Working with Lists in Java

Duke Hutchings (dhutchings@elon.edu)

In our previous work we learned about how to use loops to allow a user to enter a sequence of values and then report on the results. We can use loops to count the number of values, sum up a sequence of numbers, and indicate the smallest or largest value. But with the knowledge we have so far, there are some things we cannot do with loops. For example, we cannot take in *all* the values as input and print the values in sorted order.

The concept of a *list* allows us to record all of the values. Colloquially we use the term *list* to refer to a collection of items but in Java we have two important restrictions:

- every item in a list must have the same data type (or be an object of the same type)
- each item in a list has a unique *index* (in other words, in Java, the list is a strict sequence of items)

Many lists we create on paper contain a variety of information and might mix words, numbers, pictures, etc. In Java, each piece of information in the list must be of the same type. We can have lists of numbers, or lists of words, or lists of pictures. We cannot have a list with numbers and words and pictures (at least, not until we get into the upper-levels of computer science courses).

Also, many lists we create on paper might have items all over the paper and not in any particular order. In Java, the items are strictly ordered and, like characters in a `String`, the first item in a list is at index 0.

Java allows many list actions. We will focus on seven fundamental actions:

- create a new list
- add an item to the end of a list
- add an item in the middle or beginning of a list
- get an item from a list at a particular index
- remove an item from a list at a particular index
- determine the number of items in a list (the list's *size*)
- sort a list

Let's take a look at some examples of these actions starting on the next page.

Examples of Selected List Actions

Action	**Result**
Create a new list named *groceries*	groceries
Add *milk* as an item to the *groceries* list	groceries 0. milk
Add *cereal* as an item to the *groceries* list	groceries 0. milk 1. cereal
Add *half and half* to the *groceries* list at index 0. (Notice that the index of all other items changed)	groceries 0. half and half 1. milk 2. cereal
Add *granola* to the *groceries* list at index 2.	groceries 0. half and half 1. milk 2. granola 3. cereal
Add *peaches* to the *groceries* list at index 2.	groceries 0. half and half 1. milk 2. peaches 3. granola 4. cereal
Remove the item at index 3 in *groceries*.	groceries 0. half and half 1. milk 2. peaches 3. cereal
Remove the item at index 0 from *groceries*. (Notice that the index of all other items changed)	groceries 0. milk 1. peaches 2. cereal
Sort the *groceries* list.	groceries 0. cereal 1. milk 2. peaches

Observations to make

- When we make a list, it starts empty
- We can take only one action at a time. To add 2 things to a list, we add one thing, then add another.
- The was not an example of getting an item. We'll see that later.
- There was not an example of asking for the size of the list. We'll see that later too.

Let's now look at how lists work in Java code.

Creating a New List

Recall from Sprint 1 the concept of an *object*. Objects model multiple pieces of data and offer methods to get information from and make changes to the object. In Java, lists consist of a sequence of individual objects. But the list itself is also an object, so it is created in a similar fashion to any other object. Java offers many kinds of lists; the type we will use is `ArrayList`. For example:

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

The code above creates a list of `String` objects. The name of the list is `words`. Let's say we wanted to create a list of birth dates for students in a class. We could use the `Day` object we used in Sprint one.

```
ArrayList<Day> birthDates = new ArrayList<Day>();
```

Or maybe we would like a list of images for a slide show...

```
ArrayList<Picture> slideShow = new ArrayList<Picture>();
```

For whatever type of list we want, we have a generic syntax to invoke the `ArrayList` constructor:

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

When created, lists have no items. Conceptually, we are designating a particular blank piece of paper as our list and putting the list title at the top, as exemplified earlier. The act of writing items on the paper corresponds to the two `ArrayList` methods available to add items to the list.

Adding Items to the End of a List

The `add` method allows items to be added to a list. There are two versions of add. One version adds an item to the end of a list while the other version adds an item elsewhere in a list. This second version requires that the index at which to add the item be specified. We'll inspect this second version later.

Here's sample code that allows a user to type in words one at a time until entering *q* to stop and stores each word in a list. The blue section below shows sample inputs and output from running this code.

```
Scanner in = new Scanner(System.in);
String m = "Enter a word or q to quit: ";

ArrayList<String> terms = new ArrayList<String>();

System.out.print(m);
String word = in.nextLine();
while (!word.equals("q"))
{
  terms.add(word);
  System.out.println("   Now the list is: " + terms);
  System.out.println();
  System.out.print(m);
  word = in.nextLine();
}
System.out.println("Final List: " + terms);
```

```
Enter a word or q to quit: this
   Now the list is [this]

Enter a word or q to quit: is
   Now the list is [this, is]

Enter a word or q to quit: a
   Now the list is [this, is, a]

Enter a word or q to quit: test
   Now the list is [this, is, a, test]

Enter a word or q to quit: q
Final List: [this, is, a, test]
```

At the end of the code above, `terms` is a list consisting of each word typed by the user. When printing the list to the screen, we see it indicated in square brackets with each item separated by a comma. Note that the list is **not** a `String`. However, Java is able to print out each of the list items to the screen.

We will return to the discussion of adding items to a list in a moment. Let's take a look now at getting items from a list (which will also require us to look at how to determine the *size* of a list).

Determining the Size of a List and Getting Items from a List

The `get` method allows items to be acquired from a list. We must be careful here about language and the meaning of *get*. In Java *getting* an item generally means acquiring a copy of a value without affecting anything else. In particular with lists, *getting* an item from a list does **not** take it out of the list. Rather, it allows for an inspection of the item.

If we wish to get *every* item from a list, we will need to know how big the list is. Java uses the term *size*.

Here's sample code that shows how to acquire each item in a list using the `get` and `size` methods. Suppose that `terms` is the same list created in the previous example.

```
System.out.println("Reminder... list is: " + terms);

for (int i = 0; i < terms.size(); i++)
{
  String word = terms.get(i);
  System.out.println(word + " at index " + i);
}

int listSize = terms.size();
System.out.println("Size of list: " + listSize)
```

```
Reminder... list is: [this, is, a, test]
this at index 0
is at index 1
a at index 2
test at index 3
Size of list: 4
```

Notice how getting the items from the list did not change the size of the list. Also notice that, like `String`, items in a list are indexed starting at 0, not 1. Now that we know a little about indexing, let's return to our discussion of the add method and look at how to add items at a particular index rather than at the end.

Adding Items to a List at a Specific Index

We previously saw that the add method adds an item to the end of a list. Another version of the add method allows the adding of an item at a specific index in the list. Let's look at some examples.

```
System.out.println("Reminder... list is: " + terms);

terms.add(0, "mario");
System.out.println();
System.out.println("Added mario at index 0");
System.out.println("Now the list is: " + terms);

terms.add(3, "not");
System.out.println();
System.out.println("Added not at index 3");
System.out.println("Now the list is: " + terms);

terms.add(6, "flower");
System.out.println();
System.out.println("Added flower at index 6");
System.out.println("Now the list is: " + terms);
```

```
Reminder... list is: [this, is, a, test]

Added mario at index 0
Now the list is: [mario, this, is, a, test]

Added not at index 3
Now the list is: [mario, this, is, not, a, test]

Added flower at index 6
Now the list is: [mario, this, is, not, a, test, flower]
```

Notice how you can still add items to the end of the list by specifying the index after the last index in the list (above, the last item in the list was at index 5 but we were able to add *flower* at index 6). If you try to add an item at an index beyond the size of the list or try to add an item at a negative index, your program will crash.

Removing Items from a List at a Specific Index

We can remove items at a specific index from a list.

```
System.out.println("Reminder... list is: " + terms);

terms.remove(0);
System.out.println();
System.out.println("Removed item at index 0");
System.out.println("Now the list is: " + terms);

terms.remove(2);
System.out.println();
System.out.println("Removed item at index 2");
System.out.println("Now the list is: " + terms);

terms.remove(4);
System.out.println();
System.out.println("Removed item at index 4");
System.out.println("Now the list is: " + terms);
```

```
Reminder... list is: [mario, this, is, not, a, test, flower]

Removed item at index 0
Now the list is: [this, is, not, a, test, flower]

Removed item at index 2
Now the list is: [this, is, a, test, flower]

Removed item at index 4
Now the list is: [this, is, a, test]
```

Notice how removing an item from a list affects the index of all items in the list *after* the item was removed. Because of this feature, removing all items from a list that meet a certain criteria should be accomplished with a *for* loop that iterates through the list *backwards*. On the next page we see an example where we remove all of the items in the list that contain the letter *e*.

Example <u>with Mistake</u>: Removing All the Words with *e*

This is how ***<u>not</u>*** to remove all the items from a list of `String` objects that have an *e*.

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

people.add("mario");
people.add("luigi");
people.add("peach");
people.add("bowser");
people.add("yoshi");
System.out.println("The list is: " + people);
System.out.println();

for (int i = 0; i < people.size(); i++)
{
  String name = people.get(i);
  if (name.contains("e"))
  {
    people.remove(i);
    System.out.print("--Removed " + name + " at index " + i);
    System.out.println(" -- The list is now: " + people);
  }
  else
  {
    System.out.println(name + " at index " + i + " has no e");
  }
}
System.out.println("The final list is: " + people);
```

```
The list is: [mario, luigi, peach, bowser, yoshi]

mario at index 0 has no e
luigi at index 1 has no e
-- Removed peach at index 2 -- The list is now: [mario, luigi, bowser, yoshi]
yoshi at index 3 has no e
The final list is [mario, luigi, bowser, yoshi]
```

When we found that *peach* had an *e*, we removed it. But this also changed the indexes of the names that come after. So *bowser* was at index 3 and moved to index 2 and *yoshi* was at index 4 and moved to index 3. The looping variable *i* doesn't know anything about this, so it just moves from its current value of 2 onto 3. In effect we skip over *bowser*.

There are several ways to counter this effect. Perhaps the easiest way is to go backwards through the list.

Example with Correction: Removing All the Words with *e*

This is one way to remove all the items from a list of `String` objects that have an *e*.

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

people.add("mario");
people.add("luigi");
people.add("peach");
people.add("bowser");
people.add("yoshi");
System.out.println("The list is: " + people);
System.out.println();

for (int i = people.size() - 1; i > -1; i--)
{
  String name = people.get(i);
  if (name.contains("e"))
  {
    people.remove(i);
    System.out.print("--Removed " + name + " at index " + i);
    System.out.println(" -- The list is now: " + people);
  }
  else
  {
    System.out.println(name + " at index " + i + " has no e");
  }
}
System.out.println("The final list is: " + people);
```

```
The list is: [mario, luigi, peach, bowser, yoshi]

yoshi at index 4 has no e
-- Removed bowser at index 3 -- The list is now: [mario, luigi, peach, yoshi]
-- Removed peach at index 2 -- The list is now: [mario, luigi, yoshi]
luigi at index 1 has no e
mario at index 0 has no e
The final list is [mario, luigi, yoshi]
```

Every single line of code stayed the same except for the for loop. By looping backward, we encounter *bowser* as the first item that has an *e*. When we remove it, the indexes of the list items prior to *bowser* are unaffected and since the looping variable *i* is decreasing, we will be sure to visit all of the indexes.

Sorting a List

A common list action is sorting. Java makes this pretty easy.

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

people.add("mario");
people.add("luigi");
people.add("peach");
people.add("bowser");
people.add("yoshi");
System.out.println("The list is: " + people);

people.sort(null);
System.out.println("The sorted list is: " + people);
```

```
The list is: [mario, luigi, peach, bowser, yoshi]
The sorted list is: [bowser, luigi, mario, peach, yoshi]
```

Sorting is most commonly done when we wish to determine the *nth* largest or *nth* smallest item in a list. The smallest item is at index 0 after a sort. The second smallest item is at index 1, etc. The largest item is at an index of one less than the size of the list. The second largest item is at index of two less than the size of the list, etc.

You might be wondering about the word `null` in the call to `sort`. The ArrayList API documentation states that a value must be used to help with the sorting of a list and that `null` can be provided to sort the list in a customary fashion (here, `String` objects sorted in alphabetical order). If you move onto CS2, you will learn about the code necessary to sort the list in other ways.

Two Common List Actions: Count and Sublist

Given a list of items we might want to know *how many* items meet certain criteria or we might want a *new* list only containing items that meet the criteria. The approach to both problems is the same. Only a few details differ. Below is an example of each.

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

// Count the e words
int count = 0;
for (int i = 0; i < list.size(); i++)
{
  String word = list.get(i);
  if (word.contains("e"))
  {
    count++;
  }
}
System.out.print("Result: ");
System.out.println(count);
```

```
[mario, luigi, peach, bowser, yoshi]
Result: 2
```

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

// Make a new list of the e words
ArrayList<String> eWords = new ArrayList<String>();
for (int i = 0; i < list.size(); i++)
{
  String word = list.get(i);
  if (word.contains("e"))
  {
    eWords.add(word);
  }
}
System.out.print("Result: ");
System.out.println(eWords);
```

```
[mario, luigi, peach, bowser, yoshi]
Result: [peach, bowser]
```

Also notice that in the code on the right, we have simultaneously solved the problem on the left: since we have a list of all of the *e*-words, we can use the `size` method to determine how many words there are.

Lists of Numbers

A key restriction on ArrayList is that the data type of the list must be a class and in particular cannot be a native data type. The code lines below are all compilation/syntax errors.

```
ArrayList<int> grades = new ArrayList<int>();
ArrayList<double> rates = new ArrayList<double>();
```

To overcome this issue, Java provides *wrapper* classes that are technically objects, but store only one value. The code lines below are the correct ways to implement lists of the different native data types.

```
ArrayList<Integer> grades = new ArrayList<Integer>();
ArrayList<Double> rates = new ArrayList<Double>();
```

The `Integer` class stores a single `int` variable. We need only use the `Integer` name when we create the list. After that, we can use `int` variables as usual. Java will do the behind the scenes work to convert to/from an object.

An example appears on the next page.

Example: List of Numbers

```
Scanner in = new Scanner(System.in);
String m = "Enter a grade or -1 to quit: ";

ArrayList<Integer> grades = new ArrayList<Integer>();

System.out.print(m);
int score = in.nextInt();
while (score > -1)
{
  grades.add(score); // score is an int, not an Integer.  Java converts for you.
  System.out.print(m);
  score = in.nextInt();
}

for (int i = 0; i < grades.size(); i++)
{
  int item = grades.get(i); // Java converts the Integer returned by get to an int
  System.out.println("Item at index " + i + " is " + item);
}

grades.sort(null);
System.out.println("Sorted grades: " + grades);
```

```
Enter a grade or -1 to quit: 8
Enter a grade or -1 to quit: 6
Enter a grade or -1 to quit: 7
Enter a grade or -1 to quit: 5
Enter a grade or -1 to quit: 3
Enter a grade or -1 to quit: 0
Enter a grade or -1 to quit: 9
Enter a grade or -1 to quit: -1

Item at index 0 is 8
Item at index 1 is 6
Item at index 2 is 7
Item at index 3 is 5
Item at index 4 is 3
Item at index 5 is 0
Item at index 6 is 9

Sorted grades: [0, 3, 5, 6, 7, 8, 9]
```

The Enhanced For Loop

The action of getting each item in a list from the beginning to the end is so common that Java offers a compact syntax for the for loop. In this compact syntax you are guaranteed to get each item but you don't necessarily know where the item is. Consequently, this compact syntax can be used only to get each item and cannot be used to alter (that is, add to or remove from) the list.

On the left, we we the usual syntax. On the right, the compact syntax. Both sets of code calculate the average of the list values.

```
int sum = 0;
int amt = grades.size();
for (int i = 0; i < amt; i++)
{
  int item = grades.get(i);
  sum = sum + item;
}
double average = (double)sum / amt;
```

```
int sum = 0;
int amt = grades.size();
for (int item : grades)
{
  // don't call get; handled by loop syntax
  sum = sum + item;
}
double average = (double)sum / amt;
```