Synchronization

Synchronization in threads is a critical concept in concurrent programming that ensures that multiple threads can safely access shared resources without causing data corruption or inconsistency. When threads are executing concurrently, they might interact with shared data, and synchronization helps to coordinate these interactions.

Here's a basic explanation and examples to help a beginner understand synchronization in Java.

What is Synchronization?

Synchronization is the process of controlling access to shared resources by multiple threads to prevent data inconsistency. Without synchronization, threads might interfere with each other, leading to unpredictable results.

Basic Concepts

1. **Shared Resource:** A resource (like a variable or an object) that is accessed by multiple threads.
2. **Critical Section:** A part of the code where the shared resource is accessed or modified. Only one thread should be allowed to execute in the critical section at a time.
3. **Synchronized Keyword:** A keyword in Java that ensures that only one thread can execute a block of code or a method at a time.

Example 1: Synchronizing a Method

In this example, we have a simple bank account where multiple threads deposit money concurrently. Without synchronization, the balance might not be updated correctly.

Without Synchronization

```
package com.mphasis.thread.conc;
public class BankAccount {
   private int balance = 0;

   public void deposit(int amount) {
      balance += amount;
   }
```

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

    public static void main(String[] args) throws InterruptedException {
        BankAccount account = new BankAccount();
        Runnable depositTask = () -> {
            for (int i = 0; i < 1000; i++) {
                account.deposit(1);
            }
        };

        Thread t1 = new Thread(depositTask);
        Thread t2 = new Thread(depositTask);

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

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

        System.out.println("Final balance: " + account.getBalance()); //
Might not be 2000
    }
}
```

In this example, without synchronization, the balance might not be correctly updated because both threads might read and write to the balance simultaneously.

With Synchronization

```
public class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        balance += amount;
    }
```

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

    public static void main(String[] args) throws InterruptedException {
        BankAccount account = new BankAccount();
        Runnable depositTask = () -> {
            for (int i = 0; i < 1000; i++) {
                account.deposit(1);
            }
        };

        Thread t1 = new Thread(depositTask);
        Thread t2 = new Thread(depositTask);

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

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

        System.out.println("Final balance: " + account.getBalance()); // Should be 2000
    }
}
```

Explanation:

This code creates a BankAccount class with a private balance variable and methods to deposit money and get the current balance. In the main method, it creates a BankAccount object and defines a depositTask that deposits 1 unit of money into the account 1000 times. It then creates two threads, t1 and t2, each running the depositTask. The main thread starts both t1 and t2, waits for them to finish using t1.join() and t2.join(), and then prints the final balance of the account.

The issue with this code is that the deposit method is not thread-safe, meaning that it is possible for both t1 and t2 to access and modify the balance variable at the same time, leading to a race condition. As a result, the final balance printed may not be 2000 as expected, but could be less due to the concurrent deposits. To fix this issue, the deposit method should be synchronized to ensure that only one thread can access it at a time.

- The deposit method is marked with the synchronized keyword. This ensures that only one thread can execute the deposit method at a time for the same instance of BankAccount.

Example 2: Synchronizing a Block of Code

Sometimes, you only need to synchronize a specific part of a method rather than the whole method. You can do this using synchronized blocks.

```
package com.mphasis.thread.conc;

public class Counter {
    private int count = 0;

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

    public int getCount() {
        return count;
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);

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

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

        System.out.println("Final count: " + counter.getCount()); // Should
be 2000
    }
}
```

Explanation:

- In this example, only the critical section (the `count++` line) is synchronized using a synchronized block. This ensures that only one thread can execute this block of code at a time.

Example 3: Synchronizing Static Methods

Static methods are shared among all instances of a class. To synchronize access to a static method, you need to synchronize on the class itself.

```
package com.mphasis.thread.conc;

public class SharedResource {
    private static int count = 0;

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

    public static int getCount() {
        return count;
    }

    public static void main(String[] args) throws InterruptedException {
        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                SharedResource.increment();
            }
        };
```

```
        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);

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

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

        System.out.println("Final count: " + SharedResource.getCount()); // Should be 2000
    }
}
```

Explanation:

- The `increment` method is a static method, so it's synchronized on the class (`SharedResource.class`). This ensures that all threads have to wait their turn to execute this method.

Summary

- **Synchronization** is used to ensure that only one thread at a time can access a critical section of code or resource.
- **`synchronized` Keyword** can be used to synchronize methods or blocks of code.
- **Synchronized Methods**: All methods are locked on the instance (or class for static methods).
- **Synchronized Blocks**: Allows more fine-grained control over which parts of a method are synchronized.