ArrayList Basics

This lesson will generally describe the List Interface, but it's often better to learn about these abstractions using an example. Therefore, we're going to start off with the basics of `ArrayList`. Hopefully, our more general description of the `List` interface will make more sense once you've internalized some of the specifics that refer to the `ArrayList`, which is an implementation of the more general purpose `List` interface.

An **ArrayList** (also referred to as a list) is a resizable array where the resizing is performed behind the scenes. The underlying array grows or shrinks as needed, and elements shift automatically to create or remove gaps. You can use an `ArrayList` instead of an `Array` at any time, but `ArrayLists` are especially useful **when you don’t know the size of the array you need or when you need to add and remove items frequently.**

Like arrays, `ArrayList` elements are indexed, beginning at 0. When inserting or removing items from a list, index values of elements will change frequently because the items will shift to create or remove gaps. As a result, the size of an `ArrayList` also fluctuates.

`ArrayLists` are declared and instantiated as follows:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
ArrayList<E> listName = new ArrayList<E>();
```

In the above, E represents an **object type**, like String, Integer, or a class of our own creation. The size of the list, when first initialized, is 0. **Note: primitives do not work with** `ArrayLists`. If you want to put primitives in an ArrayList, you must wrap the values to become **Objects forms, such as** `Integer` **for** `int`.

Once instantiated, ArrayLists call methods with the dot operator to access, modify, or remove items in the list. The general syntax of an `ArrayList` method call is `listName.methodName(args…)`

The following `ArrayList` methods are often used, so we've demonstrated them in the code samples below. Access the full Java Quick Reference to view the full list of methods if you like. Note: In the examples below, `E` represents whatever data type is being stored in the ArrayList:

- `int size()`: returns the number of elements in the list.
- `boolean add(E obj)`: appends `obj` to the end of the list and returns `true` if successful.
- `void add(int index, E obj)`: moves any current objects at the given index or beyond to the right (i.e., to a higher index) and inserts the given `obj` at the given `index`.
- `E remove(int index)`: removes the item at the `index` and shifts the remaining items to the left (i.e., to a lower index).
- `E get(int index)`: returns the item in the list at the given `index`.
- `E set(int index, E obj)`: replaces the item at the given index with the `obj` you've passed in as a parameter.

#Code Sample #1 - ArrayList of Strings

Run the code as is, then play around with the specific parameters to see the effects. Can you cause some errors? What did you do, and how do you fix them? You're going to be working with Collections rather extensively from here on out, so get in there and get your hands dirty:

Run

ArrayListBasics.java

```
import java.util.*; // This is the required import for an ArryaList and/or Lis
t

public class ArrayListBasics {
    public static void main(String[] args) {

        // DECLARE, INITIALIZE, SIZE, and PRINT
        ArrayList<String> nameList = null; /* Declare */
        System.out.println(nameList);

        nameList = new ArrayList<String>(); /* Allocate memory*/
        System.out.println(nameList);
        System.out.println("size is " + nameList.size());

        // ADD w/o index
        nameList.add("Andrea");
        System.out.println(nameList);
        nameList.add("Bob");
        System.out.println(nameList);
        nameList.add("Carrie");
        System.out.println(nameList);
        nameList.add("Eduardo");
        System.out.println(nameList);
        System.out.println("size is " + nameList.size());

        // ADD w/ index
        nameList.add(3, "Dierdre");
        System.out.println(nameList);
        System.out.println("size is " + nameList.size());

        // GET
        System.out.println("name at index 2 is " + nameList.get(2));

        // SET
        nameList.set(1, "Tori");
        System.out.println(nameList);
        System.out.println("size is " + nameList.size());

        // REMOVE
        nameList.remove(3);
        System.out.println(nameList);
        System.out.println("size is " + nameList.size());
    }
}
```

#Wrapper Classes

ArrayLists can only hold object types, so they cannot be created using primitive types like `int`, `double`, `boolean`, etc. Luckily, the Wrapper classes we've already learned about can be used to seamlessly store primitive values into lists. For example, the `Integer` class can be used to store `int` values and the `Double` class can be used to store `double` values, as shown in the code below.

#Code Sample #2 - ArrayList of Integers

Again, spend a few minutes playing with the code below. What kind of error do you get if you try to pass in a primitive? What happens if you pass in the wrong kinds of values? Breaking things is as important as making things work when first learning a new technology.

Run

WrapperBasics.java

```
import java.util.*; // REQUIRED IMPORT for ArryaList and List

public class WrapperBasics {
    public static void main(String[] args) {

       // DECLARE, INITIALIZE, SIZE, and PRINT
       ArrayList<Integer> numbers = null;
       System.out.println(numbers);

       numbers = new ArrayList<Integer>();
       System.out.println(numbers);
       System.out.println("size is " + numbers.size());

       // ADD w/o index
       numbers.add(1);                      // Autoboxing example
       System.out.println(numbers);
       numbers.add(3);
       System.out.println(numbers);
       numbers.add(4);
       System.out.println(numbers);
       numbers.add(Integer.valueOf(5));     // no boxing necessary
       System.out.println(numbers);
       System.out.println("size is " + numbers.size());

       // ADD w/ index
       numbers.add(1, 2);
       System.out.println(numbers);
       System.out.println("size is " + numbers.size());

       // GET
       int x = numbers.get(2);       // Unboxing example
       System.out.println("number at index 2 is " + x);

       // SET
       numbers.set(3, 100);
       System.out.println(numbers);
       System.out.println("size is " + numbers.size());
```

```
        // REMOVE
        numbers.remove(3);
        System.out.println(numbers);
        System.out.println("size is " + numbers.size());
    }
}
```

#Tracing Code

Given an ArrayList called `values` with the following integers:

`[25, 73, 14, 85, 9]`

Drag the blocks to show the output after running the code below. Remember: not all blocks will necessarily be used in the solution.

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
values.set(3, 555);
System.out.println(values);

values.add(2, 5);
System.out.println(values);

values.add(777);
System.out.println(values);

values.remove(1);
System.out.println(values);
```

Options

```
[73, 5, 14, 555, 9, 777]
[25, 73, 5, 555, 9]
[25, 73, 14, 555, 9]
[25, 73, 5, 14, 555, 9]
```

```
[25, 73, 5, 14, 555, 9, 777]
[25, 5, 14, 555, 9, 777]
```

Solution

InfoWarningTip

Make sure you attempt the problem above before scrolling down. A lot of coding is learning how to read/trace what's going on in a program, especially when running the entire thing won't help you understand the individual steps. Take your time here, so you aren't hit with impostor syndrome later.

#How to get the Answer Above

Trace the contents of the `ArrayList` one line at a time:

Index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**size of list**
initial	25	73	14	85	9			5
set(3,555)	25	73	14	555	9			5
add(2,5)	25	73	5	14	555	9		6
add(777)	25	73	5	14	555	9	777	7
remove(1)	25	5	14	555	9	777		6

#List Interface

Now that you've seen the ArrayList in action, it should be easier to understand its more abstract parent interface: the `List`.

As discussed in the previous lesson, `ArrayList` **implements** the `List` interface. That means that everything described above is one way to utilize the `List` interface:

turning the `List` interface's *super* powers into an object that you can work with. By itself, the `List` interface represents an **ordered** collection of items that also keeps track of the order in which items were added to the given list. It **extends** the Collection interface, but, as an interface, cannot be used directly to create objects. Instead, we create objects using the following classes that implement `List`'s functionality:

- **ArrayList** is a resizable array found in the `java.util` package, and behaves as described in this lesson. Technically speaking, when we use an `ArrayList` and add a new item. If no more available slots exist, a new array with more slots is created, and the old array is copied into the new one. Finally, the old array is left to be garbage collected later.
- **LinkedList** is very similar to the `ArrayList` and can be used in similar ways, but the way that it works behind the scenes is quite different. Rather than having an array inside of an object as the `ArrayList` does, the `LinkedList` works by stringing together a series of **containers**. Each container contains a link to the next container in the list, so when we add a new element to the `LinkedList`, it simply creates a new container and links it to the previous element. This means in practice that people generally use `ArrayList` for **data storage**, but they use `LinkedList` for **data manipulation.** We'll talk more about `LinkedLists` in the lessons that follow.
- **Vector** can be thought of as a precursor to `ArrayList`. Essentially, it is a less-optimized version of the `ArrayList`, so we won't really be going into much detail here. If you'd like to read more, here's a short tutorial about it.
- **Stack** represents a Last In, First Out (LIFO) `List`. That means when you push a new element to the list, it ends up at the top. The same goes for if you remove (pop) an item from the list–it'll pop right off of the top. This is an extension of the `Vector` class, so it isn't much used anymore. Here's a short tutorial for those of you interested in it.

The `List` interface comes with its own built-in methods that are passed down to its subclasses:

- `boolean add(ElementType element)`**:** This adds an element to the given list according to the rules that govern the given list type.

- `boolean addAll(int index, Collection c)`: This adds a collection to the given index. You can use this one to add multiple values to a list you've already created.
- `object remove(int index)`: This removes whatever `object` exists at the given `index`.
- `object get(int index)`: This returns the value of the `object` at the given `index`.
- `object set(int index, Obj obj)`: This overwrites the `object` at the given `index` with the object `obj` you provide.
- `int indexOf(Object obj)`: Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
- `int lastIndexOf(Object obj)`: Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

There are others, but these are the basics. We'll be talking about more implementations of the `List` interface as we continue throughout the week.

Remember to watch the differences between the various lists/collections and think about when they will be helpful.

#`ArrayList` or `List`?

Technically, when declaring any list (`ArrayList`, `LinkedList`, etc.), you can do so by using the `List` interface directly or the specific interface that you're going to use. Here's an example of the two options for declaring an `ArrayList`:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdownPascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
List<T> variableName = new ArrayList<T>();
ArrayList<T> variableName = new ArrayList<T>();
```

So what's the difference? The difference is rooted in the behaviour of interfaces and polymorphism. If you declare an interface and initialize it with an implementing class, that class object would only have access to the methods that the `List` interface itself features. Because `ArrayList` is essentially a "child" of the `List` interface, implementing `ArrayList` will give you access to all of those "parent" methods, as well as everything specific to the `ArrayList` interface itself. Remember: just like child classes inherit all of the "super" powers of their parent/ancestor classes, so do the sub-interfaces inherit the methods of their "parents". When you implement an interface, think about what you'll need. If you need to be able to switch out implementations for the interface easily, then it's better to "declare" the object as an **interface** on the left-hand side and pass an implementing **class** on the right–as shown in the first option above. On the other hand, if you need specialized implementations for the interface's methods from the child class and won't care much about switching out implementing classes, then declare the object using the class type–as shown in option 2 above.

InfoWarningTip
This pattern will hold true as we explore the other collections as well. The child interfaces will have more functionality specific to their purpose, and you'll probably want to implement, say, `HashMap` instead of the basic `Map` interface. But, as with everything in code, it will depend on your circumstances, just like we described above.

#Knowledge Check

The ArrayList class ____ the List interface.

The List interface represents _____ set of data.

In the IDE below, use `ArrayList` methods to add the following elements to the `colors` list: Red, Green, Orange, White, Black. Make sure you maintain the given order in your assignments.

Run

Main.java

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

```java
public class Main {

    public static void main(String[] args) {

        List<String> colors = new ArrayList<>();

        // Add the colors in the given order
        colors.add("Red");
        colors.add("Green");
        colors.add("Orange");
        colors.add("White");
        colors.add("Black");

        // Prints the colors list to the terminal window.
        System.out.println(colors);
    }
}
```

Run

Main.java

```java
// Run this code, examine it, then answer the short answer question below:
public class Main {
    public static void main(String[] args) {

        List<Integers> grades = { 1, 2, 3 };

        System.out.printf("The size of the array is %d", grades.length());
    }
}
```

CONSOLESHELL

In your own words, why didn't the code block above run? What did you do to fix it?

The type specified in the list should be Integer, not Integers.

create an ArrayList of Integer objects, add integers to it,

Of the following, which choice represents the text that will be printed to the console?

```
import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
ArrayList<String> names = new ArrayList<>();
names.add("David");
names.add("Alice");
names.add("Bob");
names.add("Samir");
names.add("David");
int lastDavid = names.lastIndexOf("David");

System.out.println("Last David at " + lastDavid);
}
}
```

#Extra Resources

We highly recommend watching this video. John goes over the differences between Arrays and ArrayLists and what the various declarations and constructors do. Remember, you can always watch videos like this one at a higher speed, then slow things down if something confuses you. Alternatively, you can try to follow along in your own IDE and see what happens when you recreate his examples. It's up to you and the time you have available.

https://youtu.be/NbYgm0r7u6o