Collections (Part 1) tutorial

In this tutorial, you'll create and use a `List` collection object from the Java Collections library.

The `List` is an ordered, indexed data structure like an array. What sets it apart from an array is its flexibility. You can add elements to and remove elements from a `List` and not have to worry about the size, unlike an array.

To get started, import this project into IntelliJ. You'll write your code in the `src/main/java/com/techelevator/Tutorial.java` file.

In `Tutorial.java`, you'll see some comments where you can type your code for each step.

Step One: Declare a `List`

Find the first comment in `Tutorial.java`. You'll add your code after this line:

```
// Step One: Declare a List
```

When creating a new `List`, similarly to an array, you must specify what data type the `List` holds first.

To create an array, you specify the data type like this:

```
String[] nameList;
```

For `List`s, this is how you specify its data type:

```
List<String> nameList;
```

This specifies the new variable as being able to contain a `List` of `String`s. But you don't have a list yet. Like arrays, declaring a variable doesn't create it. You also need to create a new `List`.

Declare this variable and create a new `ArrayList` by putting this line of code after the first comment:

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

Recall what the student book said about *programming to an interface* and `ArrayList` fulfilling `List`'s contract.

This creates a new `ArrayList` of `String`s and assigns that new `ArrayList` to the `nameList` variable. Now you can work with the `nameList` variable.

Step Two: Add values to a `List`

Find the second comment in `Tutorial.java`. You'll add your code after this line:

```
// Step Two: Add values to a List
```

You now have a `List` of `String`s named `nameList`, but it's empty with a size of 0. You can add elements to it and expand it from its current size. In this step, you'll add the names of some famous programmers to the `List`.

To add an element to the end of the `List`, use the `add()` method of the `List` object. Add these lines after the second comment:

```
nameList.add("Ada");
nameList.add("Grace");
nameList.add("Margaret");
nameList.add("Adele");
```

Each time you call the `add()` method, the `List` takes the element passed to it, and adds it to the end of the `List`. Afterwards, the `List` looks like this:

```
[ "Ada", "Grace", "Margaret", "Adele" ]
```

Step Three: Loop through a `List` in a for loop

Find the third comment in `Tutorial.java`. You'll add your code after this line:

```
// Step Three: Looping through a List in a for loop
```

You can print out each of these elements using a for loop. `List` items are also indexed starting at `0`, like arrays.

If you wanted to loop through an array, your for loop would look like this:

```
for (int i = 0; i < nameArray.length; i++) {

}
```

When using a `List`, you can find out how many elements it contains with the `size()` method:

```
for (int i = 0; i < nameList.size(); i++) {

}
```

Then within the for loop, you can print out each element with its index. Use the `get()` method and the index to access each element, like you would with an array:

```
for (int i = 0; i < nameList.size(); i++) {
    System.out.println("The name at index " + i + " is " + nameList.get(i));
}
```

If you run your code now, you'll see the following:

```
The name at index 0 is Ada
The name at index 1 is Grace
The name at index 2 is Margaret
The name at index 3 is Adele
```

Step Four: Remove an item from a `List`

Find the fourth comment in `Tutorial.java`. You'll add your code after this line:

```
// Step Four: Remove an item
```

One of the biggest advantages of `List`s is that you can remove any item from a `List` at any time.

To remove an item from a `List`, you pass the list item to the `remove()` method:

```
nameList.remove("Ada");
```

If you don't know the value of the item, but you know you want to remove an item at a particular location in a `List`, you can use the `remove()` method, passing the index of the item you want removed:

```
nameList.remove(0);
```

Choose *one* of the methods—passing the list item or passing the index of the item—and place it after the fourth comment. You'll confirm that the item was removed in the next step.

Step Five: Loop through `List` in a foreach loop

Find the fifth comment in `Tutorial.java`. You'll add your code after this line:

```
// Step Five: Looping through List in a foreach loop
```

foreach loops aren't limited to Collections objects. You can use them with arrays, too.

One advantage of using the foreach loop is that you don't need to keep track of an index. Instead, the foreach loop assigns the current item to a temporary variable that's available only in the loop.

You define a foreach loop like this:

```
for (String name : nameList) {

}
```

This code means, "for each name in the nameList." It's important to define the temporary variable—`String name` in this case—as having the same data type as the collection.

Now you can print `name` as the loop runs:

```
for (String name : nameList) {
    System.out.println("Name: " + name);
}
```

If you run your code again, you'll notice that "Ada", or whichever item you chose, is no longer printed because you removed that value in step four:

```
Name: Grace
Name: Margaret
Name: Adele
```

Summary

After completing this tutorial, you understand:

- The differences between a `List` and an array.
- The common operations of a `List` and how to use them.
- How to use the foreach loop to iterate through a collection.