Concurrency and Multithreading in Java

Concurrency and multithreading are core aspects of modern programming, allowing efficient execution of multiple tasks. Concurrency refers to executing multiple tasks seemingly at the same time, while multithreading achieves this by running multiple threads concurrently within a program. Java provides built-in concurrency utilities and concurrent collections to handle multi-threaded operations efficiently.

Concurrency vs. Multithreading
Multithreading Example:

```java
public class MultithreadingExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread 1 - Count: " + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread 2 - Count: " + i);
            }
        });

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

In this example, two threads run concurrently, each executing a loop. This showcases multithreading, where multiple threads share the same resources (CPU) and run simultaneously.

Concurrency Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConcurrencyExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.submit(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Task 1 - Count: " + i);
            }
        });

        executor.submit(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Task 2 - Count: " + i);
            }
        });

        executor.shutdown();
    }
)
```

In this case, the `ExecutorService` manages the tasks concurrently by allocating thread pools to execute them. Unlike multithreading, concurrency abstracts the low-level thread management.

Concurrent Collections

Java offers specialized collections to safely manage data in a concurrent environment, such as `ConcurrentHashMap` and `CopyOnWriteArrayList`.

ConcurrentHashMap Code Examples

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentCollectionExample {
    public static void main(String[] args) {
        ConcurrentHashMap<Integer, String> map = new
ConcurrentHashMap<>();

        map.put(1, "Java");
        map.put(2, "Concurrency");

        Thread t1 = new Thread(() -> {
            map.put(3, "Thread-safe");
            System.out.println("Thread 1 - Map: " + map);
        });

        Thread t2 = new Thread(() -> {
            map.put(4, "Collections");
            System.out.println("Thread 2 - Map: " + map);
        });

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

ConcurrentHashMap ensures thread-safe operations without locking the entire map, thus allowing multiple threads to update different parts concurrently.

CopyOnWriteArrayList

```
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new
CopyOnWriteArrayList<>();

        list.add("Java");
        list.add("Concurrency");
```

```java
        Thread t1 = new Thread(() -> {
            list.add("Thread-safe");
            System.out.println("Thread 1 - List: " + list);
        });

        Thread t2 = new Thread(() -> {
            list.add("Collections");
            System.out.println("Thread 2 - List: " + list);
        });

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

CopyOnWriteArrayList ensures safe concurrent modification by copying the list upon every write, minimizing conflicts.

Performance Comparison

The following code compares the performance of concurrent and non-concurrent collections:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class PerformanceComparison {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> concurrentList = new CopyOnWriteArrayList<>();

        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            arrayList.add(i);
        }
        long end = System.currentTimeMillis();
        System.out.println("ArrayList time: " + (end - start) +
"ms");

        start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            concurrentList.add(i);
        }
        end = System.currentTimeMillis();
        System.out.println("CopyOnWriteArrayList time: " + (end -
start) + "ms");
    }
}
```

Conclusion

- Concurrency vs. Multithreading: Concurrency abstracts thread management, while multithreading explicitly uses threads.
- Concurrent Collections: Java provides ConcurrentHashMap and CopyOnWriteArrayList to ensure safe concurrent operations.
- Performance: Concurrent collections are slower in write-heavy operations due to their thread-safety mechanisms but perform better in read-heavy scenarios compared to non-concurrent collections.