CS125 : Introduction to Computer Science

Lecture Notes #10
Processing Data Collections

Lecture 10 : Processing Data Collections

Arrays and Loops

As you have already started to see, a great deal of our array code will rely on loops. This is because the loop will allow us to progressively increase (or decrease) a subscript variable of an array `arr`. As the variable increases from `0` through `arr.length - 1`, we will be able to access each cell of the array in turn. That is helpful because much of the loop processing code we write will be of the following form:

```
given a collection of array cells that we care about,
do some sort of particular work on every cell in that
collection
```

This usually gets set up using a `for`-loop statement. Whatever collection of array cells we have, we can consider the lowest-indexed cell as having index "`lo`", and the highest-indexed cell as having index "`hi`":

```
   -----------------------------------
   |    |       . . .            |   |
   |____|________________________|___|
     lo         . . .              hi
```

and in that case, the control for our `for`-loop could be set up as follows:

```
for (int i = lo; i <= hi; i++)
```

For example, if we have an array `arr` of `size` cells, and we want to perform work on all the cells, then our loop will need to traverse the entire array:

```
    -----------------------------------
    |    |       . . .            |   |
arr |____|________________________|___|
      0         . . .            size-1

for (int i = 0; i <= size - 1; i++)
```

We could also access the `length` value for the array:

```
for (int i = 0; i < arr.length; i++)
```

But if, for example, you didn't want to process the cell with index `0`, and only wanted to process the cells indexed `1` through `size - 1` (that is, `1` through `arr.length - 1`), then you'd just have "`lo`" be `1`:

```
    -----------------------------------
    |    |    |  . . .            |   |
arr |____|____|___________________|___|
      0    1     . . .           size-1

         |____________________________|
         here is the part we care about
```

and so in that case, the `for`-loop would start at `1`, not `0`:

```
for (int i = 1; i <= size - 1; i++)
```

So, whenever we are dealing with processing "every cell in our collection", we need to decide which cells are actually in the collection we care about, and then just have a `for`-loop run from the lowest index of those cells, to the highest index of those cells.

After that, we can consider writing most array-based code as a four step process:

1. Figure out what we want to do for each cell. Presumably we are traversing across the array to perform some particular work at each cell; what is that work? For some simpler problems, this step is easily figured out from the problem statement itself; for other, harder problems, this step is the hardest part of writing array code.

2. If we wanted to perform that work on one ordinary variable (i.e. a variable that we had declared on its own, rather than a variable that was a cell of an array), how would we do that? Figure that code out.

3. If we wanted to perform that work on the array cell with index `0`, how would we do that? This step is generally a slight alteration of step 2's solution – but nevertheless, it can be easier to figure step 2 out before worrying about bringing array syntax into the mix.

4. Finally, we can add in a loop to vary the index of the array cell we are dealing with. That way, instead of performing the work only on the array cell with index `0`, we will perform the work on every array cell we care about.

Performing the same work on every cell, unconditionally

At times, you want to perform the exact same work on every cell, without any chance at all that some cells get processed differently from other cells. For example, suppose that, given an array `arr` of type `int`, we wish to print every cell (i.e. print the value in each of the cells), one per line. Let's consider the four steps above with respect to this problem.

1. What is it we want to do for each cell? Well, in this case, the problem statement pretty clearly spells it out – we want to print the value of each cell (and start a newline after each value is printed).

2. If we just had an ordinary variable `x` of type `int`, how would we print that variable?

   ```
   System.out.println(x);  // this is how we would do that
   ```

3. How would we accomplish this for cell `0` of the array?

   ```
   System.out.println(arr[0]);
   ```

4. Finally, we want to do the above for all indices in the array, so have a `for`-loop run from `0` to `size - 1` and vary the array index:

   ```
   for (int i = 0; i < arr.length; i++)
      System.out.println(arr[i]);
   ```

For another example, given an array `arr` of type `int`, add `10` to every cell.

1. Once again, figuring out what to do at every cell is pretty straightforward here – the problem states it pretty clearly. Add `10` to every cell.
2. If we had some integer variable `x`, and wanted to add `10` to it, we would have the following code:

   ```
   x = x + 10;
   ```

3. To do that to cell `0` of the array, you would use:

   ```
   arr[0] = arr[0] + 10;
   ```

4. And finally, since we want to do this at every cell, we have our `for`-loop run from `0` through `arr.length - 1`:

   ```
   for (int i = 0; i < arr.length; i++)
      arr[i] = arr[i] + 10;
   ```

Performing some work on each cell, conditionally

Sometimes, we will want to traverse an array, but only perform work on certain cells. For example: given an array `arr` of type `int`, print all values in the array that are greater than `60`:

1. To begin with, we look at the problem statement and figure out what to do at every cell. It says we want to print all values that are greater than `60`, but of course, we don't know if a value is greater than `60` until we check! So, what we need to do at every cell is (1) see if the value is greater than `60`, and (2) if so, print it out.
2. If we had some regular `int` variable `x`, then we could do the following:

   ```
   if (x > 60)
   ```

   to check if `x` were greater than `60`. And then, if the condition is true, we just have a print statement, similar to the one we had earlier.

   ```
   if (x > 60)
      System.out.println(x);
   ```

3. Once we have the above code snippet, then to perform the same sort of code on the first cell of an array, we just replace `x` with `arr[0]` in both places:

   ```
   if (arr[0] > 60)
      System.out.println(arr[0]);
   ```

4. And finally, since we want to check this for every cell, the `for`-loop again runs from `0` to `arr.length - 1`, and we vary the array index.

    ```
    for (int i = 0; i < arr.length; i++)
       if (arr[i] > 60)
          System.out.println(arr[i]);
    ```

For another example, given an array `arr` of type `int`, we want to print out all even numbers in the array.

1. As with the previous example, we need to at least look at every cell to determine whether it satisfies our condition or not. So, the work we need to do at every cell is to see if the value is an even number, and if so, to print it out.

2. Recall that modulus is our way of determining whether a number is even – i.e. divide by `2` and check the remainder:

    ```
    x % 2 == 0
    ```

    If the above condition is true, the number is even. Otherwise, the result of the modulus operation would be `1` and the expression above would be false and the number would be odd. So, if the condition is true, you print the number.

    ```
    if (x % 2 == 0)
       System.out.println(x);
    ```

3. Performing this calculation on the first cell of the array is then a matter of replacing `x` with that first cell:

    ```
    if (arr[0] % 2 == 0)
       System.out.println(arr[0]);
    ```

4. And finally, we want to perform the above calculation at every cell of the array:

    ```
    for (int i = 0; i < arr.length; i++)
       if (arr[i] % 2 == 0)
          System.out.println(arr[i]);
    ```

Processing a few cells outside of the loop

Sometimes we perform work on every cell, but not the same work. It could be that we perform a certain task on *most* cells (meaning we'd write a loop to handle that repetition of the task), but one or more cells must have *different* work performed on them and thus must be handled on their own, outside of the loop. In that case, step 1 becomes harder to complete, since we need to figure out what work is done on each cell and it is not always the same.

For example, given an array `arr` of type `int`, print the values on one line, separated by ellipses, and start a new line when the task is done. For example, if you were given this array:

```
--------------------------
|    |    |    |    |    |
| 8  | 9  | 3  | 1  | 7  |
|____|____|____|____|____|

  0    1    2    3    4
```

we want to print

```
8...9...3...1...7
```

and then start a new line.

1. To begin, it might be easiest to consider the example above:

   ```
   8...9...3...1...7
   ```

   There are two differences between the printing of the last number in that sequence, and the printing of the earlier numbers:

   - the last number has no ellipses after it
   - the last number has a new line after it

   The other numbers are all similar in how they get printed, and so you are repeating the same basic work for all those numbers:

   ```
   8...9...3...1...7
   |__||__||__||__|
     same work
     repeated 4 times
   ```

   So, it appears that what we want, is as follows:

   - For all cells except the last one, print the value of that cell, then print ellipses, but do NOT start a new line afterwards.
   - For the last cell of the array, print the value, do NOT print ellipses, and then start a new line.

2. If we have an integer variable `x`, and we want to print its value, followed by ellipses, and then NOT start a new line, we would do the following:

   ```
   System.out.print(x + "...");
   ```

   And, if we ant to print the value, WITHOUT ellipses, and then start a new line, we would do the following:

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

   Those are the two expressions we need to be using in this problem. The first will be used for most of our array cells, and the second will be used for the last array cell.

3. Writing similar code for an array cell is just a matter of replacing `x` with `arr[0]`:

   ```
   System.out.print(arr[0] + "...");
   System.out.println(arr[0]);
   ```

4. Finally, we need to encase the first statement above in a loop, and have the second statement above, occur after the loop is done. The loop should NOT process the last cell, so it should only run up through index `arr.length - 2`, or in other words, the loop should continue as long as our index is less than `arr.length - 1`. Once the loop is done, we use the second statement above to access the last cell (the cell with index `arr.length - 1`).

   ```
   for (int i = 0; i < arr.length - 1; i++)
      System.out.print(arr[i] + "...");

   // i is out of scope at this point

   System.out.println(arr[arr.length - 1]);
   ```

   The following is an alternative to the above code, which keeps `i` in scope so that it can be used on the last line.

   ```
   int i;
   for (i = 0; i < arr.length - 1; i++)
      System.out.print(arr[i] + "...");

   // i is still in scope at this point, and if the loop has
   // exited, then the condition of the loop was false, which
   // means i == arr.length - 1

   System.out.println(arr[i]);
   ```

As another example, given an array `arr` of type `int`, find the minimum value in the array.

1. This problem, as stated, is tricky, since we haven't even made clear what to do with the minimum value. We find it, and...? What? We need to do something with the minimum; we can't just find it and forget about it.

   It is reasonable to assume that if we are searching for the minimum, we intend to make use of that information, and therefore, knowing what the minimum is, might be helpful. So, let's store the minimum in a variable (we can call this variable `min`). That restates the original problem as follows: given an array `arr` of type `int`, find the minimum value in the array and store it in the variable `min`.

   Now, it would not be possible to know for sure that we have the minimum value of the array, unless we look at every cell at least once – if there are cells we didn't look at, those values could have been smaller than the value we think is the minimum, so we can't get away with not looking at those cells. This means, at any point in the code, if we have not finished looking at all the cells, we can't be sure we have the minimum.

   However – we *could*, at least hypothetically, know what the minimum is of all the values we have seen so far. For example, if we've seen five values so far, we should at least be able to know what the smallest is out of those five. We just need to keep track of what we think is the smallest value, and be willing to update that if we find a smaller one. For example, if `min` holds the smallest of the first five values, and we look at the sixth value and that is smaller than `min`, then `min` should be updated to hold the sixth value, since that is the smallest value we've seen so far. But if the sixth value is NOT smaller than `min`, then that means not only is `min` the smallest of the first five values, but it's also the smallest of the first six values, and we can leave `min` alone for this step.

   So for most cells, the work at that cell will involve checking to see if that cell is less than or greater than the smallest value we had previously seen up to that point – and if it is less than the smallest value we had seen up to that point, saving our cell's value as the new "smallest value so far".

   The one problem is that we can't do this to start the code, because if `min` is not yet initialized, we cannot compare it to anything. So, to begin with, we will just set `min` equal to `arr[0]`. That is, if all we have seen is `arr[0]`, well, of course that is the "minimum value so far", since it's the only value we've seen! And then once `min` is initialized to be equal to `arr[0]`, then we can run the above "check if next value is smaller" code for all the other cells – for each cell after `arr[0]`, if that cell's value is less than whatever `min` is at that moment, then re-assign `min` to hold that cell's value.

2. Imagine we have a variable `x` of type `int`, and are currently storing the minimum value in a variable `min`. We want to see if `x` is less than `min` and, if so, update `min` to hold `x`, the new minimum value. We begin with a condition:

   ```
   if (x < min)
   ```

   and in the event that the above condition is true, you write the value of `x` into `min`, since you've found a new minimum value:

   ```
   if (x < min)
      min = x;
   ```

3. If you want to see how the above code would run on an array cell, just replace `x` with an array cell access:

   ```
   if (arr[0] < min)
      min = arr[0];
   ```

4. Finally, as stated in step 1, we shouldn't have to do any comparisons when looking at the first cell of the array – at that point, the first cell is the only value we've seen, so certainly, it has to be the minimum out of all the values we have seen so far at that point:

   ```
   int min = arr[0];
   ```

   Once we've got a value stored inside `min`, then for all other array cells, we can compare that cell's value to the minimum so far. Note that the `for`-loop starts with `i` being assigned the index of the array's second cell, since we've already dealt with the first cell.

   ```
   int min = arr[0];
   for (int i = 1; i < arr.length; i++)
      if (arr[i] < min)
         min = arr[i];
   ```