Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Collections

Our next major topic involves common mechanisms for naming collections of items.

Motivation for Collections

Sometimes we have a lot of very similar data, and we would like to do similar things to each datum.

For example, suppose we wanted to extend our "PurchaseTracker" example to remember all of the `PurchsedItem` objects we create, to be printed out at the end.

We would need names for all of those objects. Since we don't know how many there would be, we don't know how many names to declare.

Java and other programming languages provide a number of mechanisms to help here. We will consider two in Java. First, we will look at a Java class called the `ArrayList`, and later a lower-level construct common to most modern programming language called *arrays*. Each allows us to use one name for an entire collection of objects.

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. We will consider just one of those structures to start: the `ArrayList`.

`ArrayList` is a class that implements an *abstract data type* provided by the standard Java utility library.

Let's see how to use them through an example: we will enhance 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 `PurchasedItem`s. 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 first 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:

See Example: Spells

Again, we construct an `ArrayList` and `add` items to it. In this case, `Association`s 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 `Association`s to be considered equal, their keys must match, but their values do not.

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 at `http://docs.oracle.com/javase/7/docs/api/`.

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 "for-each" 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 *for-each* 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 a for-each 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 for-each 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 `ArrayList`s 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, which you will expand as part of your next lab, `ArrayList`s 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.

Java Arrays

The `ArrayList` is a Java class, provided as a standard utility with every Java environment. But it is built on top of a more fundamental programming language construct called an *array*.

In mathematics, we can refer to large groups of numbers (for example) by attaching subscripts to names. We can talk about numbers n_1, n_2,... An array lets us do the same thing with computer languages.

Suppose we wish to have a group of elements all of which have type `ThingAMaJig` and we wish to call the group `things`. Then we write the declaration of `things` as

```
ThingAMaJig[] things;
```

The only difference between this and the declaration of a single item of type `ThingAMaJig` is the occurrence of "`[ ]`" after the type.

Like all other objects, a group of elements needs to be created:

```
things = new ThingAMaJig[25];
```

Again, notice the square brackets. The number in parentheses (25) indicates the number of slots to create, each of which can hold one of the elements. We can now refer to individual elements using subscripts. However, in programming languages we cannot easily set the subscripts in a smaller font placed slightly lower than regular type. As a result we use the ubiquitous "`[ ]`" to indicate a subscript. If, as above, we define `things` to have 25 elements, they may be referred to as:

```
things[0], things[1], ..., things[24]
```

We start numbering the subscripts at `0`, and hence the last subscript is one smaller than the total number of elements. Thus in the example above the subscripts go from 0 to 24.

One warning: When we initialize an array as above, we only create slots for all of the elements, we do not necessarily fill the slots with elements. Actually, the default values of the elements of the array are the same as for instance variables of the same type. If `ThingAMaJig` is an object type, then the initial values of all elements is `null`, while if it is `int`, then the initial values will all be `0`. Thus you will want to be careful to put the appropriate values in the array before using them (especially before sending message to them! – that's a `NullPointerException` waiting to happen).

In many ways, and array works like an `ArrayList`, but we will see several differences.

For example:

See Example: SpellsArray

Notice how this differs from the `ArrayList` version.

- 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`.

Another example:

See Example: GradeRangeCounter

This one includes examples of arrays declared and initialized as `final`, and an example of an array of `int` allocated with `new`.

Inserting and Removing with Arrays

There is quite a bit to keep track of when using arrays, especially when objects are being added. We need to manage both the size of the array and the number of items it contains. If it fills, we either need to make sure we do not attempt to add another element, or reconstruct the array with a larger size.

As a wrapup of our initial discussion of arrays, let's consider two more situations and how we need to deal with them: adding a new item in the middle of an array, and removing an item from the end.

For these examples, we will use arrays of numbers. Arrays can store numbers just as well as they can store references to objects.

Suppose we have an array of `int` large enough to hold 20 numbers.

The array would be declared as an instance variable:

```
private int[] a;
```

along with another instance variable indicating the number of `int`s currently stored in `a`:

```
private int count;
```

and constructed and initialized:

```
a = new int[20];
count = 0;
```

At some point in the program, `count` contains 10, meaning that elements 0 through 9 of `a` contain meaningful values.

Now, suppose we want to add a new item to the array. So far, we have done something like this:

```
a[count] = 17;
count++;
```

This will put a 17 into element 10, and increment the `count` to 11.

But suppose that instead, we want to put the 17 into element 5, and without overwriting any of the data currently in the array. Perhaps the array is maintaining the numbers in order from smallest to largest.

In this case, we'd first need to "move up" all of the elements in positions 5 through 9 to instead be in positions 6 through 10, add the 17 to position 5, and then increment `count`.

If the variable `insertAt` contains the position at which we wish to add a new value, and that new value is in the variable `val`:

```
for (int i=count; i>insertAt; i--) {
   a[i] = a[i-1]
}
a[insertAt] = val;
count++;
```

Now, suppose we would like to remove a value in the middle. Instead of "moving up" values to make space, we need to "move down" the values to fill in the hole that would be left by removing the value.

If the variable `removeAt` contains the index of the value to be removed:

```
for (int i=removeAt+1; i<count; i++) {
  a[i-1] = a[i];
}
count--;
```

The loop is only necessary if we wish to maintain relative order among the remaining items in the array. If that is not important (as is often the case with our graphical objects), we might simply write:

```
a[removeAt] = a[count-1];
count--;
```

In circumstances where we are likely to insert or remove into the middle of an array during its life-time, it usually makes sense to take advantage of the higher-level functionality of the `ArrayList`.

Array and `ArrayList` Summary

The following list summarizes the key differences and similarities between arrays and `ArrayList`s.

Declaration To declare an array of elements of some type `T`:

```
T[] ar;
```

where `T` can be any type, including primitive types or Object types.

And to declare an `ArrayList` that can hold items of type `T`:

```
ArrayList<T> al;
```

where `T` must be an object type. If we want to store a primitive type, we must use Java's corresponding object wrappers (e.g., `Integer` when we want to store items of type `int`).

Construction To construct (allocate space for) our array of `n` elements of type `T`:

```
ar = new T[n];
```

Once constructed, the array will always have space for `n` elements of type `T` – if we want a larger or smaller array, we would have to construct a new one.

The array constructed will have the default value for the datatype stored in each entry. For object types, all entries begin as `null`. For primitive number types, they begin as 0. For `boolean` arrays, they begin as `false`.

To construct an `ArrayList`:

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

This `ArrayList` initially does not contain any values. Its size will be determined by the number of elements we add to it.

Adding an Element To add an element to an array, we have to specify the position at which we wish to add the new element:

```
ar[i] = t;
```

This will place the item `t` at position `i` into our array. `i` must be in the range 0 to `n`-1 if we constructed our array to have `n` entries. If there was already some data stored in position `i`, it will be overwritten with `t`.

If we want to add the item to the "end" of the array, that is, the first unoccupied slot in the array, we will need an additional variable to keep track of the number of currently-occupied slots. If this is called `aSize`, and we have been careful to make sure the `aSize` elements in the array occupy slots 0 through `aSize`-1, we can add the element with:

```
ar[aSize] = t;
aSize++;
```

With an `ArrayList`, the `add` method takes care of this:

```
al.add(t);
```

Retrieving an Element To get an item from an array, we use the same notation. To put the value from position `i` in the array into some variable `t`:

```
t = ar[i];
```

Whereas with the `ArrayList`, we need to call a method:

```
t = al.get(i);
```

Visiting All Elements To loop over all elements in the array:

```
for (int i=0; i<aSize; i++) {
  t = ar[i];
  // do something with t
}
```

and an `ArrayList`;

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

In both cases, we can also use the for-each loop.

Two-Dimensional Arrays

We can create arrays to hold objects of any type, either basic data types like `int` and `double`, or instances of objects.

Nothing stops us from defining arrays of arrays. To declare an array, each of whose elements is an array of `int`:

```
int[][] twoDArray;
```

While it is normally written without parentheses, we can think of the above declaration as defining `twoDArray` as having type `(int []) []`. Thus each element of `twoDArray` is an array of `int`s.

Despite the fact that Java will treat this as an array of arrays, we usually think about this as a two-dimensional array, with the elements arranged in a two-dimensional table so that `twoDArray[i][j]` can be seen as the element in the ith row and jth column. For example here is the layout for a two-dimensional array `a` with 6 rows (numbered 0 to 5) and 4 columns:

	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]
4	a[4][0]	a[4][1]	a[4][2]	a[4][3]
5	a[5][0]	a[5][1]	a[5][2]	a[5][3]

Viewed in this way, our two-dimensional array is a grid, much like a map or a spreadsheet. This is a natural way to store things like tables of data or matrices.

We access elements of two-dimensional arrays in a manner similar to that used for one dimensional arrays, except that we must provide both the row and column to access an element, giving the row number first.

We create a two-dimensional array by providing the number of rows and columns. Thus we can create the two-dimensional array above by writing:

```
int[][] a = new int[6][4];
```

(Though as good programmers, you would define constants for the number of rows and the number of columns.)

A nested `for` loop is the most common way to access or update the elements of a two-dimensional array. One loop walks through the rows and the other walks through the columns. For example, if we wanted to assign a unique number to each cell of our two-dimensional array, we could do the following:

```
for (int row = 0; row < 6; row++) {
    for (int col = 0; col < 4; col++) {
        a[row][col] = 4*row + col + 1;
    }
}
```

This assigns the numbers 1 through 24 to the elements of array `a`. The array is filled by assigning values to the elements in the first row, then the second row, etc. and results in:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
```

And if we wanted to print the above, we can write a loop:

```
for (int row = 0; row < 6; row++) {
    for (int col = 0; col < 4; col++) {
        System.out.print(a[row][col] + " ");
    }
    System.out.println();
}
```

You could modify the above to be slightly more interesting by computing a multiplication table.

We could just as well process all the elements of column 0 first, then all of column 1, etc., by swapping the order of our loops:

```
for (int col = 0; col < 4; col++)
    for (int row = 0; row < 6; row++)
        ...
```

For the most part, it doesn't matter which order you choose, though for large arrays it is generally a good idea to traverse the array in the same order that your programming language will store the values in memory. For Java (and C, C++), the data is stored by rows, known as *row major* order. However, an two-dimensional array in FORTRAN is stored in *column major* order. You will almost certainly see this again if you go on and take courses like Computer Organization or Operating Systems.

Two-Dimensional Matrices

A very common use of two-dimensional arrays is the representation of matrices. We will look at an example of a class that represents two-dimensional square matrices and provides some basic operations on them.

See Example: Matrix2D

The class is capable of holding a square matrix of `double` values of any positive dimension.

Comments within the example explain much of what is happening. Note in particular the use of the two-dimensional array as an instance variable which stores the matrix entries, the use of exceptions to handle error conditions, and the `main` method that tests out the methods of the class. We will be seeing much more about exception handling soon.