SIENAcollege
Computer Science

Computer Science 120
Introduction to Programming
Siena College
Spring 2011

Topic Notes: Arrays

Our next major topic involves naming collections of items. But first, we will look at a loop construct that we will 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 spe
    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.

Arrays

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.

In mathematics this is done by attaching subscripts to names. We can talk about numbers n_1, n_2,... We want to be able to do the same thing with computer languages. The name for this type of group of elements is an *array*.

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).

Armed with this new construct, let's augment the shirt dragging program to be able to drag around more shirts.

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 to determine which, if any, has been pressed. Finally, in `onMouseExit`, we tell all of the shirts to move back to their starting positions.

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.

In our final enhancement to this example, 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]);
```

Arrays in Custom Objects

Arrays can be used to store any of the data types we have considered, and can be used in anywhere we use regular variables.

First, we look at a program that doesn't use arrays:

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 array to hold the center stripes of our road segment. Notice the same steps: declare a variable with an array type, construct it with `new`, then fill in the entries with the appropriate types of objects.

In the constructor, we do the construction of the array (we first compute the number of stripes we'll draw, so we know how how big to make the array), then create the actual stripes.

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

Notice there that we need to know how many elements are in our array, but the variable we used back in the constructor (`numStripes`) has gone out of scope and is no longer available. Fortunately, arrays in Java come with this information as standard equipment. After any array has been constructed, its length can be determined by using the `.length` field.

A few words of caution here:

- We have taken care to make our array exactly large enough to hold the number of objects we placed in it. If we attempted to put an object into `centerStripes[5]` after constructing it to have 5 slots (which, remember, are numbered 0 through 4), we would get an

`ArrayIndexOutOfBoundsException`. Basically, your program would crash in the same way you've all seen with `NullPointerException`s.

- It is perfectly legal to have an array of a given size but to use only some of the slots to store objects. However, we need to be careful not to attempt to use the values in those slots – they'll be `null`. The `.length` value is the number of slots in the array when we constructed it, not the number of its slots that contain actual data.

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 array 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 array to keep track of the components of one of the road segments, we keep an array of `RoadSegment` objects in the `WindowController` class.

The array `segments` is declared as an instance variable and is constructed in the `begin` method, large enough to hold 4 `RoadSegment` object references. Since the number of objects we'll store is not predetermined, we have a decision to make.

Some factors to consider:

- Once we construct an array (*i.e.*, `new`), its size cannot be changed. If we wish to change the size of the array, we need to construct a new array, copy the contents of the old array to the new, then throw away the old array.
- If we make the array too large to start and we never use most of the slots, we have wasted that space.
- If we make the array too small, we will quickly need to construct a new, larger one and copy over the contents.

The solution used here is to start with a fairly small array (4), but then double it in size each time it fills up.

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--;
```