QC-3 :MultiThreading

look i want to demonstate stringbuilder is not threadsafe so just write normal code for it

To demonstrate that `StringBuilder` is not thread-safe, you can run multiple threads that concurrently modify the same `StringBuilder` object without synchronization. This can lead to inconsistent results due to race conditions.

Here's a simplified version of the code:

```
javaCopy code
package com.revature.StringDemo;

public class StringBuilderDemo extends Thread {
    private static StringBuilder str = new StringBuilder();

    public void run() {
        for (int i = 0; i < 1000; i++) {
            str.append("a");
        }
    }

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

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

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }

        System.out.println("Length of StringBuilder: " + str.
length());
    }
}
```

Explanation:

- **No Synchronization**: The `StringBuilder` object is accessed and modified by multiple threads without any synchronization, which can lead to race conditions.
- **Concurrent Modification**: Since `StringBuilder` is not thread-safe, running this code may result in a `StringBuilder` length that is less than the expected value (2000), because multiple threads could overwrite each other's modifications.

When you run this code, the output may vary between executions, and it's possible that the final length will be less than 2000, demonstrating that `StringBuilder` is not thread-safe.

4o

Q.what is join here and what will happen if i dont write it?

The `join()` method in Java is used to ensure that one thread waits for another thread to finish its execution before it continues. When you call `t1.join()` or `t2.join()`, the main thread (the one executing the `main` method) pauses its execution and waits until `t1` and `t2` complete their execution.

Purpose of `join()`:

- **Thread Synchronization**: `join()` is used to make sure that the main thread waits for the completion of other threads before proceeding. This is crucial when you want to make sure that the results of all threads are fully computed before performing an action, such as printing the final result.

What happens if you don't use `join()`:

- If you don't call `join()`, the main thread may continue executing without waiting for the other threads (`t1` and `t2`) to finish. This means the main thread might reach the point where it prints the length of the `StringBuilder` before the other threads have completed their appending operations.
- As a result, the output might not reflect the work done by all threads, leading to incorrect or partial results. For example, the length of `StringBuilder` might be less than expected because the main thread printed the length before the other threads finished their work.

Q: What is multithreading, and how does it improve the performance of a Java application?

Answer:

Multithreading is a programming technique that allows multiple threads to run concurrently within a single process. Each thread represents a separate path of execution, meaning that a multithreaded program can perform multiple tasks simultaneously. This can lead to better utilization of CPU resources, especially on multi-core processors, where different threads can be executed on different cores in parallel.

How it improves performance:

- **Concurrency:** Multithreading enables tasks to run concurrently, allowing more efficient use of CPU time. For instance, while one thread is waiting for I/O operations (like reading from a file), another thread can execute computations.
- **Responsiveness:** In a GUI application, multithreading can help maintain responsiveness. For example, the UI can remain responsive to user input while another thread handles time-consuming tasks in the background.
- **Throughput:** Multithreading can increase the throughput of a program by performing multiple operations at the same time, thereby speeding up the overall execution.

Real-life Example:

Imagine a restaurant kitchen where multiple chefs are working simultaneously on different parts of a meal—one chef is preparing the salad, another is grilling the meat, and a third is baking the dessert. This parallel effort allows the meal to be prepared faster than if a single chef were doing all the tasks sequentially.

Technical Example:

In a web server, each incoming client request can be handled by a separate thread. While one thread processes the request for serving a web page, another thread can handle a database query, and yet another can serve static content, all at the same time. This leads to faster response times and better handling of multiple client requests.

Q: Explain how threads work in Java and how the OS manages them.

Answer:

In Java, a thread is a lightweight sub-process, the smallest unit of processing. Threads in Java are instances of the `Thread` class or classes that implement the `Runnable` interface. When you create and start a thread in Java, the JVM maps the thread to an OS-level thread managed by the operating system.

OS Management:

- **Scheduling:** The OS is responsible for scheduling threads. It decides which thread runs at any given time based on its priority and the scheduling algorithm in use (like Round Robin, Priority Scheduling, etc.).
- **Context Switching:** The OS handles context switching between threads, saving and restoring the state of threads so that execution can resume from where it left off.
- **Resource Allocation:** The OS allocates resources like CPU time and memory to each thread. It also ensures that no thread starves or gets stuck without resources.

Real-life Example:

Think of a factory where different machines (representing threads) are managed by a supervisor (the OS). The supervisor decides which machine works on which

task, when to pause a machine, and when to start another, ensuring that all machines are efficiently utilized.

Technical Example:

In Java, when you call `Thread.start()`, the JVM requests the OS to create a new native thread, and the OS scheduler will manage it. The OS determines which thread gets CPU time based on factors like thread priority and system load.

Q: What are the potential issues with multithreading?

Answer:

While multithreading can improve performance, it introduces several challenges:

1. **Race Conditions:(DATA INCONSISTENCY):** Occur when two or more threads try to access shared data at the same time, leading to inconsistent or incorrect outcomes.
2. **Deadlocks:** Happen when two or more threads are waiting for each other to release resources, causing all of them to be stuck indefinitely.
3. **Thread Interference:** When one thread modifies shared data while another thread is in the middle of reading or writing it, leading to unpredictable results.
4. **Increased Complexity:** Multithreaded programs are harder to design, test, and debug due to the non-deterministic nature of thread execution.
5. **Resource Overhead:** Creating and managing threads consume system resources. Too many threads can lead to resource exhaustion and reduce performance due to excessive context switching.

Real-life Example:

Imagine two people trying to write on the same whiteboard simultaneously. If they don’t coordinate, they might overwrite each other's work (race condition). If each person is waiting for the other to finish writing before they start, and neither starts writing, it leads to a deadlock.

Technical Example:

Consider a banking application where two threads simultaneously try to withdraw money from the same account. If not properly synchronized, both threads might

read the same balance and allow withdrawals that exceed the actual balance, leading to inconsistencies.

Q: Describe the lifecycle of a thread in Java.

Answer:

The lifecycle of a thread in Java consists of several states:

1. **New:** The thread is created but not yet started. It remains in this state until the `start()` method is called.
2. **Runnable:** After calling `start()`, the thread moves to the runnable state. It is ready to run and is waiting for CPU time. It could be running or waiting to run, depending on the OS's thread scheduler.
3. **Blocked:** A thread enters this state when it is waiting for a monitor lock (e.g., trying to enter a synchronized block that is already locked by another thread).
4. **Waiting:** A thread is in this state when it is waiting indefinitely for another thread to perform a particular action (e.g., calling `Object.wait()`).
5. **Timed Waiting:** This state is similar to the waiting state, but the thread is waiting for a specified amount of time (e.g., `Thread.sleep()` or `Object.wait(timeout)`).
6. **Terminated:** The thread has finished execution and exits the lifecycle. This happens when the `run()` method completes or when the thread is stopped forcibly.

Real-life Example:

Consider a relay race. The runner (thread) is in the "New" state when they're on standby. Once they start running, they enter the "Runnable" state. If they reach a checkpoint where they need to wait for another runner to pass the baton, they are in the "Waiting" state. If they are told to wait for a specific amount of time before starting again, they are in the "Timed Waiting" state. Once they cross the finish line, they are in the "Terminated" state.

Technical Example:

In a web server, a thread handling a client request may be in the “Runnable” state while processing the request, enter the “Blocked” state if it needs to access a synchronized resource, and finally move to the “Terminated” state after the request has been processed and the response has been sent.

Q: What happens when a thread is in the `NEW` state?

Answer:

When a thread is in the `NEW` state, it has been created but has not yet started its execution. The thread remains in this state until the `start()` method is called. In this state, the thread is merely an object and does not consume any CPU resources.

Real-life Example:

Imagine a car that is assembled and ready to drive but has not yet been started. The car is in the “NEW” state, waiting for the ignition to be turned on.

Technical Example:

In Java, `Thread t = new Thread();` creates a thread in the `NEW` state. It won't begin execution until `t.start()` is called.

Q.Can we call thread.start() method twice?

NO-Will get illegalThreadStateException

Q: What is the difference between `WAITING` and `TIMED_WAITING`?

Answer:

- `WAITING`**:** In this state, a thread is waiting indefinitely for another thread to perform a specific action. The thread will stay in this state until it is explicitly woken up by another thread (using `notify()` or `notifyAll()`).
- `TIMED_WAITING`**:** This state is similar to `WAITING`, but with a time limit. The thread remains in the `TIMED_WAITING` state until either the specified time elapses or it is woken up by another thread.

Real-life Example:

- **WAITING:** A person waiting in a queue without knowing when their turn will come (indefinite waiting).
- **TIMED_WAITING:** A person setting an alarm to wake up after 30 minutes (waiting for a specific time).

Technical Example:

- **WAITING:** A thread that calls `Object.wait()` will enter the `WAITING` state until another thread calls `notify()` on the same object.
- **TIMED_WAITING:** A thread that calls `Thread.sleep(1000)` will enter the `TIMED_WAITING` state and wake up after 1000 milliseconds.

Q: How do thread priorities work in Java?

Answer:

Thread priorities in Java are used to suggest the relative importance of threads to the thread scheduler. Each thread has a priority, which is an integer value between `Thread.MIN_PRIORITY` (1) and `Thread.MAX_PRIORITY` (10). The default priority is `Thread.NORM_PRIORITY` (5). The thread scheduler uses these priorities to determine the order in which threads are executed. Threads with higher priority are more likely to be executed before threads with lower priority.

However, it's important to note that thread priorities are just hints to the scheduler. The actual scheduling behavior is platform-dependent, and the JVM does not guarantee that higher-priority threads will always run before lower-priority ones.

Real-life Example:

Imagine a customer support center where requests are prioritized. Urgent requests (high priority) are handled first, while less urgent ones (low priority) are handled later. However, if all the representatives are busy with urgent requests, a low-priority request might have to wait longer.

Technical Example:

In Java, you can set a thread's priority using the `setPriority(int newPriority)` method. For example:

```
javaCopy code
Thread t1 = new Thread();
t1.setPriority(Thread.MAX_PRIORITY);
```

This sets `t1` to have the highest priority. However, if the system is heavily loaded, even this thread might not get immediate CPU time.

Q. How to Change Priority of main Thread:

Thread.currentThread().setPriority(Thread.MAX_PRIORITY);

Q: What is the default priority of a thread?

Answer:

The default priority of a thread in Java is `Thread.NORM_PRIORITY`, which is typically set to `5`. This is the middle value in the priority range (1 to 10).

Real-life Example:

Consider an office where employees work on tasks of medium importance by default unless specified otherwise. This reflects the default priority assigned to threads.

Technical Example:

If you create a thread in Java without explicitly setting its priority:

```
javaCopy code
Thread t = new Thread();
```

The thread `t` will have a priority of `5` (NORM_PRIORITY) by default.

Q: Can thread priorities guarantee the order of execution? Why or why not?

Answer:

No, thread priorities do not guarantee the order of execution. While higher-priority threads are generally favored by the thread scheduler, the actual execution order is not guaranteed and is dependent on the JVM implementation and the underlying operating system's thread scheduler.

Reasons:

- **Platform Dependency:** Thread scheduling is platform-dependent, and different operating systems may handle priorities differently.
- **Time-Slicing:** Even if a thread has a higher priority, it might not run immediately if the CPU is currently occupied with another thread.
- **Non-Deterministic Scheduling:** The JVM does not enforce strict priority-based scheduling, and factors like I/O operations, interrupts, and system load can affect the order of execution.

Real-life Example:

Imagine a road with multiple lanes where emergency vehicles (high-priority threads) are expected to pass first. However, due to traffic congestion or other unpredictable factors, even high-priority vehicles might get delayed.

Technical Example:

In a multithreaded Java program, even if you set a thread's priority to `MAX_PRIORITY`, there is no guarantee it will run before threads with lower priorities, especially if the system is under heavy load or if other threads are already running.

Q: What are the two ways to create a thread in Java?

Answer:

In Java, you can create a thread in two primary ways:

1. **By Extending the `Thread` Class:**
    - Create a new class that extends the `Thread` class.

- Override the `run()` method with the code you want the thread to execute.
- Create an instance of this class and call the `start()` method to begin execution.

```
javaCopy code
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
    }
}
MyThread t1 = new MyThread();
t1.start();
```

2. **By Implementing the `Runnable` Interface:**
   - Create a class that implements the `Runnable` interface.
   - Implement the `run()` method with the code you want the thread to execute.
   - Pass an instance of this class to a `Thread` object and call the `start()` method.

```
javaCopy code
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running.");
    }
}
Thread t1 = new Thread(new MyRunnable());
t1.start();
```

Q: Compare the two methods of creating a thread: implementing `Runnable` vs. extending `Thread`.

Answer:

- **Flexibility:**
  - **Runnable:** Allows you to implement other interfaces or extend other classes since Java supports single inheritance. You can reuse the same `Runnable` instance across multiple threads.
  - **Thread:** Limits your class to extending only `Thread`, restricting inheritance of other classes.
- **Code Reusability:**
  - **Runnable:** Better for scenarios where the task can be reused by different threads. It separates the task (behavior) from the thread (execution).
  - **Thread:** Less reusable as the task and thread are tightly coupled.
- **Memory and Performance:**
  - **Runnable:** Lighter in terms of memory usage since you don't have to extend a whole new class.
  - **Thread:** Slightly heavier as it creates a new instance of `Thread`.
- **Scalability:**
  - **Runnable:** More scalable for large applications because it allows better separation of concerns.
  - **Thread:** Suitable for simpler applications but becomes less manageable in complex systems.

Real-life Example:

- **Runnable:** Hiring a chef (Runnable) to work in multiple kitchens (Thread objects).
- **Thread:** Building a new kitchen (Thread) for every chef.

Technical Example:

In a game engine, you might have a `GameLogic` class that implements `Runnable`, which can be used to create threads for different parts of the game. If you extended `Thread`, you'd need to write additional code to manage different game logic threads, leading to more complex and less maintainable code.

Q: Why might you prefer implementing `Runnable` over extending the `Thread` class?

Answer:

Implementing `Runnable` is often preferred over extending `Thread` for several reasons:

1. **Multiple Inheritance:** Java does not support multiple inheritance of classes, but a class can implement multiple interfaces. By implementing `Runnable`, your class remains free to extend another class if needed.
2. **Separation of Concerns:** Implementing `Runnable` promotes better design by separating the task to be performed (the `run()` method) from the thread itself. This makes the code more modular and easier to maintain.
3. **Reusability:** A `Runnable` object can be passed to multiple `Thread` instances, allowing the same task to be executed by different threads, which is not possible when extending `Thread`.
4. **Resource Efficiency:** Implementing `Runnable` is more resource-efficient because it doesn't require creating a new instance of `Thread` class with its overhead.

Real-life Example:

Consider a freelancer (Runnable) who can work for different companies (Thread instances). If the freelancer had to become a full-time employee (extending Thread) at each company, they wouldn't be able to work elsewhere, limiting their flexibility.

Technical Example:

In a server application, you might have a `Task` class that implements `Runnable`. Different tasks can be executed by passing the `Task` object to different `Thread` objects, making the system more flexible and easier to manage.

5. Runnable Interface vs Thread Class

Q: What are the advantages of implementing the `Runnable` interface over extending the `Thread` class?

Answer:

The advantages of implementing the `Runnable` interface over extending the `Thread` class include:

1. **Multiple Inheritance Flexibility:** By implementing `Runnable`, a class can still extend another class, which is crucial if your class needs to inherit behavior from a superclass.
2. **Decoupling Task and Execution:** Implementing `Runnable` separates the task from the thread, allowing the same task to be executed by different threads, improving reusability and modularity.
3. **Resource Efficiency:** Using `Runnable` can reduce overhead, as it avoids creating a new thread object for each task. Multiple threads can share the same `Runnable` instance, saving memory and improving performance.
4. **Better Design Practices:** Implementing `Runnable` adheres to better design principles like Single Responsibility Principle, where a class has only one reason to change.

Real-life Example:

Hiring a contractor (Runnable) who can work for different companies (Thread instances) versus having to employ a new worker for each company task (extending Thread).

Technical Example:

Consider a scenario where you have a `DataProcessor` class that implements `Runnable`. You can create multiple threads to process different parts of data concurrently. If you extended `Thread`, each data processing thread would require its own unique subclass, making the codebase harder to manage.

Q: Explain the purpose of the `start()` method in a thread.

Answer:

The `start()` method is used to begin the execution of a new thread in Java. When `start()` is called on a `Thread` object, the Java Virtual Machine (JVM) creates a new thread of execution. The thread scheduler assigns the thread a unique identifier,

allocates resources, and invokes the thread's `run()` method. Unlike calling `run()` directly, `start()` truly spawns a separate, concurrent thread.

Real-life Example:

Think of `start()` as pressing the "on" button on a machine. The machine (thread) begins working independently, performing its tasks while you can continue doing something else.

Technical Example:

```
javaCopy code
Thread t = new Thread(() -> System.out.println("Thread is run
ning"));
t.start(); // Starts the thread, allowing it to run concurren
tly
```

Q: What happens if you call the `run()` method directly instead of `start()`?

Answer:

If you call the `run()` method directly instead of `start()`, the code within `run()` will execute in the current thread rather than in a new thread. This means no new thread is created, and the `run()` method behaves like a normal method call, running synchronously in the existing thread.

Q: How does the `join()` method work?

Answer:

The `join()` method in Java allows one thread to wait for the completion of another thread. When you call `join()` on a thread, the current thread pauses its execution until the thread on which `join()` was called has finished executing. This is useful when you need to ensure that a particular task is completed before proceeding further in the program.

Real-life Example:

Consider waiting for your friend to finish their meal before you both leave a restaurant together. You won't move until they are done (thread waits for another

thread to finish).

Q: Describe the difference between `sleep()` and `yield()`.

Answer:

- `sleep()` **Method:**
  - **Purpose:** Causes the current thread to pause its execution for a specified period, allowing other threads to execute.
  - **Behavior:** The thread remains in the "TIMED_WAITING" state for the duration of the sleep and resumes execution afterward. Other threads may run during this time, but there is no guarantee.
  - **Example:** If a thread is processing data and needs to wait for an I/O operation, it might call `sleep()` to avoid busy-waiting.
- `yield()` **Method:**
  - **Purpose:** Suggests to the thread scheduler that the current thread is willing to yield its execution time, allowing other threads of the same or higher priority to run.
  - **Behavior:** The thread moves back to the "RUNNABLE" state, and the thread scheduler decides whether to let it continue or allow another thread to run. This is more of a hint than a command.
  - **Example:** If a thread is performing a less urgent task and wants to give a chance to other threads to execute, it might call `yield()`.

Real-life Example:

- `sleep()`**:** Taking a nap for a set time.
- `yield()`**:** Stepping aside in a queue to let someone else go ahead.

Q: What is the use of the `setDaemon()` method?

Answer:

The `setDaemon(boolean on)` method is used to mark a thread as a daemon thread. Daemon threads are background threads that do not prevent the JVM from exiting when the program finishes. They are typically used for tasks like garbage collection, monitoring, or other background services that should not block the program's termination.

By default, threads are non-daemon. To make a thread a daemon, you must call `setDaemon(true)` before starting the thread.

Real-life Example:

Consider a janitor who cleans up after everyone has left the building. The janitor's work (daemon thread) doesn't stop the building from closing (program termination).

Q: Explain the `isAlive()` method. How can it be used in thread management?

Answer:

The `isAlive()` method checks if a thread is still active. A thread is considered alive if it has been started and has not yet finished executing. This method returns `true` if the thread is still running or in the process of starting, and `false` if it has completed its execution.

Usage in Thread Management:

- **Monitoring:** `isAlive()` is useful for monitoring the status of threads. For example, before performing a certain operation that depends on the completion of a thread, you might check if that thread is still alive.
- **Debugging:** It helps in debugging by allowing you to determine if a thread has unexpectedly terminated.

Real-life Example:

Checking if your car engine is still running (thread is alive) before attempting to drive.

Technical Example:

```
javaCopy code
Thread t = new Thread(() -> {
```

```
    for(int i = 0; i < 5; i++) {
        System.out.println("Thread is running");
    }
});
t.start();
if (t.isAlive()) {
    System.out.println("Thread is still running");
} else {
    System.out.println("Thread has finished execution");
}
```

Q: What is synchronization in Java, and why is it important in multithreading?

Answer:

Synchronization in Java is a mechanism that ensures that only one thread can access a resource or execute a block of code at a time. It is crucial in multithreading to prevent race conditions, where multiple threads modify shared resources simultaneously, leading to inconsistent results.

Importance:

- **Data Consistency:** Synchronization ensures that shared resources are accessed in a controlled manner, maintaining data consistency.
- **Thread Safety:** It prevents threads from interfering with each other, ensuring that critical sections of code are executed by only one thread at a time.

Real-life Example:

Imagine multiple people trying to withdraw money from the same bank account simultaneously. Synchronization is like a lock on the account that ensures only one person can withdraw money at a time.

Technical Example:

Consider a situation where multiple threads are trying to increment a shared counter:

```
javaCopy code
public class Counter {
    private int count = 0;

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

    public int getCount() {
        return count;
    }
}
```

Q: How does the `synchronized` keyword work in Java?

Answer:

The `synchronized` keyword in Java is used to create synchronized methods or blocks. When a method or block is synchronized, a lock (also known as a monitor) is associated with the object or class that owns the code. Before a thread can execute the synchronized code, it must acquire the lock. If another thread has already acquired the lock, the current thread must wait until the lock is released.

Real-life Example:

Think of a synchronized method as a restroom in a public place. If someone is using it, others have to wait until it's free (lock is released).

Q: Explain the concept of a monitor in the context of synchronization.

Answer:

A monitor is a synchronization construct that allows threads to have mutual exclusive access to critical sections of code. In Java, every object has an implicit monitor associated with it. When a thread enters a synchronized method or block, it acquires the monitor (lock) of the object. Other threads trying to access synchronized code on the same object must wait until the monitor is released.

Key Points:

- Only one thread can hold a monitor for a particular object at any time.
- The monitor is automatically released when the thread exits the synchronized code, either normally or by throwing an exception.

Real-life Example:

Think of a monitor as a turnstile at a stadium entrance. Only one person (thread) can pass through at a time, and others must wait their turn.

Q: What is the difference between synchronized methods and synchronized blocks?

Answer:

- **Synchronized Methods:**
  - **Scope:** The entire method is synchronized, meaning that the thread must acquire the lock for the object before it can execute any part of the method.
  - **Lock Level:** Locks the monitor of the object (or class for static methods) for the entire method duration.
  - **Use Case:** Simple cases where the whole method needs to be synchronized.
- **Synchronized Blocks:**
  - **Scope:** Only a specific block of code within a method is synchronized. The rest of the method can execute concurrently.
  - **Lock Level:** Allows finer control over which part of the method is synchronized and on which object's monitor to lock.
  - **Use Case:** When you want to reduce the scope of synchronization to improve performance, allowing some parts of the method to execute concurrently.

Real-life Example:

A synchronized method is like locking a room whenever you enter or leave, while a synchronized block is like locking only the cabinet inside the room when you need to use it.

Example Scenario

Imagine a banking system where multiple threads are accessing a shared `BankAccount` object to perform deposit and withdrawal operations. Let's see when to use a synchronized method and when to use a synchronized block.

Example 1: Synchronized Method

In this example, we'll use a synchronized method to ensure that only one thread can access the `deposit` and `withdraw` methods at a time.

```
javaCopy code
class BankAccount {
    private int balance = 0;

    // Synchronized method
    public synchronized void deposit(int amount) {
        balance += amount;
    }

    // Synchronized method
    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds");
        }
    }

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

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

        // Multiple threads trying to access the account
        Thread t1 = new Thread(() -> account.deposit(100));
        Thread t2 = new Thread(() -> account.withdraw(50));

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

When to Use Synchronized Method:

- Use when all operations within the method should be executed by only one thread at a time.
- This is simple and ensures thread safety for the entire method.

Example 2: Synchronized Block

Now, let's assume we want to add some logging to the `deposit` and `withdraw` methods. However, we only want to synchronize the part that modifies the balance, not the logging.

```
javaCopy code
class BankAccount {
    private int balance = 0;

    public void deposit(int amount) {
        // Non-critical section
        System.out.println(Thread.currentThread().getName() +
" is depositing: " + amount);
```

```
        // Synchronized block
        synchronized (this) {
            balance += amount;
        }

        // Non-critical section
        System.out.println(Thread.currentThread().getName() +
" completed deposit.");
    }

    public void withdraw(int amount) {
        // Non-critical section
        System.out.println(Thread.currentThread().getName() +
" is withdrawing: " + amount);

        // Synchronized block
        synchronized (this) {
            if (balance >= amount) {
                balance -= amount;
            } else {
                System.out.println("Insufficient funds");
            }
        }

        // Non-critical section
        System.out.println(Thread.currentThread().getName() +
" completed withdrawal.");
    }

    public int getBalance() {
        return balance;
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        BankAccount account = new BankAccount();

        // Multiple threads trying to access the account
        Thread t1 = new Thread(() -> account.deposit(100));
        Thread t2 = new Thread(() -> account.withdraw(50));

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

When to Use Synchronized Block:

- Use when you only need to synchronize a part of the method, allowing other threads to execute non-critical sections concurrently.
- This improves performance by reducing the amount of code that is locked.

Q: How can you avoid race conditions in a multithreaded program?

-Sync block

-Sync Methods

-Sync Datastructure

-Collections.syncset/list/map

CODES-

1. Creating a Thread Using Both Runnable Interface and Thread Class

Using Runnable Interface:

```
javaCopy code
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Runnable: " + i);
        }
    }
}

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

Using Thread Class:

```
javaCopy code
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Thread: " + i);
        }
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
```

```
    }
}
```

Q: What is the difference between `notify()` and `notifyAll()` in Java?

Answer:

Both `notify()` and `notifyAll()` are used in the context of thread communication. They are methods provided by the `Object` class to manage the way threads are awakened from the `WAITING` state when they are waiting on the object's monitor.

- `notify()`:
  - **Behavior:** Wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, one of them is chosen at random (the choice is not predictable).
  - **Use Case:** Useful when you know that only one thread needs to proceed after a certain condition is met.
- `notifyAll()`:
  - **Behavior:** Wakes up all the threads that are waiting on the object's monitor. However, only one of them will proceed once it successfully acquires the lock, and the others will return to the `WAITING` state.
  - **Use Case:** Useful when you want to notify all waiting threads to check if they can proceed. This is often used in situations where multiple threads may need to act based on a state change.

Real-life Example:

- `notify()`: Imagine you're in a conference room with multiple people (threads) waiting for their turn to speak (acquire the monitor). The speaker (the thread that holds the monitor) points to one person and says, "You're next," allowing only that person to speak.
- `notifyAll()`: The speaker announces to everyone, "You can all speak now," but only one person can start talking at a time (one thread acquires the monitor first), while the others must wait until they get their chance.

Livelock

Definition:

Livelock is a situation in which two or more threads continually change states in response to each other, but none of them make progress. Unlike deadlock, where threads are stuck waiting for each other to release resources, livelock involves threads actively changing their state but still failing to progress.

Example:

Imagine two people trying to pass each other in a narrow hallway. Each person continuously steps to the side to let the other pass, but because they are both moving to the side at the same time, they never actually manage to pass each other.

Producer-Consumer Problem

Definition:

The producer-consumer problem is a classic synchronization problem where producers generate data and put it into a shared buffer, while consumers take data from the buffer. The challenge is to ensure that the buffer does not overflow (when it’s full) or underflow (when it’s empty), and to coordinate access to the buffer so that the producer and consumer operate efficiently without stepping on each other’s toes.

Real-life Example:

Consider a kitchen with a chef (producer) and a waiter (consumer). The chef prepares dishes and places them on a counter (buffer). The waiter picks up the dishes from the counter and serves them to customers. If the counter is full, the chef must wait until the waiter takes some dishes away. Conversely, if the counter is empty, the waiter must wait until the chef prepares more dishes