Names: Nadege Gaju

Understanding the Differences Between Concurrency and Multithreading

Concurrency and multithreading are related concepts but differ in how they manage the execution of tasks.

- **Concurrency**: Refers to the ability of a system to handle multiple tasks simultaneously. It doesn't necessarily mean that these tasks are being executed at the same time. Instead, concurrency allows multiple tasks to make progress over time by interleaving their execution.
- **Multithreading**: A specific way to achieve concurrency where multiple threads (smaller units of a process) run in parallel. Multithreading allows a program to perform multiple tasks simultaneously by using different threads within a single process.

Example:

Imagine you're a barista at a coffee shop, and you have several orders to prepare.

- **Concurrency**: You start preparing an espresso, then while the espresso machine is running, you begin making a sandwich. Once the espresso is done, you switch back to it, and then return to the sandwich. You're managing multiple tasks concurrently, switching between them.
- **Multithreading**: You have an assistant. While you focus on making the espresso, your assistant prepares the sandwich. Both tasks are done simultaneously by different people (threads).

Common Concurrency Issues

1. **Race Conditions**: Occur when two or more threads access shared data and try to change it at the same time. The outcome depends on the order in which the threads execute, leading to unpredictable results.

Example:

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


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

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

If two threads call the increment() method simultaneously, they might read the same value of count, increment it, and write the same result back, leading to an incorrect count.

2. **Deadlocks**: Occur when two or more threads are blocked forever, each waiting for the other to release a resource.

Example:

```
public class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            synchronized (lock2) {
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            synchronized (lock1) {
            }
        }
    }
}
```

If one thread executes method1() and another executes method2(), they may each hold one lock and wait indefinitely for the other, causing a deadlock.

3. **Livelocks**: Occur when threads keep changing their state in response to the other threads without making any progress. Unlike deadlocks, the threads are not blocked but still can't proceed.

Thread-Safe Collection Classes in Java

Java provides thread-safe collection classes to handle concurrency issues when working with collections.

1. **Vector**: A thread-safe version of ArrayList. It synchronizes each operation, ensuring that only one thread can modify the collection at a time.

Example:

```
Vector<Integer> vector = new Vector<>();

vector.add(1);

vector.add(2);
```

2. **Hashtable**: A thread-safe version of HashMap. It synchronizes all methods so that only one thread can access the table at a time.

Example:

```
Hashtable<String, String> hashtable = new Hashtable<>();

hashtable.put("key1", "value1");

hashtable.put("key2", "value2");
```

3. **Collections.synchronizedList**: A wrapper for ArrayList or other lists that adds synchronization to make it thread-safe.

Example:

```
List<Integer> list = Collections.synchronizedList(new ArrayList<>());

list.add(1);

list.add(2);
```

4. **ConcurrentHashMap**: A more advanced, high-performance alternative to Hashtable. It divides the map into segments and only locks the segment being accessed by a thread, allowing greater concurrency.

Example:

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();

map.put("key1", "value1");

map.put("key2", "value2");
```

Conclusion

Understanding concurrency and multithreading, along with the associated issues, is crucial for building reliable and scalable applications. Using thread-safe collections like ConcurrentHashMap can help us avoid common pitfalls such as race conditions and deadlocks.