Learn With Guidemy

Tackling Real-World Concurrency Challenges with Expert Solutions

Java Concurrency Unlocked

Guidemy

Objectives

After this module you will be able to:

- Understand the core concepts, principles, and techniques related to threading and concurrent programming in Java.
- Enhance problem-solving skills by working through actual concurrency issues from top-tier investment banks.
- Learn the best practices for addressing race conditions, synchronization issues, and deadlocks in Java applications.
- Implement the Producer-Consumer pattern and control thread execution in Java applications.

Guidemy

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

What is Concurrency?

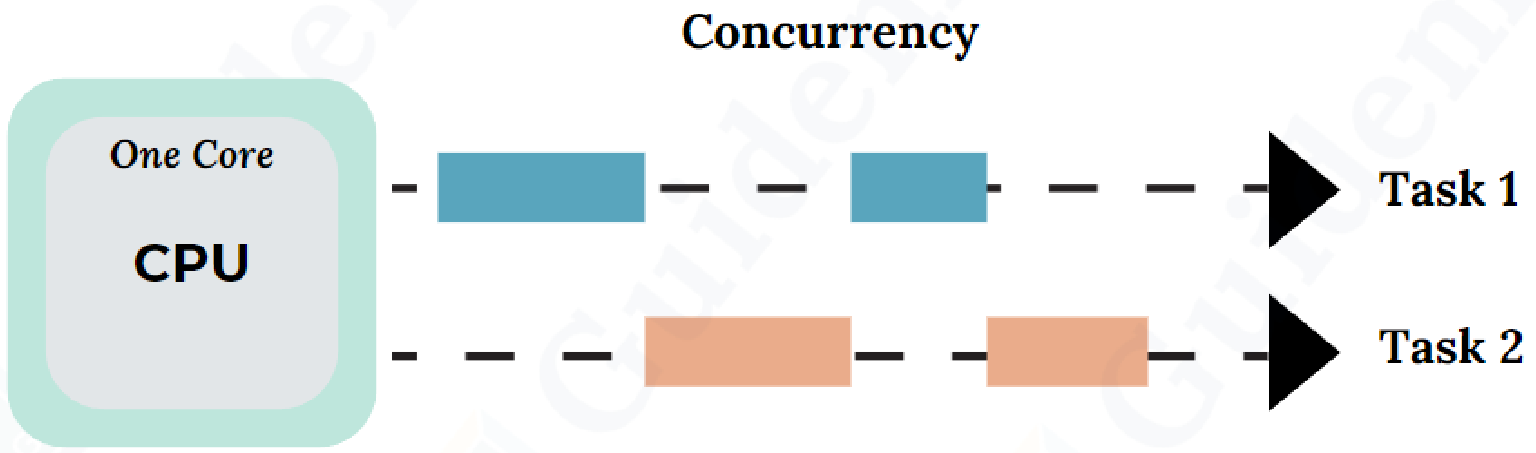
- In computer science, concurrency is the execution of the multiple instruction sequences **at the same time**.
- In more technical terms, concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed **out-of-order** or in **partial** order, without affecting the outcome.
- This allows for **parallel** execution of the concurrent units, which can significantly improve overall speed of the execution in multiprocessor and multicore systems.

What is Concurrency?

- It may also refer to the ***decomposability*** of a program, algorithm, or problem into order-independent or partially-ordered components or units of computation.

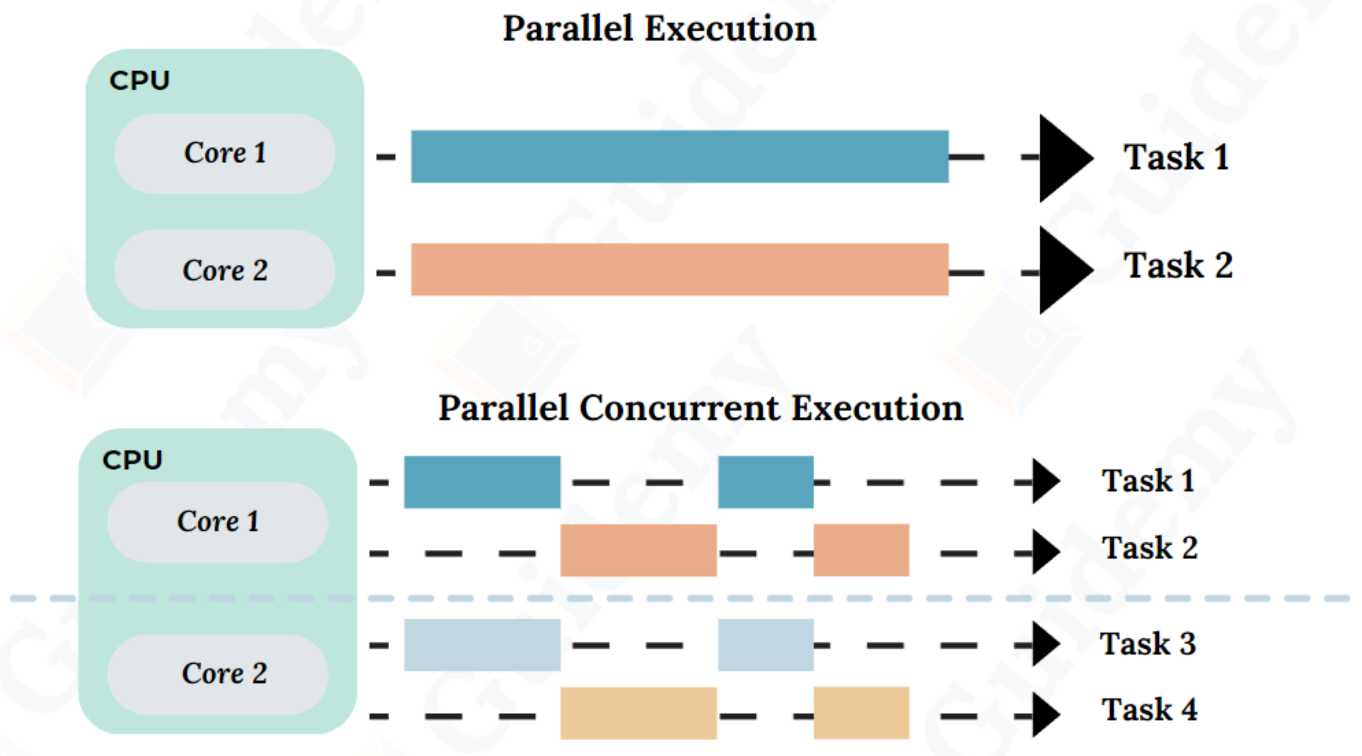
Case 1: CPU with only one core

- CPU will be executing processes one by one, individually by **time slice**.
- A time slice is short time frame that gets assigned to process for CPU execution.



Case 2: CPU with multiple cores

- Only on multicore CPU system, multiple processes execute at the same time on different cores.



What is CPU Scheduling?

- CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I/O etc., thus making full use of the CPU.
- Whenever the CPU becomes idle, the operating system must select one of the processes in the line ready for launch.
- The selection process is done by a **temporary** (CPU) scheduler.
- The Scheduler selects between memory processes ready to launch and assigns the CPU to one of them.

When will the Scheduler pause a thread?

- Scheduler may **pause** a thread due to:
  - The thread is waiting for some more data
  - The thread is waiting for another thread to do something
  - CPU should be shared equally among threads

What is a Java Thread?

- We can define threads as a light-weight **subprocess** within the smallest unit of **processes** and having *separate paths* of execution.
- These threads use **shared memory**, but they **act independently**.
    - Hence, if there is an exception in a thread, that will **not** affect the working of other threads despite them sharing the same memory.

Few points about Java Thread

- Thread is a set of instructions defined at *Operating System* level
- Lightweight sub-process through which we can perform **multiple** activities within a single process
- An application can be composed of **multiple** threads
  - => JVM itself works with several threads like GC, JIT, etc.
- Different threads can be executed at the **same** time on different cores or CPUs

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

Race Condition

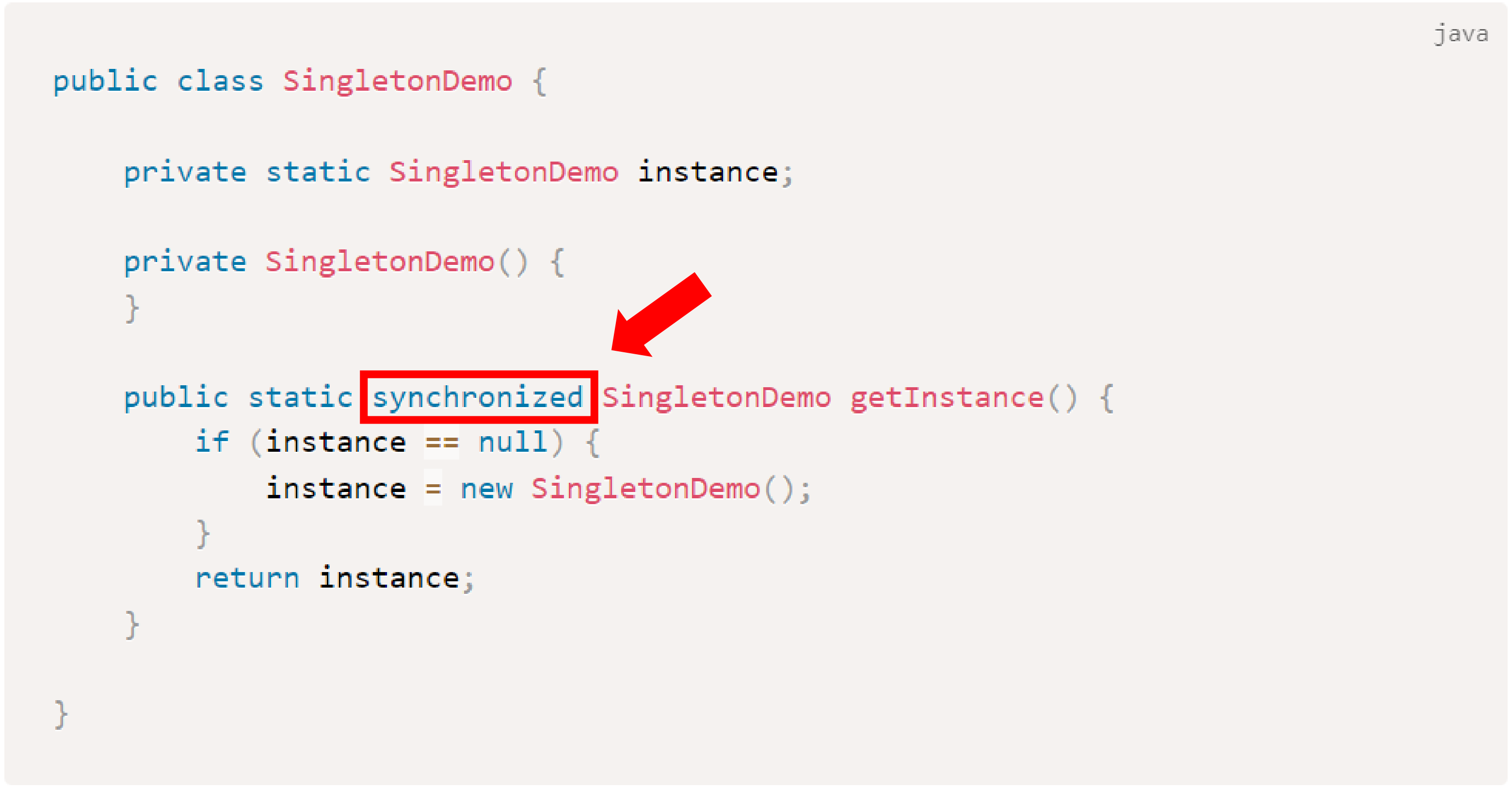
- A race condition, in the simplest terms, is when two or more actions **compete to complete first**, causing **unpredictable** results.
- Example:
  - Imagine two people trying to withdraw money from a joint bank account with a balance of $100. Both of them want to withdraw $80, but there's not enough money in the account for both withdrawals.
- In programming, this can happen when two or more threads access and modify a shared resource **simultaneously** without proper synchronization, leading to unpredictable and unintended results.

Interview Questions #1: Merrill Lynch - Demonstrate race condition in Singleton pattern

java

```java
public class SingletonDemo {

    private static SingletonDemo instance;

    private SingletonDemo() {
    }

    public static SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }
        return instance;
    }

}
```

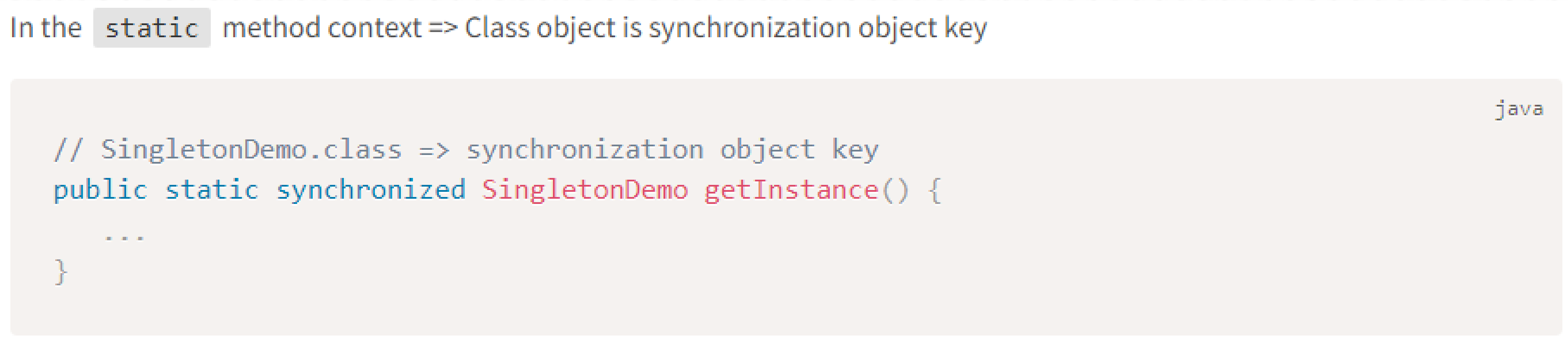
Interview Questions #1: Merrill Lynch - Demonstrate race condition in Singleton pattern **(Solution)**



```
java
public class SingletonDemo {

    private static SingletonDemo instance;

    private SingletonDemo() {
    }

    public static synchronized SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }
        return instance;
    }

}
```

More on Synchronization

- For synchronization to work, we need a synchronization object key also called as **monitor** or **mutex**.
  - Every Java object can play as monitor or mutex.



In the `static` method context => Class object is synchronization object key

java

```java
// SingletonDemo.class => synchronization object key
public static synchronized SingletonDemo getInstance() {
    ...
}
```

More on Synchronization

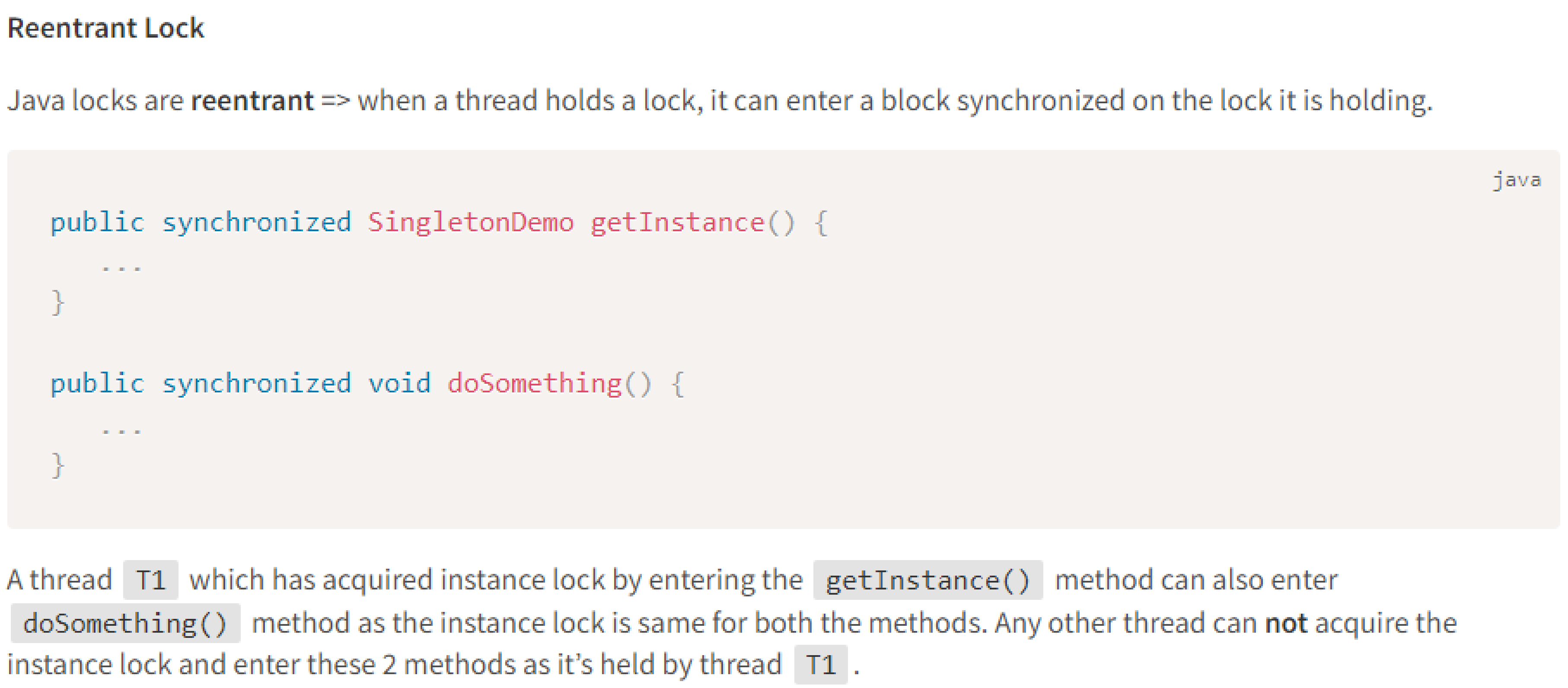
In the `non-static` method context => instance object is synchronization object key

```java
// instance object which invokes this method is synchronization object key
public synchronized SingletonDemo getInstance() {
    ...
}
```

Or, we can also explicitly use a Java object as synchronization object key using **synchronized block**

```java
// instance object used explicitly as synchronization object key in synchronized block
public SingletonDemo getInstance() {
    synchronized(this) {
      ...
    }
}
```

More on Synchronization



Reentrant Lock

Java locks are **reentrant** => when a thread holds a lock, it can enter a block synchronized on the lock it is holding.

java

```java
public synchronized SingletonDemo getInstance() {
    ...
}

public synchronized void doSomething() {
    ...
}
```

A thread `T1` which has acquired instance lock by entering the `getInstance()` method can also enter `doSomething()` method as the instance lock is same for both the methods. Any other thread can **not** acquire the instance lock and enter these 2 methods as it's held by thread `T1`.

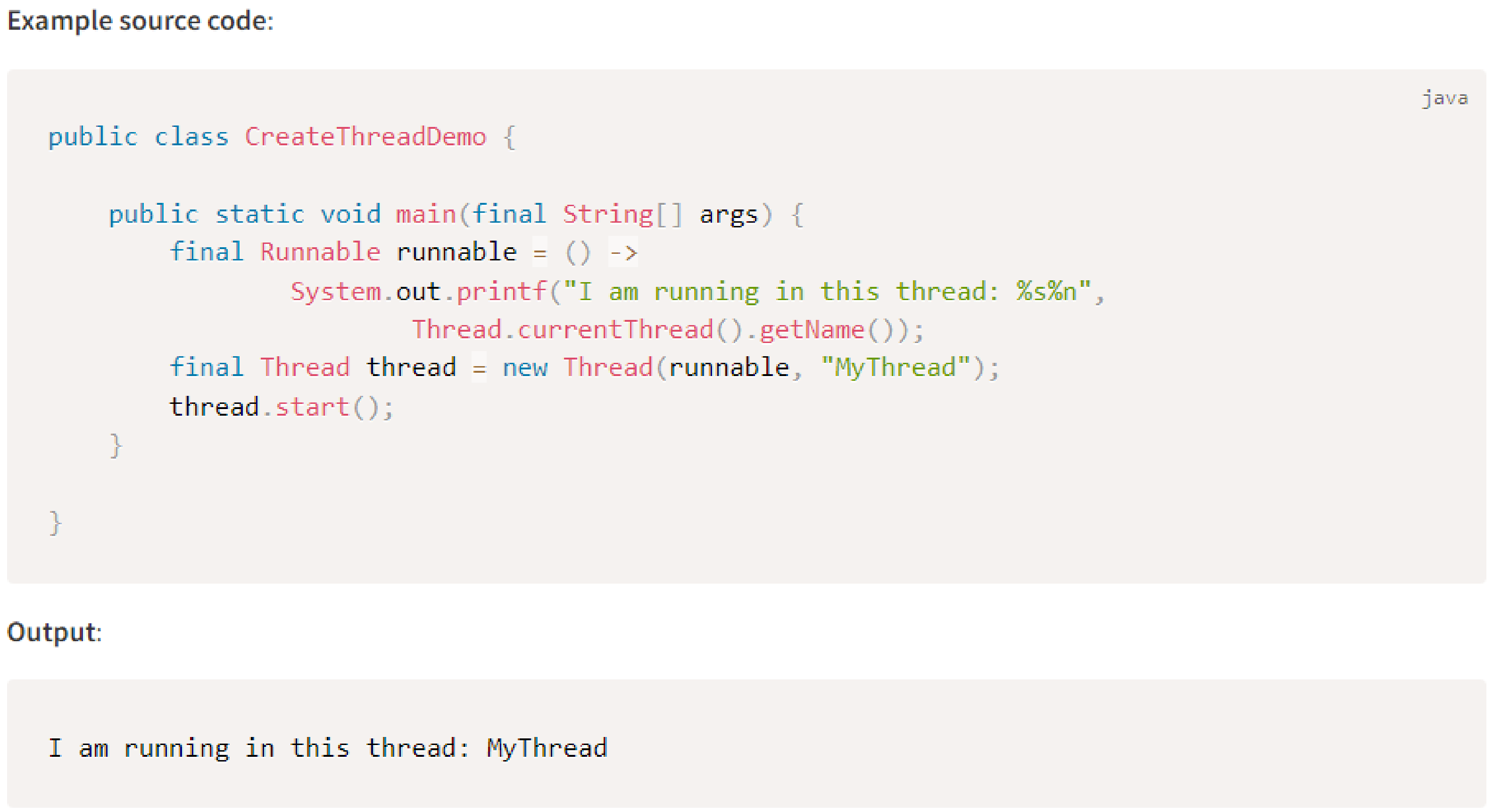
Deadlock

- A *deadlock* is a situation where a thread T1 holds a key needed by a thread T2, and T2 holds the key needed by T1.
- In this situation, both the threads will keep on **waiting for each other indefinitely**.
- There is nothing we can do if the deadlock situation happens but to **restart the JVM**.
- However, even identifying a deadlock is very complex in the modern JVMs or monitoring tools.

How to create and run threads in Java?

- The most basic way is to:
  1. Create a **Runnable** instance
  2. Pass it to the **Thread** constructor
  3. Invoke **start()** method on Thread object

How to create and run threads in Java?



Example source code:

```java
public class CreateThreadDemo {

    public static void main(final String[] args) {
        final Runnable runnable = () ->
                System.out.printf("I am running in this thread: %s%n",
                        Thread.currentThread().getName());
        final Thread thread = new Thread(runnable, "MyThread");
        thread.start();
    }

}
```

Output:

```
I am running in this thread: MyThread
```

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

Guidemy

Interview Questions #2: JP Morgan Chase - Demonstrate synchronization issue and fix the code

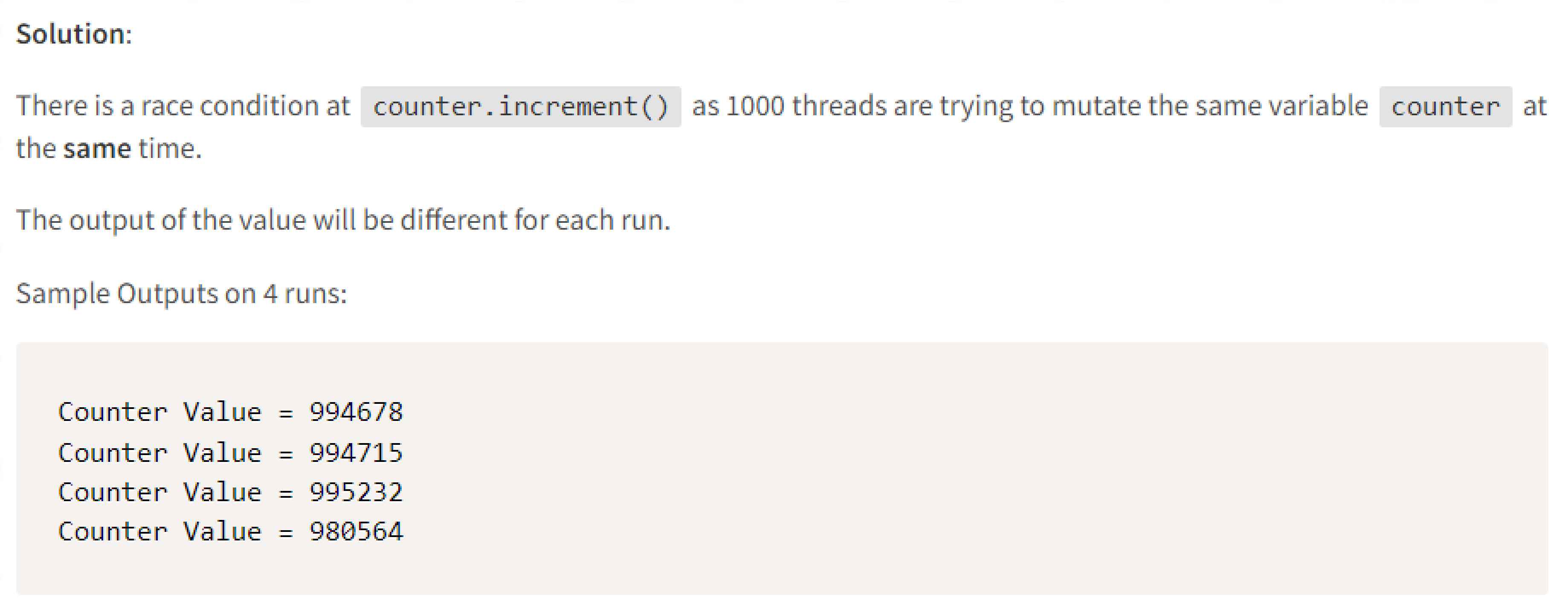
```java
public class Counter {

    private long counter;

    public Counter(final long counter) {
        this.counter = counter;
    }

    public long getCounter() {
        return counter;
    }

    public void increment() {
        counter += 1L;
    }

}
```

Interview Questions #2: JP Morgan Chase - Demonstrate synchronization issue and fix the code

1. What is the output of the counter value?
2. Is the output going to be consistent for every run? If not, what is the issue?
3. If any race condition, fix the code.

```java
public class RaceConditionDemo {

    public static void main(final String[] args) throws InterruptedException {
        final Counter counter = new Counter(0L);
        final Runnable r = () -> {
            for (int i = 0; i < 1_000; i++) {
                counter.increment();
            }
        };

        final Thread[] threads = new Thread[1_000];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(r);
            threads[i].start();
        }

        for (int i = 0; i < threads.length; i++) {
            threads[i].join();
        }

        System.out.printf("Counter Value = %d%n", counter.getCounter());
    }

}
```

Interview Questions #2: JP Morgan Chase - Demonstrate synchronization issue and fix the code (**Solution**)



Solution:

There is a race condition at `counter.increment()` as 1000 threads are trying to mutate the same variable `counter` at the **same** time.

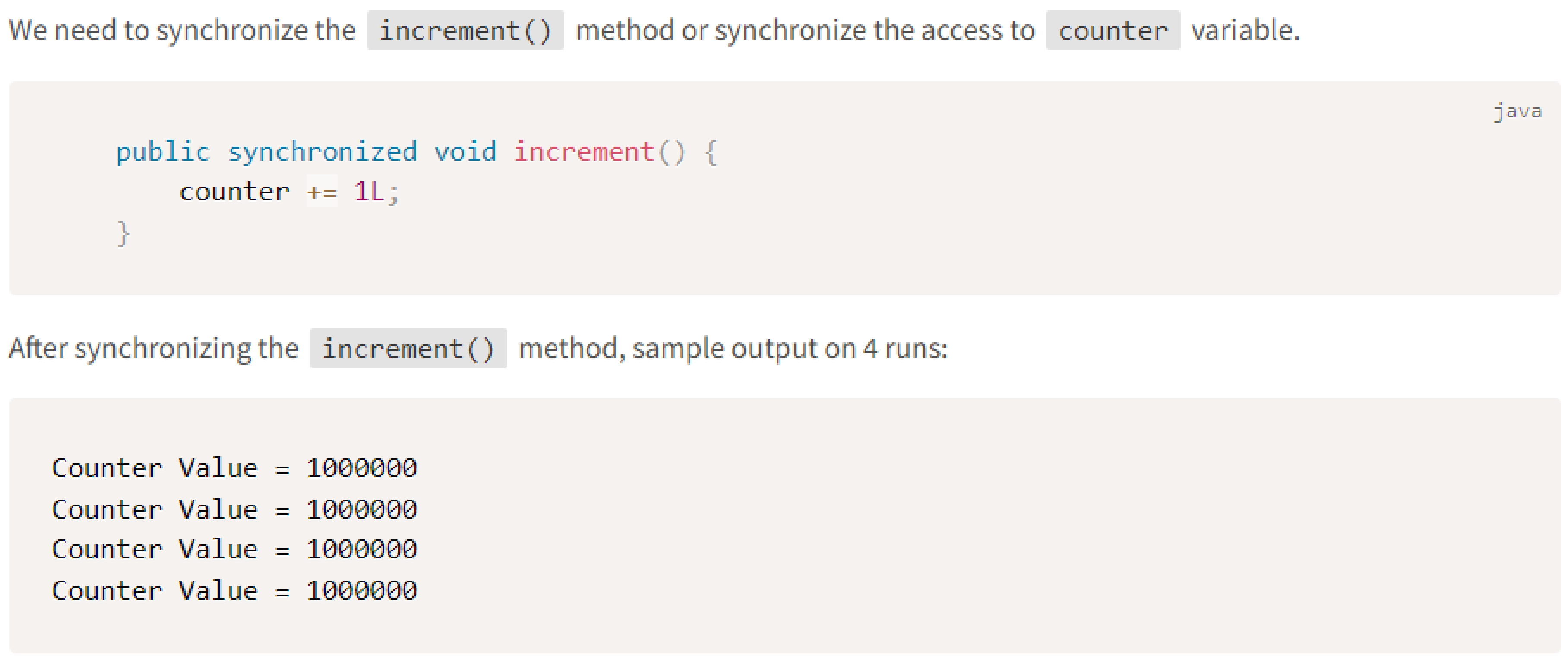
The output of the value will be different for each run.

Sample Outputs on 4 runs:

```
Counter Value = 994678
Counter Value = 994715
Counter Value = 995232
Counter Value = 980564
```

Synchronization Issue

Interview Questions #2: JP Morgan Chase - Demonstrate synchronization issue and fix the code (**Solution**)



We need to synchronize the `increment()` method or synchronize the access to `counter` variable.

```java
    public synchronized void increment() {
        counter += 1L;
    }
```

After synchronizing the `increment()` method, sample output on 4 runs:

```
Counter Value = 1000000
Counter Value = 1000000
Counter Value = 1000000
Counter Value = 1000000
```

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

Guidemy

Interview Problem #3: Goldman Sachs - Demonstrate deadlock issue and fix the code

- Write a program to demonstrate deadlock issue where a thread T1 holds a key needed by a thread T2, and T2 holds the key needed by T1. Fix the code.

Deadlock Issue

```java
import java.util.concurrent.TimeUnit;

public class DeadlockDemo {

    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(final String[] args) {
        new Thread1().start();
        new Thread2().start();
    }

    private static class Thread1 extends Thread {
        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 1: Has lock1");
                try {
                    TimeUnit.MILLISECONDS.sleep(100L);
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("Thread 1: Waiting for lock2");
                synchronized (lock2) {
                    System.out.println("Thread 1: Has lock1 and lock2");
                }
                System.out.println("Thread 1: Released lock2");
            }
            System.out.println("Thread 1: Released lock1. Exiting...");
        }
    }
```

```java
    private static class Thread2 extends Thread {
        public void run() {
            synchronized (lock2) {
                System.out.println("Thread 2: Has lock2");
                try {
                    TimeUnit.MILLISECONDS.sleep(100L);
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 2: Waiting for lock1");
                synchronized (lock1) {
                    System.out.println("Thread 2: Has lock1 and lock2");
                }
                System.out.println("Thread 2: Released lock1");
            }
            System.out.println("Thread 2: Released lock2. Exiting...");
        }
    }

}
```

`Thread1` acquires `lock1` and `Thread2` acquires `lock2`. Now both threads are waiting for other lock held by different thread causing deadlock.

Output is stuck and application keeps on running (does not finish) with deadlock between 2 threads:

```
Thread 1: Has lock1
Thread 2: Has lock2
Thread 2: Waiting for lock1
Thread 1: Waiting for lock2
```

Interview Problem #3: Goldman Sachs - Demonstrate deadlock issue and fix the code (**Solution**)

- Deadlock issue can be fixed by maintaining the **same sequence** for locks acquisition.
- Both Thread1 and Thread2 can acquire lock1 and lock2 in the **same sequence** thus avoiding the deadlock issue.

Deadlock Issue

```java
import java.util.concurrent.TimeUnit;

public class DeadlockDemo {

    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(final String[] args) {
        new Thread1().start();
        new Thread2().start();
    }

    private static class Thread1 extends Thread {
        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 1: Has lock1");
                try {
                    TimeUnit.MILLISECONDS.sleep(100L);
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("Thread 1: Waiting for lock2");
                synchronized (lock2) {
                    System.out.println("Thread 1: Has lock1 and lock2");
                }
                System.out.println("Thread 1: Released lock2");
            }
            System.out.println("Thread 1: Released lock1. Exiting...");
        }
    }
```

```java
    private static class Thread2 extends Thread {
        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 2: Has lock1");
                try {
                    TimeUnit.MILLISECONDS.sleep(100L);
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 2: Waiting for lock2");
                synchronized (lock2) {
                    System.out.println("Thread 2: Has lock1 and lock2");
                }
                System.out.println("Thread 2: Released lock2");
            }
            System.out.println("Thread 2: Released lock1. Exiting...");
        }
    }

}
```

Output is correct now with no deadlock:

```
Thread 1: Has lock1
Thread 1: Waiting for lock2
Thread 1: Has lock1 and lock2
Thread 1: Released lock2
Thread 1: Released lock1. Exiting...
Thread 2: Has lock1
Thread 2: Waiting for lock2
Thread 2: Has lock1 and lock2
Thread 2: Released lock2
Thread 2: Released lock1. Exiting...
```

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

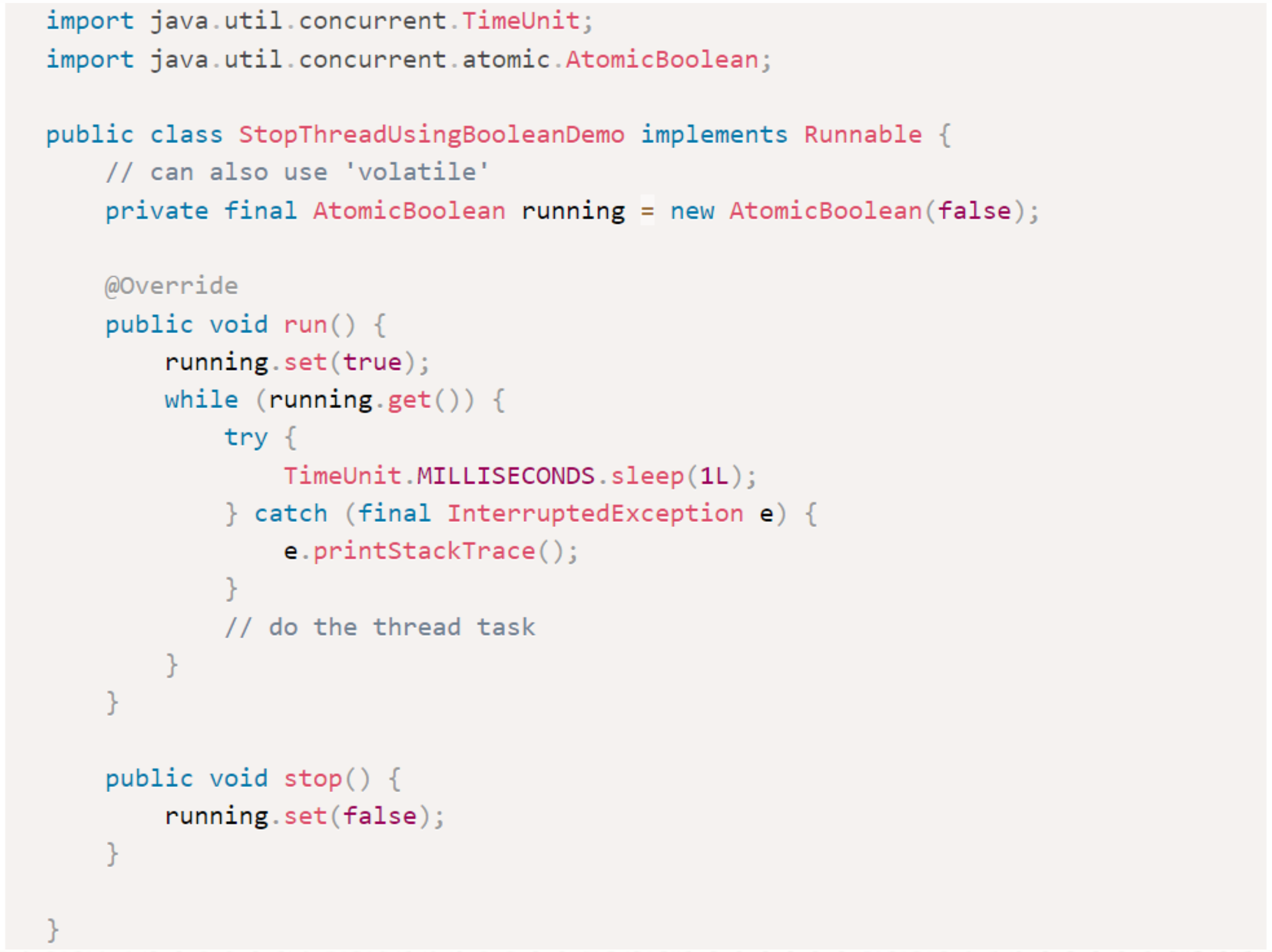
Thread States

Recap

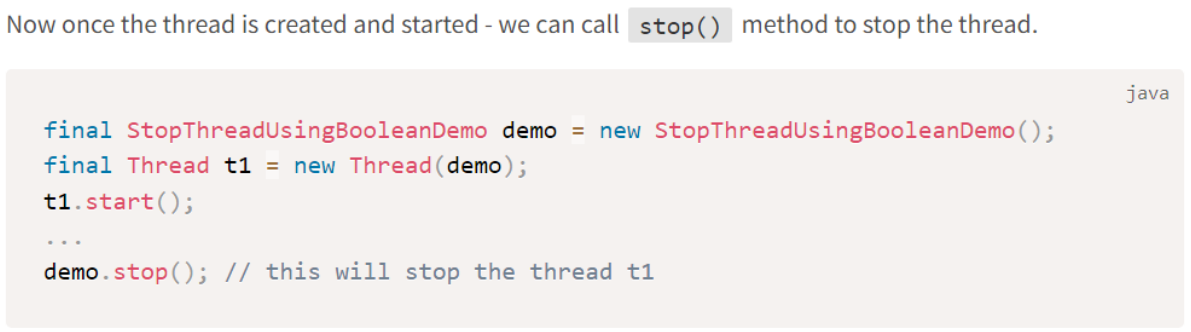
Interview Problem #4: Barclays - How to stop a thread in Java?

- We should **NOT** use **Thread.stop()** method as it is deprecated.
- **Thread.stop()** can lead to monitored objects being *corrupted*, and it is *inherently unsafe*.
- Solutions:
  1. Use a **thread-safe boolean variable** to control thread execution
  2. Call **interrupt()** on a running thread

Solution #1: Use a thread-safe boolean variable to control thread execution



```java
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;

public class StopThreadUsingBooleanDemo implements Runnable {
    // can also use 'volatile'
    private final AtomicBoolean running = new AtomicBoolean(false);

    @Override
    public void run() {
        running.set(true);
        while (running.get()) {
            try {
                TimeUnit.MILLISECONDS.sleep(1L);
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
            // do the thread task
        }
    }

    public void stop() {
        running.set(false);
    }

}
```

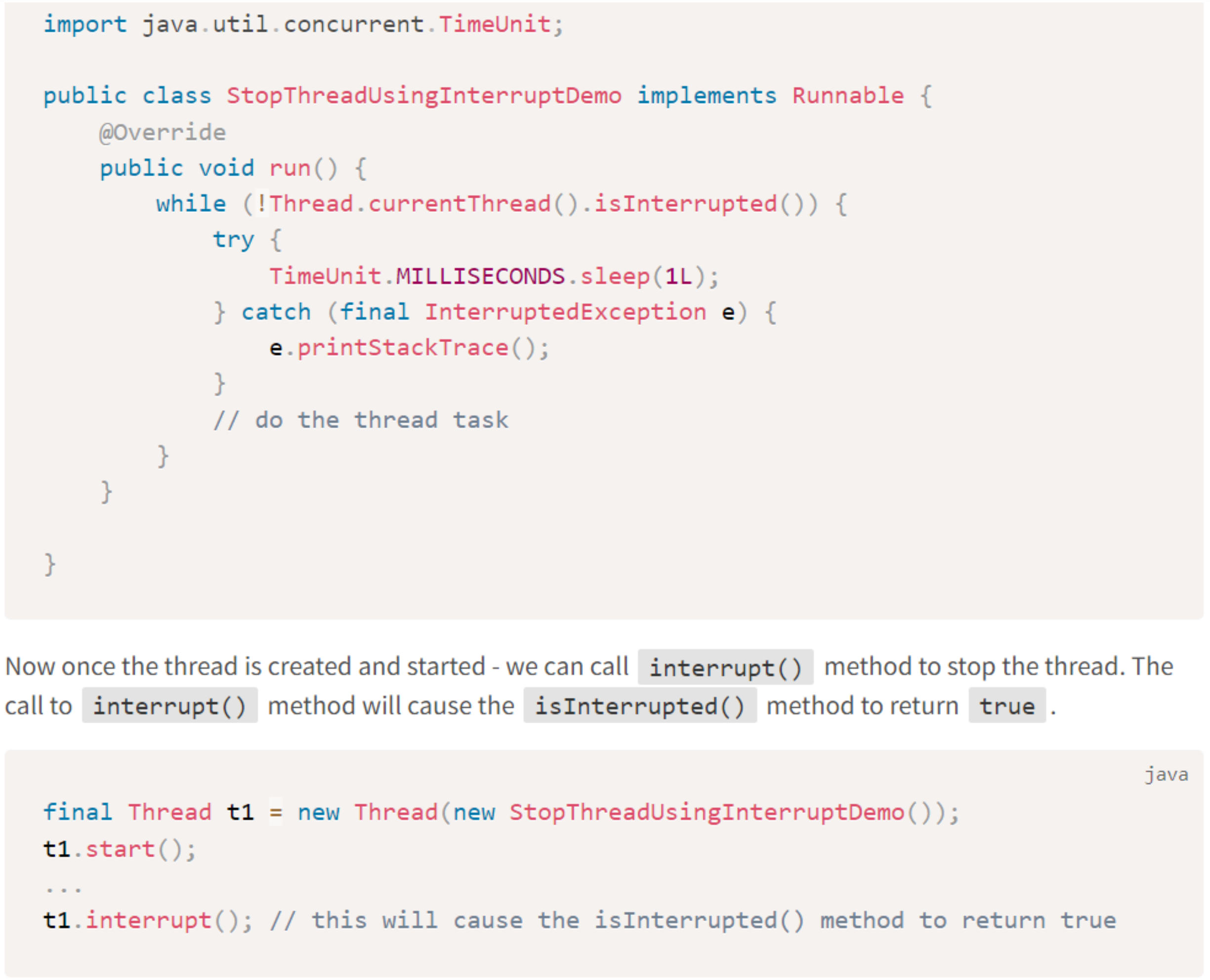


Now once the thread is created and started - we can call `stop()` method to stop the thread.

java

```java
final StopThreadUsingBooleanDemo demo = new StopThreadUsingBooleanDemo();
final Thread t1 = new Thread(demo);
t1.start();
...
demo.stop(); // this will stop the thread t1
```

Solution #2: Call interrupt() on a running thread



```java
import java.util.concurrent.TimeUnit;

public class StopThreadUsingInterruptDemo implements Runnable {
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                TimeUnit.MILLISECONDS.sleep(1L);
            } catch (final InterruptedException e) {
                e.printStackTrace();
            }
            // do the thread task
        }
    }

}
```

Now once the thread is created and started - we can call `interrupt()` method to stop the thread. The call to `interrupt()` method will cause the `isInterrupted()` method to return `true`.

```java
final Thread t1 = new Thread(new StopThreadUsingInterruptDemo());
t1.start();
...
t1.interrupt(); // this will cause the isInterrupted() method to return true
```

All the blocking methods like **wait**(), **notify**(), **notifyAll**(), **join**(), **sleep**() etc. throw **InterruptedException** based on the same **interrupted** status of thread.

The interrupt mechanism is implemented using an internal flag known as the **interrupt status**.

- Invoking non-static **Thread.interrupt**() sets this flag
- When a thread checks for an interrupt by invoking the static method **Thread.interrupted**(), interrupt status is cleared
- The non-static **Thread.isInterrupted**() method, which is used by one thread to query the interrupt status of another, does NOT change the interrupt status flag

By convention, any method that exits by throwing an **InterruptedException** clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

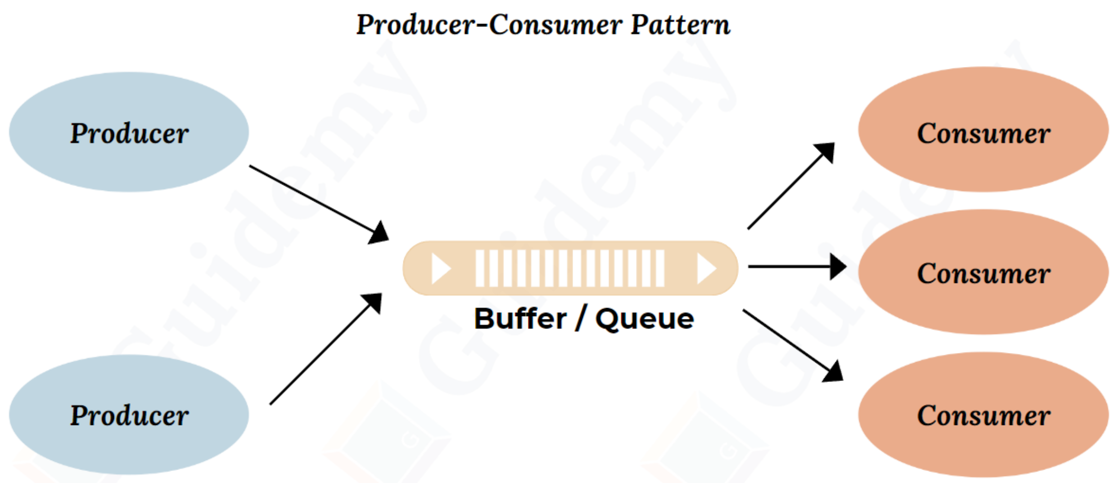
Thread States

Recap

Guidemy

Interview Problem #5: Macquarie - Explain and Implement Producer Consumer pattern

- We have a buffer - it can be an array, list, set or queue.
  - A producer produces values in a buffer.
  - A consumer consumes the values from this buffer.
  - Producers and Consumers are run in their own threads or thread pools.
- Buffer can be **bounded** (having a defined capacity) or **unbounded** (based on system memory available).
- Edge cases - The buffer can be full or empty:
  - If **full** (bounded buffer):
    - producers **cannot write** to it
  - If **empty**:
    - consumers **cannot read** from it.



Implementation #1

```java
public class ProducerDemo1<T> {
    private final T[] buffer;
    private int count = 0;

    public ProducerDemo1(final T[] buffer) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
    }

    public void produce(final T item) {
        while (isFull(buffer)) {
            // wait
        }
        buffer[count++] = item;
    }

    private boolean isFull(final T[] buffer) {
        return count == buffer.length;
    }

}
```

```java
public class ConsumerDemo1<T> {
    private final T[] buffer;
    private int count = 0;

    public ConsumerDemo1(final T[] buffer) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
    }

    public T consume() {
        while (isEmpty()) {
            // wait
        }
        return buffer[--count];
    }

    private boolean isEmpty() {
        return count == 0;
    }

}
```

Implementation #1

- **Major flaw** in this code:
  - As several threads are producing (writing) and consuming (popping) the buffer at the same time => this will result in **race condition**.
- In other words, buffer and count variables are **NOT thread-safe**.

Implementation #2: Use synchronization

```java
public synchronized void produce(final T item) {
    while (isFull(buffer)) {
        // wait
    }
    buffer[count++] = item;
}
```

```java
public synchronized T consume() {
    while (isEmpty()) {
        // wait
    }
    return buffer[--count];
}
```

- Using synchronization will help to fix the race condition problem
  - However, it will **only** synchronize all producer threads to call produce() and synchronize all consumer threads to call consume().
- Producer and Consumer threads are still **independent** of each other and not synchronized across for both produce() and consume() methods.
- We want the **common buffer** to be **thread safe** for both produce() and consume() methods.

Implementation #3: Use global lock to be used by both Producer and Consumer:

```java
public class ProducerDemo3<T> {
    private final T[] buffer;
    private final Object lock;
    private int count = 0;

    public ProducerDemo3(final T[] buffer, final Object lock) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
        this.lock = lock;
    }

    public void produce(final T item) {
        synchronized (lock) {
            while (isFull(buffer)) {
                // wait
            }
            buffer[count++] = item;
        }
    }

    private boolean isFull(final T[] buffer) {
        return count == buffer.length;
    }

}
```

```java
public class ConsumerDemo3<T> {
    private final T[] buffer;
    private final Object lock;
    private int count = 0;

    public ConsumerDemo3(final T[] buffer, final Object lock) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
        this.lock = lock;
    }

    public T consume() {
        synchronized (lock) {
            while (isEmpty()) {
                // wait
            }
            return buffer[--count];
        }
    }

    private boolean isEmpty() {
        return count == 0;
    }

}
```

Implementation #3: Use global lock to be used by both Producer and Consumer:

- Now both the buffer and Object lock are **common** to be used by both Producer and Consumer.
- However, still this design has a **major flaw**!
- Suppose if the buffer is **empty**:
  - *Consumer* threads will hold the lock object and keep on doing **busy spinning** inside: **while (isEmpty(buffer))**.
  - *Producer* threads will keep on **waiting** for this lock object held by the consumer thread **indefinitely** and never be able to produce or write anything to buffer.

Interview Problem #5: Macquarie - Explain and Implement Producer Consumer pattern **(Solution)**

- We need a mechanism to somehow “**park**” this consumer thread when the buffer is empty and release the lock.
- Then the producer thread can **acquire** this lock and write to the buffer.
- When the “parked” consumer thread is **woken up** again - the buffer will not be empty this time, and it can consume the item.
- This is the **wait() / notify() pattern**.

Interview Problem #5: Macquarie - Explain and Implement Producer Consumer pattern **(Solution)**

- The wait(), notify() and notifyAll() methods are defined in **java.lang.Object** class. These methods are invoked on a given object => normally **the object lock being used**.
  - The thread executing the invocation should hold that object key.
  - Thus, in other words, these methods **cannot** be invoked outside a synchronized block.
- Calling **wait()** releases the key (object lock) held by this thread and puts the thread in **WAIT** state. The only way to **release** a thread from a WAIT state is to **notify** it.
- Calling **notify**() release a thread in WAIT state and puts it in **RUNNABLE** state. This is the only way to release a waiting thread. The released thread is chosen **randomly**.
- For **notifyAll**(), **all** the threads are moved from WAIT state to RUNNABLE state, **however only one thread can acquire the lock again**.
  - However, the woken threads can do other task rather than waiting for the object again.

Producer-Consumer Pattern

```java
public class ProducerDemo4<T> {
    private final T[] buffer;
    private final Object lock;
    private int count = 0;

    public ProducerDemo4(final T[] buffer, final Object lock) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
        this.lock = lock;
    }

    public void produce(final T item) throws InterruptedException {
        synchronized (lock) {
            try {
                while (isFull(buffer)) {
                    lock.wait();
                }
                buffer[count++] = item;
            } finally {
                lock.notifyAll();
            }
        }
    }

    private boolean isFull(final T[] buffer) {
        return count == buffer.length;
    }

}
```

```java
public class ConsumerDemo4<T> {
    private final T[] buffer;
    private final Object lock;
    private int count = 0;

    public ConsumerDemo4(final T[] buffer, final Object lock) {
        if (buffer == null || buffer.length == 0) {
            throw new IllegalArgumentException();
        }
        this.buffer = buffer;
        this.lock = lock;
    }

    public T consume() throws InterruptedException {
        synchronized (lock) {
            try {
                while (isEmpty()) {
                    lock.wait();
                }
                return buffer[--count];
            } finally {
                lock.notifyAll();
            }
        }
    }

    private boolean isEmpty() {
        return count == 0;
    }

}
```

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

Guidemy

Thread States

- A thread has a **state** - for example, it can be running or not.
- We can get the thread state by calling **getState**() method on thread.

```
final Thread t1 = new Thread();
...
Thread.State state = t1.getState();
```

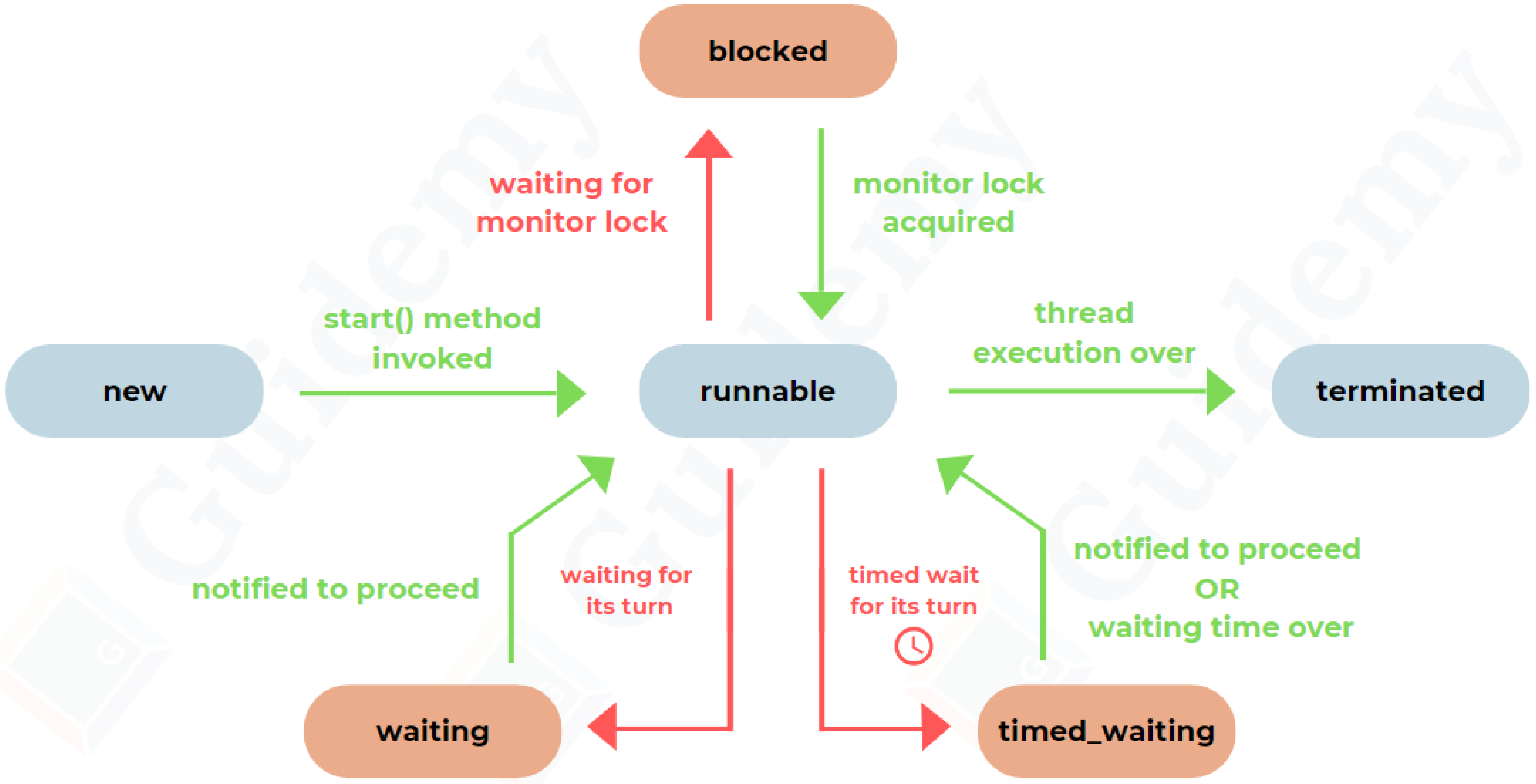
Java API already defines **enum** `Thread.State` as follows:

`public static enum Thread.State extends Enum<Thread.State>`

Thread States

- A thread can be in one of the following states:
  - **NEW**: A thread that has not yet started is in this state.
  - **RUNNABLE**: A thread executing in the Java virtual machine is in this state.
  - **BLOCKED**: A thread that is blocked waiting for a monitor lock is in this state.
  - **WAITING**: A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
  - **TIMED_WAITING**: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
  - **TERMINATED**: A thread that has exited is in this state.

Thread States



Thread States

- A thread can be in **only one state at a given point in time**.
- These states are **virtual machine states** which do **not** reflect any operating system thread states.
- NOTE: If a thread is not running, can it be given hand by the thread scheduler ?
  - Answer is **NO** => thread scheduler will only schedule threads which are in **RUNNABLE** state.

Threading Fundamentals

Race Condition

Synchronization Issue

Deadlock Issue

Stopping Threads

Producer-Consumer Pattern

Thread States

Recap

Summary

- In computer science, **concurrency** is the execution of the multiple instruction sequences at the same time.
- **CPU Scheduling** is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I/O etc., thus making full use of the CPU.
- A Java thread is a light-weight **subprocess** within the smallest unit of **processes** and having separate paths of execution.
- A **race condition**, in the simplest terms, is when two or more actions **compete to complete first**, causing **unpredictable** results.
- Race condition can be resolved using **synchronization**.

Summary

- Deadlock is where a thread T1 holds a key needed by a thread T2, and T2 holds the key needed by T1. Both threads are waiting for other lock held by different thread causing deadlock.
- Deadlock issue can be fixed by maintaining the **same sequence** for locks acquisition.
- A thread can be stopped using a **thread-safe boolean variable** or calling **interrupt()**.
- A **Producer-Consumer Pattern** is when:
  - We have a **buffer** - it can be an array, list, set or queue.
  - A producer **produces** values in a buffer.
  - A consumer **consumes** the values from this buffer.
  - Producers and Consumers are **run in their own threads or thread pools**.
- A Producer-Consumer Pattern can be implemented using a **wait-notify pattern**.

What’s Next?

- This is just about the introduction to concurrency we have covered in this free trial session.
- In the real course - we will cover additional **advanced topics** on concurrency:
  - Thread pools
  - Concurrent Collections
  - Java Memory Model
  - Advanced Locking
  - Lock-Free Algorithms, Data Structures and Techniques
  - Fork-Join Framework

What’s Next?

- Take your programming skills to the next level, secure sought-after senior Java developer roles, and skyrocket your earnings - all with Guidemy's **Advanced Software Engineering in Java** course!
- Our course teaches you advanced skills that make you stand out when you're looking for a job. Start learning with Guidemy today and watch your skills and salary grow!
- Visit https://guidemy.io/en to learn more