The ArrayList<> Class

The JAVA `List` interface defines the methods that the `ArrayList<>` class must implement. The `List` interface and classes such as `ArrayList` that implement it provide a more flexible mechanism than a traditional array. Before looking at the methods of this class, it is important to understand the syntax required to declare it.

Recall that there are up to three steps to declaring a variable:

1. The type
2. The variable name
3. Assigning the variable an initial value

When declaring a variable, the first two parts are always required, but the third part is optional, performed depending on how the variable will be used.

When declaring the type for a variable that will be an `ArrayList<>` you actually have two options, you can declare the variable as either a `List<>` **or** as an `ArrayList<>`. This is an often-overlooked benefit of having a class that implements an interface. You can use the interface like a super-class when specifying the type. But why would you do this?

It turns out that there are many other implementations of the `List` interface and if you specify the type as a `List` instead of an `ArrayList` you can substitute a different implementation very easily!

It is not necessary, but it is preferable to use `List<>` as the type.

Before you can declare a `List` you must know what type of object makes up the list. For our initial example, let's make a list of Strings, so the two possible ways to declare such a list are:

- ✓ `List<String>`
- ✓ `ArrayList<String>`

It is important to remember that the type of the object is List of String (`List<String>`) not just List!

The next part of a declaring a variable is the variable name. Let's make a list of names and since this is a list of many strings, make the variable name plural:

```
List<String> names
```

Next comes the optional part: assigning the variable an initial value.

In this case we want a new empty list to which we can add names. Remember that our `List` is going to actually be an `ArrayList` object, which means we need to call its constructor using the JAVA keyword `new`, and the constructor always shares the same name as the class. The name of the class is a bit more complicated though.

We are making an `ArrayList` of `String`, so the class is actually `ArrayList<String>`. Finally, since we are calling a constructor, which is a method, we must also include the parenthesis!

Thus the call to the constructor method reads: `new ArrayList<String>()`

And when we put the type, the variable name, and the call to the constructor together, we get:

```
List<String> names=new ArrayList<String>();
```

We could just as easily have declared our new list using its exact type instead of the interface:

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

Declaring the list object this way would simply make our code less flexible.

So now that we have instantiated a new empty list we can use the methods that are implemented as part of the List interface to interact with the it. The following methods are in JAVA AP subset:

- `int size()` returns the number of elements in the list (equivalent to `.length` for arrays).
- `boolean isEmpty()` returns `true` if the list has no elements.
- `boolean add(E element)` returns `true` if the `element` is added to the end of the list.
- `void add(int index, E element)` inserts the `element` at `index`.
- `E get(int index)` returns the `element` at `index`.
- `E set(int index, E element)` puts `element` at `index` and returns the object that was replaced.
- `E remove(int index)` removes the object at `index` and returns that object.
- `boolean contains(Object element)` returns `true` if the list contains the element.

In order to use the List interface and ArrayList class we will need to import them:

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

Then we can instantiate our `names` list using:

```
List<String> names=new ArrayList<String>();
```

Now let us interact with it using its methods:

The `int size()` method:

Just like working with arrays, it is often useful to know how many elements there currently in the `List`. Instead of using `.length` though we must use the `.size()` method.

```
int numberOfNames=names.size();
```

Since we just instantiated our list, the variable `numberOfNames` will have the value `0` after the line of code above is executed.

The `boolean isEmpty()` method:

If you simply want to find out if a list currently has no elements, you can use the `isEmpty()` method:

```
boolean areThereNames=names.isEmpty();
```

Is the equivalent of the code:

```
boolean areThereNames=names.size()==0;
```

The `boolean add(E)` method:

The first thing to notice in the method signature for this method is the use of the capital E where you would normally see an explicit type. This is the one of the ways in which the use of generics affects syntax. Because the type that makes up this list isn't known until you declare it, the type E represents whatever type makes up your particular list. Since we make a list of String, the `add()` method takes a String as a parameter, but in all the documentation, you will see a capital E as a place holder!

Let's add three names to the list using this method:

```
boolean added=names.add("Aaron");
names.add("Brandy");
names.add("Charlie");
```

After 3 calls to add:
names

index	Value
0	"Aaron"
1	"Brandy"
2	"Charlie"

We can picture the contents of the list the same way we do for arrays. The first element added to the list is put at index 0, and each new element is added at the end of the list.

Notice that this method returns a `boolean` value. For the purposes of the AP Subset there is no useful purpose to storing this result because it will always be `true`.

Therefore you can simply ignore the return result when calling this method.

The `void add(int index, E element)` method:

The add method is overloaded with another version that lets the programmer specify the index where the element is to be added. The index must be greater than 0 and less than or equal to the current size of the list. Any elements past the specified index will be moved down the list to make room.

Let's add another name to the list using this method:

```
names.add(0, "David");
```

After add at index
names

index	Value
0	"David"
1	"Aaron"
2	"Brandy"
3	"Charlie"

The `E get(int index)` method:

The `get()` method returns the element in the list at the `index`.

We will get the fourth name, found at index 3:

```
String theName=names.get(3);
```

After this line of code is executed, the value of the variable `theName` is "Charlie."

The `E set(int index, E element)` method:

The `set()` method allows a value in the list to be replaced at a given index. The old value is returned.

We will change the name found at index 2 to "Edith" and store the old value:

```
String oldName=names.set(2, "Edith");
```

After this line of code is executed, the value of the variable `oldName` is "Brandy."

After call to set
names

index	Value
0	"David"
1	"Aaron"
2	"Edith"
3	"Charlie"

The `E remove(int index)` method:

The `remove()` method removes a value from the list and returns the removed value.

Let's remove the second element, found at index 1:

```
String removedName=names.remove(1);
```

After this line of code is executed, the value of the variable `removedName` is "Aaron."

After call to remove
names

index	Value
0	"David"
1	"Edith"
2	"Charlie"

The `boolean contains(E element)` method:

The `contains()` method returns `true` if the value of the parameter `element` is found in the list.

Is "Aaron" in the list?

```
boolean inList=names.contains("Aaron");
```

After this line of code is executed, the value of the variable `inList` is `false`.

Is "Edith" in the list?

```
boolean inList2=names.contains("Edith");
```

After this line of code is executed, the value of the variable `inList2` is `true`.

Working with an `ArrayList<>`:

Now that we have seen each of the methods that make up the `List<>` interface and thus the concrete `ArrayList<>` class, it is time to see some examples of how to work with them.

The code below will declare, instantiate, fill, set, and remove names from a list of String:

```
01 import java.util.ArrayList;
02 import java.util.List;
03
04 public class FillArrayList {
05    public static void main(String[] args) {
06       List<String> names=new ArrayList<String>();
07       names.add("Aaron Braskin");
08       names.add("Brandy Gaunt");
09       names.add("Charlie White");
10       names.add(0, "David Conner");
11       names.set(2, "Gena Brenan");
12       names.remove(1);
13    }
14 }
```

Line 07: names

index	Value
0	"Aaron Braskin"

Line 08: names

index	Value
0	"Aaron Braskin"
1	"Brandy Gaunt"

Line 09: names

index	Value
0	"Aaron Braskin"
1	"Brandy Gaunt"
2	"Charlie White"

Line 10: names

index	Value
0	"David Connor"
1	"Aaron Braskin"
2	"Brandy Gaunt"
3	"Charlie White"

Line 11: names

index	Value
0	"David Connor"
1	"Aaron Braskin"
2	"Gena Brenan"
3	"Charlie White"

The contents of `names` after each call that modifies it is made is on the right:
But instead of taking this on faith, let's write some code that will actually display the contents of the list to the console.
Just like processing elements of an array, we will need a loop.

To iterate over each element of the list, we will write some very familiar looking code:
`for (int i=0; i<names.size(); i++) { }`

Notice that the only difference in our loop code from working with arrays is that we use `.size()` instead of `.length`.

Inside the loop we will need to process each element of the list. We will use the `get()` method to retrieve the value of each element at index `i`. Then the values will be output to the console:

```
13       for (int i=0; i<names.size(); i++) {
14          String name=names.get(i);
15          System.out.println("get("+i+")="+name);
16       }
17    }
18 }
```

When these lines of code are added to the program and executed, we can see the current contents of the names list in the console:

```
get(0)=David Conner
get(1)=Edith Sharf
get(2)=Charlie White
```