Vector vs ArrayList in Java

Vector has been a part of Java since its initial release in 1996, before the introduction of ArrayList in later versions.

ArrayList were added in version 1.2.

To compare the performance of **ArrayList** and **Vector**, we have declared one instance of each data structure and measured their respective execution times.

```
import java.util.ArrayList;
import java.util.Vector;

public class VectorArrayList {
    public static void main(String[] args) {
        int no_of_elements = 1000000;

        //ArrayList
        ArrayList<Integer> arrayList= new ArrayList<>();
        long start = System.currentTimeMillis();// gives the
current time in milliseconds
        for (int i = 0; i < no_of_elements; i++) {
            arrayList.add(i); //Adding 1000000 elements
        }
        long end = System.currentTimeMillis();
        System.out.println("\nAdded "+no_of_elements+"
elements to ArrayList in "+(end-start)+" ms\n");


        //Vector
        Vector<Integer> vector= new Vector<>();
        start = System.currentTimeMillis();// gives the
current time in milliseconds
        for (int i = 0; i < no_of_elements; i++) {
            arrayList.add(i); //Adding 1000000 elements
        }
        end = System.currentTimeMillis();
        System.out.println("Added "+no_of_elements+" elements
to Vector in "+(end-start)+" ms");

    }
}
```

Output

```
Added 1000000 elements to ArrayList in 42 ms

Added 1000000 elements to Vector in 135 ms
```

It can be observed that **ArrayList** performs significantly faster than **Vector**.

While ArrayList took only **42** milliseconds to execute, Vector took significantly longer at **135** milliseconds.

Let us introduce two threads each for adding elements to ArrayList and Vector, and compare their performance.

As you can see in the output, they both took almost the same time.

```
Added elements in multithreaded way to Arraylist in 163 milliseconds

Added elements in multithreaded way to Vector in 162 milliseconds
```

Again, run this program for two or three times,

```
Exception in thread "Thread-1" Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException: Index 113 out of bounds for length 109
    at java.base/java.util.ArrayList.add(ArrayList.java:455)
    at java.base/java.util.ArrayList.add(ArrayList.java:467)
    at VectorArrayList.lambda$main$1(VectorArrayList.java:20) <1 internal call>
java.lang.ArrayIndexOutOfBoundsException Create breakpoint : Index 113 out of bounds for length 109
    at java.base/java.util.ArrayList.add(ArrayList.java:455)
    at java.base/java.util.ArrayList.add(ArrayList.java:467)
    at VectorArrayList.lambda$main$0(VectorArrayList.java:14) <1 internal call>

Added elements in multithreaded way to Arraylist in 9 milliseconds

Added elements in multithreaded way to Vector in 194 milliseconds
```

Here we have got an Exception, why?

To understand the reason behind the performance difference, we can determine the size of the multithreaded Vector and ArrayList once they have completed execution.

We can add the following code to the end of the "MultiThreading in ArrayList" and "MultiThreading in Vector" sections, after the print statement,

```
System.out.println("Size of multithreaded ArrayList
is : "+multiThreadArrayList.size());
```

```
System.out.println("Size of multithreaded Vector is :
"+multiThreadVector.size());
```

As we are adding 1000000 elements, therefore size should be 2000000 of both the multithreadedvector and multithreadedlist.

```
Added elements in multithreaded way to Arraylist in 152 milliseconds
Size of multithreaded ArrayList is : 1327174

Added elements in multithreaded way to Vector in 158 milliseconds
Size of multithreaded Vector is : 2000000
```

As you can see a **Vector** does have 2000000 elements in it, but the **ArrayList** does not.

Here are some output examples, when we run these code multiple times.

i) Output-1

```
Added elements in multithreaded way to Arraylist in 174 milliseconds
Size of multithreaded ArrayList is : 1350729

Added elements in multithreaded way to Vector in 167 milliseconds
Size of multithreaded Vector is : 2000000
```

ii)

```
Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : Index 29 out of bounds for length 15
    at java.base/java.util.ArrayList.add(ArrayList.java:455)
    at java.base/java.util.ArrayList.add(ArrayList.java:467)
    at VectorArrayList.lambda$main$0(VectorArrayList.java:14) <1 internal call>

Added elements in multithreaded way to Arraylist in 45 milliseconds
Size of multithreaded ArrayList is : 1000010

Added elements in multithreaded way to Vector in 219 milliseconds
Size of multithreaded Vector is : 2000000
```

iii)

```
Added elements in multithreaded way to Arraylist in 175 milliseconds
Size of multithreaded ArrayList is : 1290493

Added elements in multithreaded way to Vector in 163 milliseconds
Size of multithreaded Vector is : 2000000
```

We can see that we are getting very unpredictable result each time.

Vector is adding exact number of elements in it, but the ArrayList does not.

Why this is happening? Why ArrayList is giving this unpredicted answer if we try to use the it in multi-threaded way?

This is because the operation on an **ArrayList** is not Synchronized (They are not Threads safe).

But operation on **Vector** is synchronized, therefore we can operate on Vector using multiple threads at the same time.

Vector can be helpful if we are using multiple threads, but that thread safety comes at a performance cost. [*takes more time*]

If we are not using threads then the Vector take all that extra time for no real gain at all.

If we are unfamiliar with Vector and prefer to use ArrayList in a multi-threaded environment without risking unpredictable output, we can make a small modification to our code.

Instead of declaring this way,

```
List<Integer> multiThreadList = new ArrayList<>();
```

We can declare this way

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

By creating a **Wrapper** around the ArrayList (synchronizing its multi-threaded operations), we can achieve automatic synchronization similar to that provided by Vector.

Output after adding Collections.synchronizedList() into the ArrayList.

```
Added elements in multithreaded way to Arraylist in 163 ms
Size of multithreaded ArrayList is : 2000000

Added elements in multithreaded way to Vector in 175 ms
Size of multithreaded Vector is : 2000000
```

The output indicates that ArrayList can add 2000000 elements successfully, with performance like that of Vector as both data structures took almost the same amount of time to execute.

Full Code

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Vector;

public class VectorVsArraylist {
    public static void main(String[] args) throws
InterruptedException {

        int size = 1000000;

        //ArrayList
        List<Integer> arrayList = new ArrayList<>();
        long start = System.currentTimeMillis(); //
gives current time in milliseconds
        for (int i = 0; i <size ; i++) {
            arrayList.add(i);
        }
        long end = System.currentTimeMillis();
      System.out.println("\nAdded "+size+" elements to
ArrayList in "+(end-start)+" ms\n");

        // Vector
```

```
        List<Integer> vector = new Vector<>();
         start = System.currentTimeMillis();
        for (int i = 0; i <size ; i++) {
            vector.add(i);
        }
         end = System.currentTimeMillis();
        System.out.println("Added "+size+" elements
to Vector in "+(end-start)+" ms");

        // Multithreading in ArrayList
        List<Integer> multiThreadList =
Collections.synchronizedList(new ArrayList<>());
//      List<Integer> multiThreadList = new
ArrayList<>();
        start = System.currentTimeMillis();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i <size ; i++) {
                multiThreadList.add(i);
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i <size ; i++) {
                multiThreadList.add(i);
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        end = System.currentTimeMillis();
        System.out.println("\nAdded elements in
multithreaded way to Arraylist in "+(end-start)+"
ms");
        System.out.println("Size of multithreaded
ArrayList is : "+multiThreadList.size());


        // Multithreading in Vector
        List<Integer> multiThreadVector = new
Vector<>();
        start = System.currentTimeMillis();
         t1 = new Thread(() -> {
            for (int i = 0; i <size ; i++) {
                multiThreadVector.add(i);
```

```
            }
        });
         t2 = new Thread(() -> {
            for (int i = 0; i <size ; i++) {
                multiThreadVector.add(i);
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        end = System.currentTimeMillis();
        System.out.println("\nAdded elements in
multithreaded way to Vector in "+(end-start)+" ms");
        System.out.println("Size of multithreaded
Vector is : "+multiThreadVector.size());


    }
}
```

Output

```
Added 1000000 elements to ArrayList in 36 ms

Added 1000000 elements to Vector in 108 ms

Added elements in multithreaded way to Arraylist in 166 ms
Size of multithreaded ArrayList is : 2000000

Added elements in multithreaded way to Vector in 178 ms
Size of multithreaded Vector is : 2000000
```

Summary

Difference between Vector and ArrayList in Java

In Java, both **Vector** and **ArrayList** are classes that implement the **List** interface and provide dynamic arrays that can grow or shrink in size as needed. However, there are some differences between the two:

i) **Synchronization**: **Vector** is synchronized, meaning that it ensures thread-safety by allowing only one thread to access the vector at a time. This makes it slower than **ArrayList**, which is not synchronized and allows multiple threads to access the array simultaneously.

ii) **Performance**: Because of its synchronization, **Vector** is generally slower than **ArrayList**. This is because the synchronization overhead adds extra time for each operation, which can become significant for large vectors.

iii) **Growth rate**: **Vector** and **ArrayList** have different growth rates. When an element is added to an **ArrayList**, it increases its size by 50% of its current capacity. **Vector**, on the other hand, doubles its size when its capacity is exceeded. This means that **Vector** may waste memory if it grows beyond its necessary size, while **ArrayList** may need to reallocate the array more frequently than necessary.

iv) **Legacy**: **Vector** is a legacy class, which means that it has been around since Java **1.0** and is still maintained for backward compatibility with older code. **ArrayList** was introduced in **Java 1.2** and is the preferred choice for most new code.

Conclusion

In most cases where operations are performed in a single thread, it is recommended to use **ArrayList** over **Vector** due to its superior performance. For multi-threaded operations, we can add synchronization using **'Collections.synchronized() '** to our **ArrayList**, rather than relying on **Vector**. Using **Vector** in a single-threaded environment can result in unnecessary performance sacrifices.