Computer Science 120
Introduction to Programming
Siena College
Spring 2012

Topic Notes: Collections

Our next major topic involves naming collections of items. But first, we will look at a loop construct that we will often make use of in that context.

`for` Loops

We have used `while` loops in a number of contexts, one of which is for counting. For example, in the falling snow example, we had the following `run` method:

```
int snowCount= 0;

// continue creating snow until the maximum amount
// has been created
while (snowCount < MAX_SNOW ) {

    snowCount = snowCount + 1;

    new FallingSnow(canvas, snowPic,
                    snowGen.nextValue(),   // x coordinate
                    snowGen.nextValue()*2/canvas.getWidth()+2); // y sp
    pause(FLAKE_INTERVAL);
}
```

If we carefully examine the loop in the falling snow example above, we can see that it has the following structure:

```
int counter = 0;
while (counter < stopVal)
{
    // do stuff
    counter++;
}
```

It turns out that we can use a different construct that localizes the code dealing with counting so that it is easier to understand. This construct is called a `for` loop. You would use it for counting by saying the following:

```
for (int counter = 0; counter < stopVal; counter++)
{
   // do stuff - but omit counter++ at end
}
```

The code in the parentheses consists of 3 parts; it is not just a condition as in `if` or `while` statements. The parts are separated by semicolons. The first part is executed once when we first reach the `for` loop. It is used to declare and initialize the counter. The second part is a condition, just as in `while` statements. It is evaluated before we enter the loop and before each subsequent iteration of the loop. It defines the stopping condition for the loop, comparing the counter to the upper limit. The third part performs an update. It is executed at the *end* of each iteration of the `for` loop, just before testing the condition again. It is used to update the counter.

How would we rewrite the falling snow example to use a `for` loop?

See Example: FallingSnowFor

Essentially we have taken three lines from the above `while` loop version and combined them into one line of the `for` loop version. Because we included the declaration of the counter inside the loop (see "`int snowCount`"), it is only available inside the loop. If you try to use it outside of the loop, Java will claim to have never heard of a variable with that name.

Notice how the for localizes the use of the counter. This has two benefits. First, it simplifies the body of the loop so that it is somewhat easier to understand the body. More importantly, it becomes evident, in one line of code, that this is a counting loop.

Other variations

Many variations are possible and we will see them frequently throughout the remainder of the course. For example, we could *count down* instead of up:

```
for (int countdown = 10; countdown >= 1; countdown--)
{
    System.out.println(countdown);
}
System.out.println ("Blast off!");
```

Summary of `for` loops

The general structure of a `for` statement is the following:

```
for ( <initialization>; <condition>; <update>)
{
  <code to repeat>
}
```

- The initialization part is executed only once, when we first reach the `for` loop.
- The condition is executed before each iteration, including the first one.
- The update part is executed after each iteration, before testing the condition.

When should you use a `for` loop instead of a while loop:

- Definitely use for loops when counting!
- Initialization, condition, update all are expressed in terms of the same variable
- The variable is not modified elsewhere in the loop.
- It is correct to do the update command as the last statement in the body of the loop.

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 "Drag2Shirts" example to have 4 shirts instead of just 2.

See Example: Drag2Shirts

We could go through the program and everywhere we see `redShirt` and `blueShirt`, add 2 more variables and 2 more segments of code to deal with the new 2 shirts.

See Example: Drag4Shirts

That was not terribly painful, but a bit tedious and error prone. Now, what if we wanted to create 10, 20, or 100 shirts to be dragged around the canvas. We'd want a better way to name the shirts as a group.

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

Those of you who will go on to take data structures 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 here: 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 replace the 4 names of `TShirt` objects in the "Drag4Shirts" example with a single `ArrayList` that holds all 4.

See Example: Drag4ShirtsArrayList

This program has the same functionality, but the 4 variables for the `TShirts` has been replaced by a single collection, an `ArrayList` of `TShirt` objects.

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

- First, if we want to make use of a Java class not in the folder with our Java files, we need to tell Java this. Like we have seen for ObjectDraw and the `java.awt` classes, we need to add an `import` statement to the top of our program. In this case,

  ```
  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 an instance variable for our `ArrayList`:

  ```
      private ArrayList<TShirt> shirts;
  ```

  This looks a little different than any variable declaration we have seen before. Since an `ArrayList` 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 `TShirts`. So we place that type inside the `<` and `>`. This is called a *type parameter*.

- Like most Java classes, we need to construct an instance of the class in order use it. This is done in the first statement of the `begin` method:

  ```
          shirts = new ArrayList<TShirt>();
  ```

  This is much like other constructions we have seen, but we again need to include the type parameter so Java will give us an `ArrayList` that is set up to hold a collection of `TShirt` objects.

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

  ```
          shirts.add(shirt);
  ```

This will take the `TShirt` named `shirt` and add it to the first available slot in the `ArrayList` named `shirts`.

Note that in this case, we are not requesting any specific location within the `ArrayList` for the shirt. 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 shirt. We just know that it knows how to do it.

When we're done with `begin`, the `ArrayList` contains references to 4 `TShirt` objects.

- In the `onMousePress` and `onMouseExit` methods, we need to access the `TShirt` objects within the `ArrayList`. We do this with the `get` method:

  ```
  TShirt shirt = shirts.get(shirtNum);
  ```

  Here, `shirtNum` is a loop index variable that will range from 0 to one less than the number of items in the `ArrayList`. We know in this case that there are 4 items, but we can get that information from the `ArrayList` itself using the `size` method, as done in the `for` loops:

  ```
  for (int shirtNum = 0; shirtNum < shirts.size(); shirtNum++)
  ```

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

  It turns out that the first item we add is given index 0, the next is given index 1, and so on. If we later wanted to get at the first one, we could say:

  ```
  shirts.get(0);
  ```

  but in many cases (like this one), we will access the items within a collection inside a loop, passing in a loop index to the `get` method.

  This is our first example of a *search* operation on a collection – we are looking through each object in the collection to find one that contains the `Location`. 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. If we want to have 10 `TShirt`s on the canvas, we would definitely want to use a collection like an `ArrayList` to keep track of them.

See Example: Drag10Shirts

Here, we also place the creation of the `TShirt`s into a loop, but just line them up in a row for simplicity. If we wanted them to be organized into rows or to use a fixed set of colors, we would need to use a more complicated loop in the `begin` method. (And we will do just that later.)

If we wanted to create 20 or 50 or 100 `Tshirt`s, we could do so by changing the loop in the `begin` method and the remainder of the code does not need to change.

`ArrayList`s in Custom Objects

One of the challenges we have seen with constructing custom objects with any level of complexity is that we need to have names for all of the graphical objects we construct. When the object includes large numbers of items, ideally created within a loop, an `ArrayList` will come in handy to help keep track of them.

First, we look at a program that doesn't use `ArrayList`s:

See Example: DrawRoads

This program draws little segments of roads when we click the mouse. Nothing is new here – we could have written this a while ago.

But now suppose we want to be able to drag one of these around.

We need to have names for all of the components of the road segment so we can do things like move it and check for containment of a point.

See Example: DragRoads

The enhancements to the `WindowController` class are all very familiar.

It's in the `RoadSegment` class that we make use of an `ArrayList` to hold the center stripes of our road segment. Notice the same steps: declare a variable with an `ArrayList` type that can hold objects of the appropriate type, construct it with `new`, then `add` entries with the appropriate types of objects.

In the constructor, we do the construction of the `ArrayList`, then create the actual stripes.

In the `move` method, we loop through the stripes, moving each one.

This is nice, but perhaps we want to combine this functionality with that of the program where we could drag around any of 10 shirts. Let's use an `ArrayList` to keep track of **all** of the road segments we've created, so we can drag **any** segment, not just the most recently drawn one.

See Example: DragAllRoads

Here, in addition to having an `ArrayList` to keep track of the components of one of the road segments, we keep an `ArrayList` of `RoadSegment` objects in the `WindowController` class.

Removing from an `ArrayList`

We can augment the last example to remove each road segment from the canvas and from the `ArrayList`. A road segment will be removed if it is being dragged when the mouse leaves the window.

See Example: DragAllRoadsRemove

The new functionality is in the `onMouseExit` method of the `DragAllRoadsRemove` class. If the `dragging` flag is true when the mouse leaves the window, the currently-dragged segment `selectedSegment`) should be removed. We first remove it from the canvas, then remove it from the `ArrayList`. We also set `dragging` back to false, since the object we were just dragging no longer exists.

First, we will look at the removal from the list, which is done with the `ArrayList`'s `remove` method. We pass as a parameter the element we want to remove, and if it is an element of the list, it is removed. It is important to note that when we remove an element from an `ArrayList` with `remove`, any subsequent entries will be "moved up". That is, if a list contains 5 elements (in positions numbered 0 through 4) and we remove the element at position 2, the `ArrayList` implementation of `remove` will shift the element that was in position 3 into position 2, and the one that was in position 4 into position 3. This means we can still use our `for` loop over the numbers from 0 to `size()-1` to visit all of our entries. In other words, `remove` does not leave a "hole" at the index from which the element was removed.

The new `removeFromCanvas` method is mostly like the ones we have seen in previous examples: to remove the custom object, we remove each of its components. The difference here is that we need to loop through the `ArrayList`, `get` and then `remove` each element. We also should remove the individual `FilledRect`s from the `ArrayList`, which we do all at once with the `clear` method.

We can also remove elements from an `ArrayList` by index rather than value. We will see examples of this soon.

Other `ArrayList` methods

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

The full documentation for the `ArrayList` can be found at `http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html`

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

- `contains` – determine if a given object is in the list
- `indexOf` – search for first occurrence of a given object in the list and return its index
- `set` – replace the contents at an index with a new element

A few more examples to bring some of this together:

See Example: MovingFlags

See Example: PongBricks

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 maximum number of elements that there are slots for. 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.

Armed with this new construct, let's revisit the shirt dragging program to use arrays.

See Example: Drag10Shirts

In this code, we we have a single array named `shirts`. This array is declared as an instance variable, constructed at the start of the `begin` method, and given values (references to actual `TShirt`s) just after.

Then in the `onMousePress` method, we loop through all of the array entries (as we did previously with an `ArrayList`) to determine which, if any, has been pressed. Finally, in `onMouseExit`, we tell all of the shirts to move back to their starting positions.

Let's see how this differs from the `ArrayList` version.

- Our instance variable declaration looks a bit different.
- When we construct the array in the `begin` 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`.

In this example, we have used an array to keep track of a collection of objects on the canvas. We can also use an array to keep track of the components of a custom object as we did with `ArrayList`s.

An enhancement to this example that shows some of the benefits of arrays, we draw the t-shirts in two rows and use a fixed array of colors for the shirts instead of random colors.

See Example: Drag10ShirtsNicer

A few things to notice here:

- We have an array of `Color`s initialized to 10 pre-defined color names that we'll use for our 10 t-shirts.
- The construction of the t-shirts takes place in a nested loop to make it easier to organize them into 2 rows of 5 shirts each.

Our next enhancement to this example is to draw and drag around 20 shirts, now in 4 rows of 5.

See Example: Drag20Shirts

Most of the program works correctly just by changing the value of the constant `NUM_ROWS` (yay constants). But...the array of colors is not large enough.

We account for this by reusing the colors once we've run out. This is accomplished with some modulo arithmetic:

```
shirts[shirtNum].setColor(shirtColors[shirtNum % shirtColors.length]);
```

Another Example

See Example: DragStudents

What you've been waiting for: being the stars of a program.

This is another "drag objects around" example, but this time the objects being dragged are your pictures.

In this example, we place the objects randomly on the canvas, but take some care to make sure they do not overlap at all. Notice the helper method `overlapsAny` that helps ensure this.

Any image being dragged is also made larger while it's being dragged.

Other than that, it's similar to dragging 10 shirts.

Inserting and Removing with Arrays

We have already seen that 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 not use graphical objects, just 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 lifetime, 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
}
```