# Dynamic Collection

Arrays are a great way to store multiple values, but what if we wanted to add a new one? If we have declared an array with 5 elements [0-4] what happens when we try to assign a value to index 5?

```
String[] students = new String[5];
students[5] = "Frank";
```

Poor Frank can't be enrolled because Java won't let us extend the array. This is the constraint of arrays - they are a fixed size. The only way we can add Frank to the array is to create a new array with the space for one more element, copy across all the existing students and then add Frank to the last index. This is a very slow way to enrol students. If only there was a more dynamic way to store our data!

## ArrayList

ArrayLists are a dynamic data structure that allow us to store a collection of values, but the size can dynamically change at runtime. This is not a built in type so we will need to include the package at the top of the file.

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

The following is how we create an ArrayList of Strings.

```
ArrayList<String> students = new ArrayList<String>();
```

This syntax may look a little confusing so let's break it down. We are creating a new ArrayList variable. The next bit between the <> symbols is the datatype we want to store in it. We then give it the identifier students and assign it the result of creating a new ArrayList of type <String>. This final () brackets are because the new keyword always calls the constructor, so we can pass it parameters if we need to. Now let's add a new item to the ArrayList.

```
students.add("Frank");
```

Wow, that was easy. Let's add someone else!

```
students.add("Mary");
```

Magic! We can even print out all the students stored in an array list by passing the entire ArrayList to the println statement.

```
System.out.println(students);
```

Unfortunately Frank has withdrawn from the unit so how can we remove him from the student list?

```
students.remove(0);
```

There are a whole bunch of other methods that exist on the ArrayList. Have a read through the **documentation (https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist?view=netframework-4.7.2)** to see what else is available.

# HashMap

The other dynamic data structure that we will be using is the HashMap. A HashMap is similar to an ArrayList, but it stores the data as a key/value pair. This can be thought of as a dictionary. When you are not sure of a word you look it up in a dictionary. The key (word) links to a value (definition). To include the HashMap type we need to add another package.

```
import java.util.HashMap;
```

And to declare a new instance, we need to provide the type for the key and the value.

```
HashMap<String, String> teachers = new HashMap<String, String>();
```

Here we are using a String key that is mapped to a String value.

To add a new value we simply need to provide a Key and Value pair.

```
teachers.put("jon", "software developer");

teachers.put("lav", "java specialist");
```

And we can access a value by providing the key.

```
System.out.println(teachers.get("lav"));
```

This should print out java specialist to the console.

Excellent! Again, we recommend having a look over the **documentation (https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html)** as being familiar with these dynamic data structures can really improve the quality of your programs!

# Foreach Loops

Now that we are dealing with dynamic data structures that can change size at runtime, it would be nice to have a loop that was just as flexible. Introducing the for-in loop!

Rather than the while/do while loops where we need to manually track the index we are up to, the for-in loop will iterate over the entire collection from start to finish.

```
ArrayList<Integer> scores = new ArrayList<Integer>();
```

```
scores.add(5);
scores.add(15);
scores.add(52);
Integer total = 0;
for(Integer score : scores) {
    total += score;
}
System.out.println(String.format("The total of the scores is: %d", total));
```

Now if we added a new score to the ArrayList it will not affect the for-in loop. No matter what the size is it will always run from the lowest to the highest index. So convenient! Let's see how that might work with a HashMap.

```
HashMap<String, String> teachers = new HashMap<String, String>();
teachers.put("jon", "software developer");
teachers.put("lav", "java specialist");
for(String key:teachers.keySet()) {
    System.out.println(teachers.get(key));
}
```

The way the for-each loop works is that iterates through the collection and  returns a copy of each object in that position. Think of the collection as a book. When one asks for the book, you may just give a photocopy of the pages. Anything they write on the pages will not make a difference to the book. However, the normal for-loop will directly refer to the index position in the collection. So any changes in that, will change the actual collection.