SIENA*college*
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2019

Topic Notes: The `ArrayList`

Arrays are a very common method to store a collection of similar items.

Arrays work very well for a lot of situations, but they come with some very important restrictions.

- their size is specified on construction, and cannot be changed without constructing a new array and copying over the contents
- all array indices must be managed explicitly
- if you want to insert an item at the start of or in the middle of an array, you need to move one or more items out of the way to make room
- if you remove an item from the start or the middle of an array and you don't want to leave a "hole" in the middle, one or more items needs to be moved around to fill in the hole

This idea of a dynamically resizeable (or, *extensible*) array leads naturally to the idea of the *arraylist* ADT, also known as a *vector*.

What kinds of operations would we like to have on something that behaves like a resizeable array?

We need the functionality of a regular array:

- construction
- add an item to the end
- insert an item in the middle
- retrieve value of an element
- remove an item

Most of the operators of an arraylist will assume that the elements are "packed" – that is:

- if we add an element, it will be added to the end by default
- if we add an element in the middle, all elements with higher subscripts are moved up to make room
- if we remove an element, all elements with higher subscripts are shifted down to fill in the space

And of course, we will want it to resize itself to have enough space for as many elements as we add.

We will look first at how such a structure as provided by the Java API can be used, then will consider how it works.

The Java `ArrayList` Class

As you continue to expand your programming skills, you will learn about a variety of ways that collections of data can be stored that vary in complexity, flexibility, and efficiency. The first of these structures that we will consider is the `ArrayList`.

`ArrayList` is a class that implements an *abstract data type (ADT)* provided by the standard Java utility library. We will look more closely later at what it means for the `ArrayList` to be an ADT.

Let's see how to use them through an example: an enhanced version of the "PurchaseTracker" example with an `ArrayList` that holds all of the `PurchasedItem` objects we create.

See Example: PurchaseTrackerAll

We consider each change that was made to the program to see the basic usage of an `ArrayList`.

- First, we need to add an `import` statement to the top of our program.

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

  This allows us to use the class name `ArrayList` in the rest of the file and Java will know we mean to use the one in the `java.util` package.

- Next, we declare a local variable in `main` for our `ArrayList` and construct an instance:

  ```
  ArrayList<PurchasedItem> items = new ArrayList<PurchasedItem>();
  ```

  Since an `ArrayList` is a generic structure that can be used to hold objects of any type, we need to tell Java what type of objects will be stored in this particular `ArrayList`. In this case, it's `PurchasedItems`. So we specify this as a type parameter both in the variable declaration and the construction.

- The `PurchasedItem` instances are then created, and we need to insert each into the `ArrayList`. This is done with the `add` method:

  ```
  items.add(item);
  ```

  This will take the `PurchasedItem` named `item` and add it to the first available slot in the `ArrayList` named `items`.

Note that in this case, we are not requesting any specific location within the `ArrayList` for the item. We will later see that we can be more specific here.

Note also that we as users of the `ArrayList` do not know (though when you take data structures, you'll have a pretty good idea) of what's going on inside the `ArrayList` to add the item. We just know that it knows how to do it.

When we're done with the `do..while` loop, the `ArrayList` contains references to all of the `PurchasedItem` objects we constructed.

- In the rest of the `main` method, we need to access the `PurchasedItem` objects within the `ArrayList`. We do this with the `get` method:

  ```
  item = items.get(0);
  ```

  in the middle of the method gives us a reference to the first `PurchasedItem` that we had added to the `ArrayList`.

  We then see a `for` loop that uses `itemNum` is a loop index variable that will range from 1 to one less than the number of items in the `ArrayList`. How many items are there? We can get that information from the `ArrayList` itself using the `size` method.

  What we see here is that the `ArrayList` has assigned a number, often called an *index*, to each `PurchasedItem` we added to the `ArrayList`, and we can pass that number to the `get` method to get back a specific `PurchasedItem` from the `ArrayList`.

  This is our good example of a *search* operation on a collection – we are looking through each object in the collection to find ones that are the "winners" in each category. More precisely, this is a *linear search* and we will say more about this later.

One of the great things about using a construct like an `ArrayList` is that we can extend our programs to keep track of a much larger number of objects. No matter how many items we enter into the program (within the bounds of our computer's memory resources, at least) we can use a collection like an `ArrayList` to keep track of them.

For a second example, consider the use of an `ArrayList` of `Association` objects:

SpellsArrayList – to be developed in class.

Again, we construct an `ArrayList` and `add` items to it. In this case, `Associations` which use `String` objects for both key and value.

There is just one `ArrayList` method here that was not in the previous: `indexOf`. This one searches through the `ArrayList` for an object that is equivalent (by the `equals` method) to the one passed as a parameter. It returns the index (position within the `ArrayList`) of the first match. If no match exists, it returns -1.

Note here that we make use of the fact that for two `Associations` to be considered equal, their keys must match, but their values do not.

Contrast this with the same program using primitive arrays instead:

See Example: SpellsArray

- Our variable declaration looks a bit different.
- When we construct the array in the `main` method, we need to tell it how many elements the array will hold (in this case, 10). With the `ArrayList`, we construct a list and we can add as many things to it as we want. The array can only ever hold the number of elements we provided when we constructed it.
- When we add items to the array, we need to specify the index explicitly. There is no way to say "just add it to the end" the way we do with `ArrayList`s.
- When we access array elements, we use the bracket notation in much the same way we use the `get` method of the `ArrayList`.
- The array remembers how many entries it contains, and we can access this information with the `.length`. This plays the role of the `size` method of the `ArrayList`.

Other `ArrayList` methods

The examples above demonstrated just a few of the capabilities of the `ArrayList` class: construction, `add`, `get`, and `size`.

The full documentation for the `ArrayList` can be found on Oracle's Java documentation site:

Java API Documentation: ArrayList at
`http://docs.oracle.com/javase/8/docs/api/util/ArrayList.html`

Here are a couple of additional methods, some of which will come up in later examples.

- `remove` – remove an object from the list
- `clear` – remove all objects from the list
- `contains` – determine if a given object is in the list
- `set` – replace the contents at an index with a new element

`ArrayList`s of Primitive Types

Java places a significant restriction on the use of primitive types as the type parameters for generic data structures such as the `ArrayList`. The following would not be valid Java:

```
ArrayList<int> a = new ArrayList<int>();
```

The type in the <> must be an object type. Fortunately, Java provides object types that correspond to each primitive type. An `Integer` object is able to store a single `int` value, a `Double` value is able to store a single `double` value, etc. So the declaration and construction above would need to be:

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

In older versions of Java, programmers would need to be careful to convert back and forth between values of the primitive types and their object encapsulators. To construct an `Integer` from an `int i`, one would need to do so explicitly:

```
a.add(new Integer(i));
```

And to retrieve the `int` value from an `Integer`, one would also do so explicitly:

```
a.get(pos).intValue();
```

However, recent versions of Java automatically convert between the primitive types and their object encapsulating classes. This is called *autoboxing* when converting from primitive to "boxed" encapsulating classes, and *autounboxing* when going back the other way.

However, the effective programmer should always keep in mind that these conversions are occurring, as there is a computational cost to each.

Another Example

Suppose we have an `ArrayList` of Integer values, and someone (by a mechanism which is not our concern) has asked us to write a method that will find the largest value in the `ArrayList`. The following method will achieve this (we assume at least one element in the `ArrayList`):

```
private static int findMax(ArrayList<Integer> a) {

  int max = a.get(0);
  for (int i=1; i<a.size(); i++) {
    int val = a.get(i);
    if (val > max) max = val;
  }
  return max;
}
```

The Enhanced `for` Loop

We have seen that a common task with a collection such as an `ArrayList` is to *iterate* over its contents. That is, "visit" every element in the list exactly once to do something to it.

It is often the case (and was in many of the examples here) that the specific index of an entry in an `ArrayList` is not important as we are iterating over its contents.

In these cases, the counting `for` loops can be replaced with a related Java construct often called the *enhanced* `for` *loop*, or a "foreach" loop.

If we have an `ArrayList` of objects of some type `T` and we wish to loop over all entries in the loop, we can replace a counting loop:

```
ArrayList<T> a = new ArrayList<T>();

...

for (int i = 0; i < a.size(); i++) {
  T item = a.get(i);
  // do something with item
}
```

with an enhanced `for` loop:

```
ArrayList<T> a = new ArrayList<T>();

...

for (T item : a) {
  // do something with item
}
```

This construct will loop enough times so that the variable `item` will be assigned to each entry in `a` exactly once through the body of the loop.

The enhanced `for` construct is not always appropriate, however. For example, in the `findMax` method above, it is more convenient to be able to get the item at position 0 as the initial "max" and then loop over the entries from positions 1 and up to check for larger values.

As you learn more Java, you will see a number of other data structures that can be used with the for-each loop construct.

An `ArrayList` Within a Custom Class

It may or may not have become clear so far that you can use `ArrayLists` in pretty much any context that you can use other data types. This includes as an instance variable in a custom class.

See Example: CourseGrades

In the above example, `ArrayLists` are used to keep track of a list of students and course grades, and within the class that represents one student's information, the list of the grades.