types of Synchronization

- There are two synchronizations in Java mentioned below:
  - Process Synchronization
  - Thread Synchronization *

- See reference https://www.geeksforgeeks.org/synchronization-in-java/

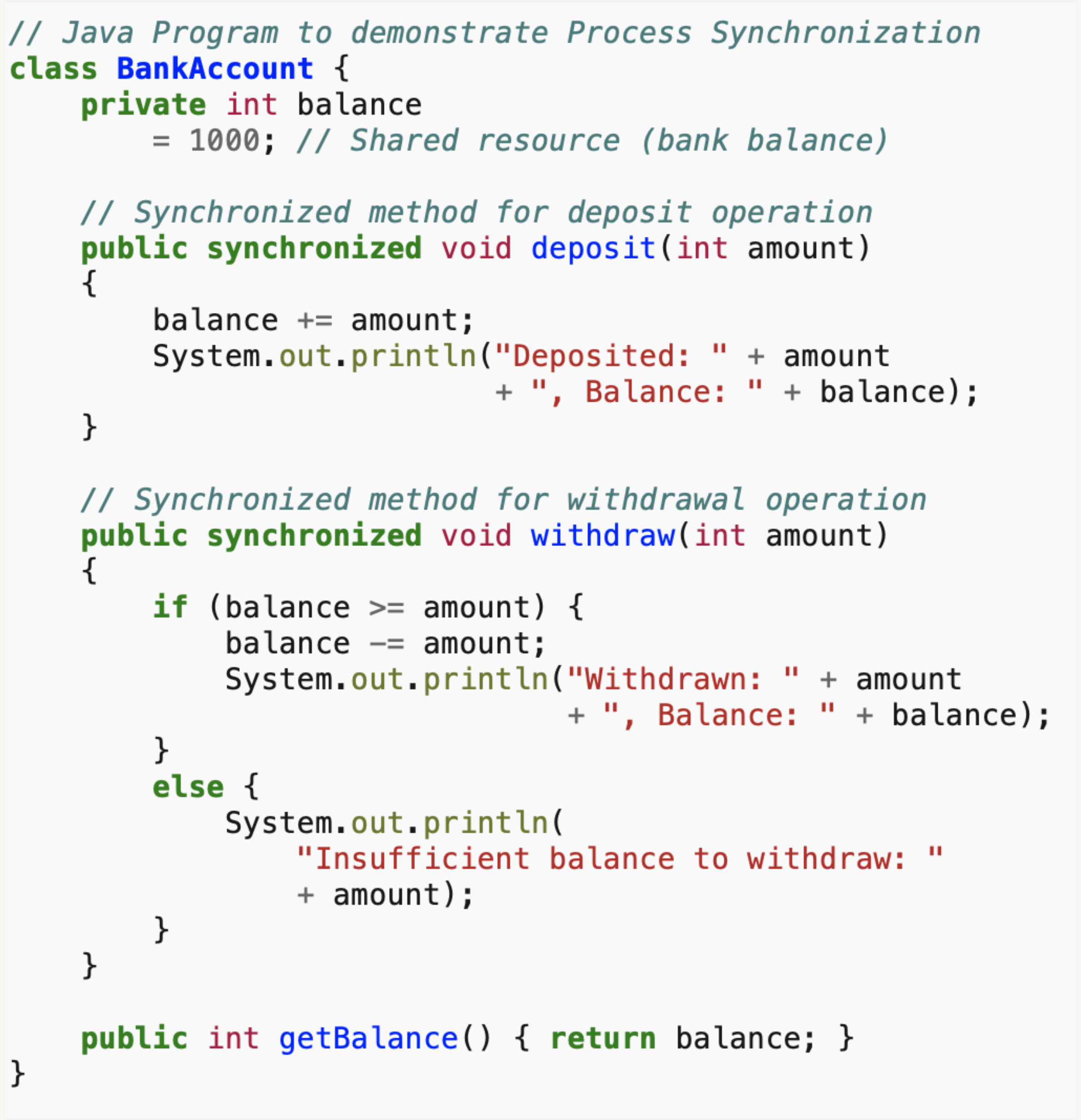
Process Synchronization

- Process Synchronization is a technique used to coordinate the execution of multiple processes. It ensures that the shared resources are safe and in order.



One Example

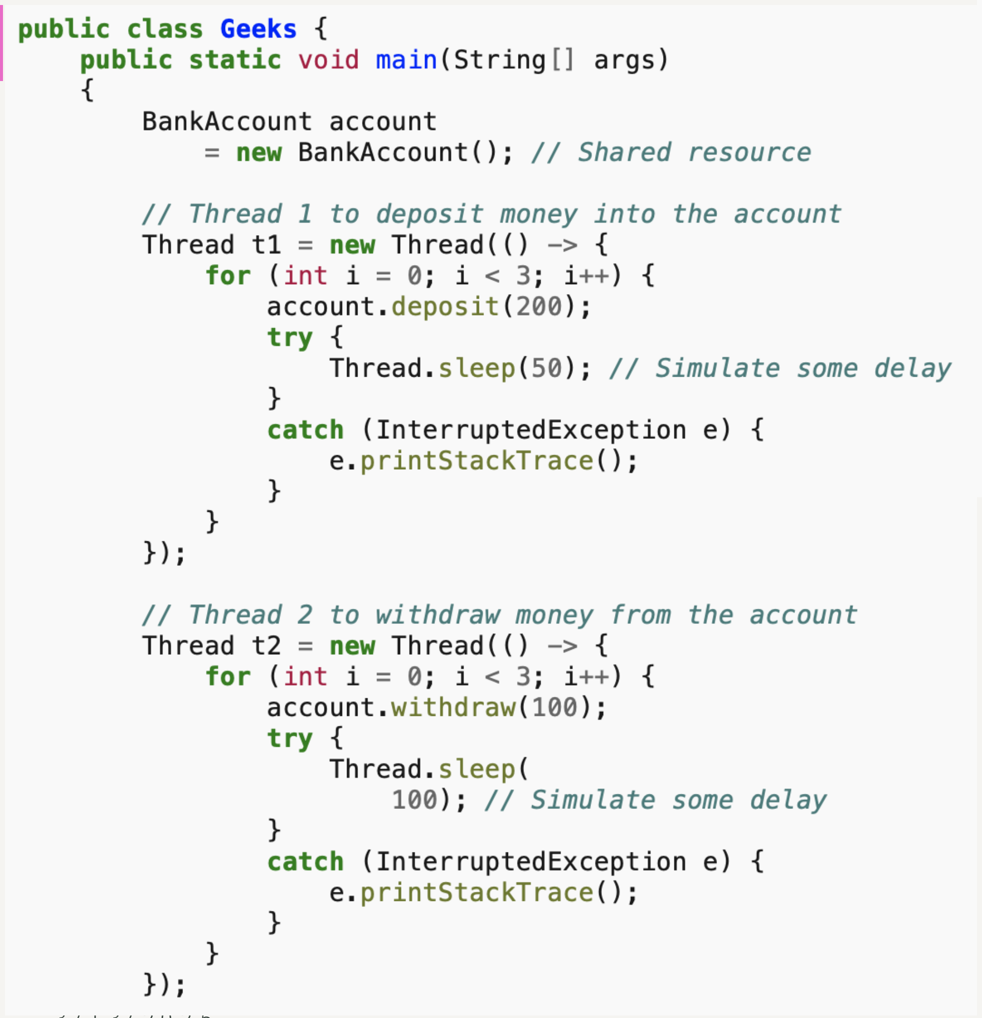
1



```java
// Java Program to demonstrate Process Synchronization
class BankAccount {
    private int balance
        = 1000; // Shared resource (bank balance)

    // Synchronized method for deposit operation
    public synchronized void deposit(int amount)
    {
        balance += amount;
        System.out.println("Deposited: " + amount
                           + ", Balance: " + balance);
    }

    // Synchronized method for withdrawal operation
    public synchronized void withdraw(int amount)
    {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount
                               + ", Balance: " + balance);
        }
        else {
            System.out.println(
                "Insufficient balance to withdraw: "
                + amount);
        }
    }

    public int getBalance() { return balance; }
}
```

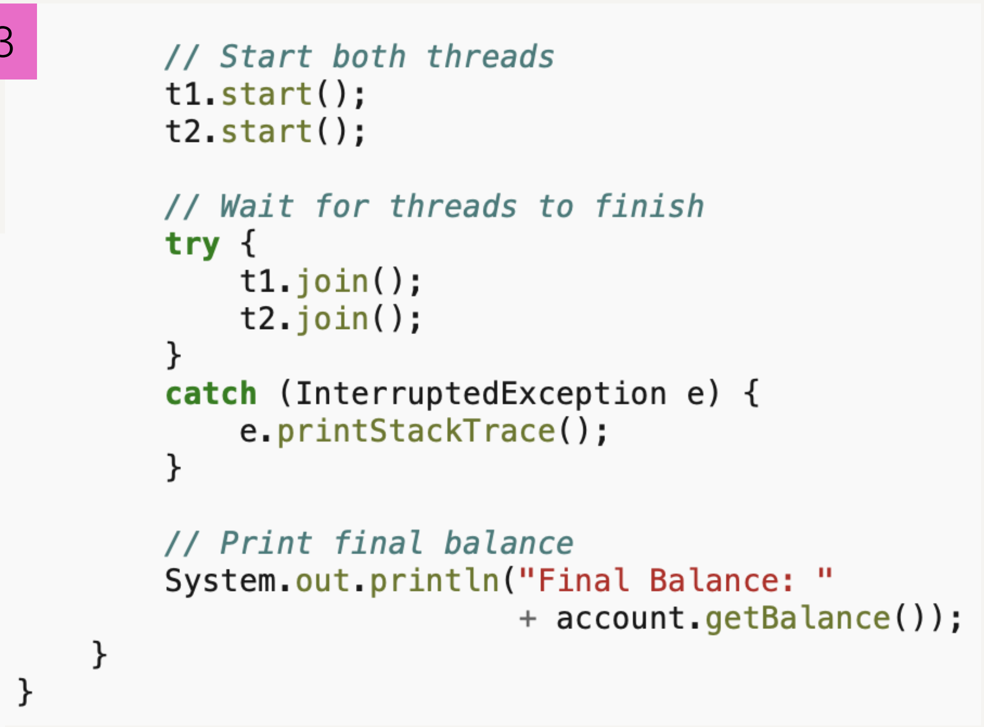
One Example: Cont.

2

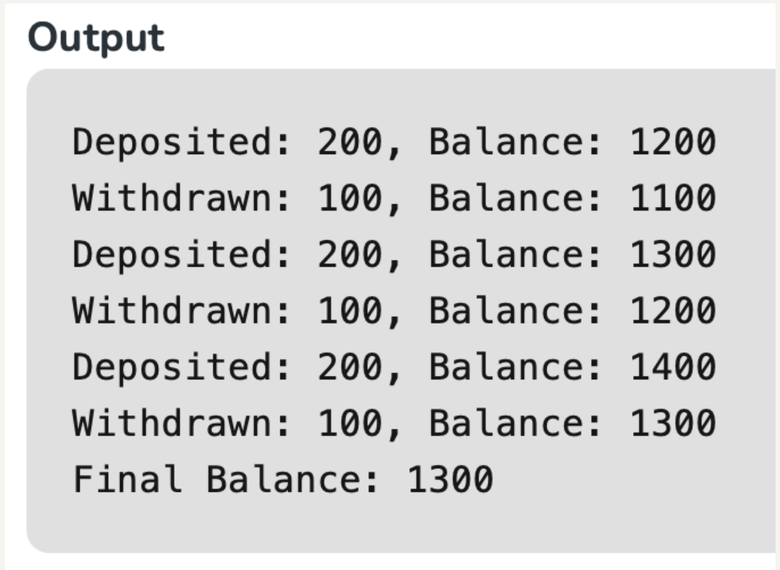


```java
public class Geeks {
    public static void main(String[] args)
    {
        BankAccount account
            = new BankAccount(); // Shared resource

        // Thread 1 to deposit money into the account
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                account.deposit(200);
                try {
                    Thread.sleep(50); // Simulate some delay
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Thread 2 to withdraw money from the account
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                account.withdraw(100);
                try {
                    Thread.sleep(
                        100); // Simulate some delay
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

3



```java
        // Start both threads
        t1.start();
        t2.start();

        // Wait for threads to finish
        try {
            t1.join();
            t2.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print final balance
        System.out.println("Final Balance: "
                           + account.getBalance());
    }
}
```



Output

```
Deposited: 200, Balance: 1200
Withdrawn: 100, Balance: 1100
Deposited: 200, Balance: 1300
Withdrawn: 100, Balance: 1200
Deposited: 200, Balance: 1400
Withdrawn: 100, Balance: 1300
Final Balance: 1300
```

More explanation for the above example

- It demonstrates process synchronization using a bank account with deposit and withdrawal operations.
- Two threads, one for depositing and one for withdrawing, perform operations on the shared account.
- The methods **deposit()** and **withdraw()** are synchronized to ensure thread safety, preventing race conditions when both threads access the balance simultaneously.
- This ensures accurate updates to the account balance.

End