Concurrent Collections

This lesson gives a brief introduction about Java's concurrent collection classes.

Concurrent Collections

A bit of history before we delve into concurrent collection classes offered by Java. When the Collections Framework was introduced in JDK 1.2, it didn't come with collections that were synchronized. However, to cater for multithreaded scenarios, **the framework provided static methods to wrap vanilla collections in thread-safe wrapper objects.** These thread-safe wrapper objects came to be known as *wrapper collections*.

Example Wrapper Collection

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

For design pattern fans, this is an example of the *decorator pattern*.

Java 5 introduced thread-safe concurrent collections as part of a much larger set of concurrency utilities. The concurrent collections remove the necessity for client-side locking. In fact, external synchronization is not even possible with these collections, as there is no one object which when locked will synchronize all instance methods. **If you need thread safety, the concurrent collections generally provide much better performance than synchronized (wrapper) collections**. This is primarily because their throughput is not reduced by the need to serialize access, as is the case with synchronized collections. Synchronized collections also suffer from the overhead of managing locks, which can be high if there is much contention.

The concurrent collections use a variety of ways to achieve thread-safety while avoiding traditional synchronization for better performance. These

while avoiding traditional synchronization for better performance. These are:

- **Copy on Write:** Concurrent collections utilizing this scheme are suitable for read-heavy use cases. An immutable copy is created of the backing collection and whenever a write operation is attempted, the copy is discarded and a new copy with the change is created. Reads of the collection don't require any synchronization, though synchronization is needed briefly when the new array is being created. Examples include `CopyOnWriteArrayList` and `CopyOnWriteArraySet`

- **Compare and Swap:** Consider a computation in which the value of a single variable is used as input to a long-running calculation whose eventual result is used to update the variable. Traditional synchronization makes the whole computation atomic, excluding any other thread from concurrently accessing the variable. This reduces opportunities for parallel execution and hurts throughput. An algorithm based on CAS behaves differently: it makes a local copy of the variable and performs the calculation without getting exclusive access. Only when it is ready to update the variable does it call CAS, which in one atomic operation compares the variable's value with its value at the start and, if they are the same, updates it with the new value. If they are not the same, the variable must have been modified by another thread; in this situation, the CAS thread can try the whole computation again using the new value, or give up, or— in some algorithms — continue, because the interference will have actually done its work for it! Collections using CAS include `ConcurrentLinkedQueue` and `ConcurrentSkipListMap`.

- **Lock:** Some collection classes use `Lock` to divide up the collection into multiple parts that can be locked separately resulting in improved concurrency. For example, `LinkedBlockingQueue` has separate locks for the head and tail ends of the queue, so that elements can be added and removed in parallel. Other collections using these locks include `ConcurrentHashMap` and most of the implementations of `BlockingQueue`.

CopyOnWrite Example

Lets take an example with a regular `ArrayList` along with a `CopyOnWriteArrayList`. We will measure the time it takes to add an item to an already initialized array. The output from running the code widget below demonstrates that the `CopyOnWriteArrayList` takes much more time than a regular `ArrayList` because under the hood, all the elements of the `CopyOnWriteArrayList` object get copied thus making an insert operation that much more expensive.

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

/**
 * Java program to illustrate CopyOnWriteArrayList
 */
public class main
{
  public static void main(String[] args)
  throws InterruptedException
  {
    //Initializing a regular Arraylist
    ArrayList<Integer> array_list = new ArrayList<>();
    array_list.ensureCapacity(500000);
    //Initializing a new CopyOnWrite Arraylist with 500,000 numbers
    CopyOnWriteArrayList<Integer> numbers = new CopyOnWriteArrayList<>(array_list);

    //Calculating the time it takes to add a number in CopyOnWrite Arraylist
    long startTime = System.nanoTime();
    numbers.add(500001);
    long endTime = System.nanoTime();
    long duration = (endTime - startTime);

    //Calculating the time it takes to add a number in regular Arraylist
    long startTime_al = System.nanoTime();
    array_list.add(500001);
    long endTime_al = System.nanoTime();
    long duration_al = (endTime_al - startTime_al);

    System.out.println("Time taken by a regular arraylist: "+ duration_al + " nano seconds");
    System.out.println("Time taken by a CopyOnWrite arraylist: "+ duration + " nano seconds")

  }
}
```