6. Arrays

So far, we've seen how to harness recursive class definitions to create an expandable list class, including the `IntList`, `SLList`, and `DLList`. In the next two sections of this book, we'll discuss how to build a list class using arrays.

This section of this book assumes you've already worked with arrays and is not intended to be a comprehensive guide to their syntax.

Array Basics

To ultimately build a list that can hold information, we need some way to get memory boxes. Prevously, we saw how we could get memory boxes with variable declarations and class instantiations. For example:

- `int x;` gives us a 32 bit memory box that stores ints.
- `Walrus w1;` gives us a 64 bit memory box that stores Walrus references.
- `Walrus w2 = new Walrus(30, 5.6);` gets us 3 total memory boxes. One 64 bit box that stores Walrus references, one 32 bit box that stores the int size of the Walrus, and a 64 bit box that stores the double tuskSize of the Walrus.

Arrays are a special type of object that consists of a numbered sequence of memory boxes. This is unlike class instances, which have named memory boxes. To get the ith item of an array, we use bracket notation as we saw in HW0 and Project 0, e.g. `A[i]` to get the `i` th element of A.

Arrays consist of:

- A fixed integer length, N
- A sequence of N memory boxes (N = length) where all boxes are of the same type, and are numbered 0 through N - 1.

Unlike classes, arrays do not have methods.

Array Creation

There are three valid notations for array creation. Try running the code below and see what happens. Click here for an interactive visualization.

- `x = new int[3];`
- `y = new int[]{1, 2, 3, 4, 5};`
- `int[] z = {9, 10, 11, 12, 13};`

All three notations create an array. The first notation, used to create `x`, will create an array of the specified length and fill in each memory box with a default value. In this case, it will create an array of length 3, and fill each of the 3 boxes with the default `int` value `0`.

The second notation, used to create `y`, creates an array with the exact size needed to accommodate the specified starting values. In this case, it creates an array of length 5, with those five specific elements.

The third notation, used to declare **and** create `z`, has the same behavior as the second notation. The only difference is that it omits the usage of `new`, and can only be used when combined with a variable declaration.

None of these notations is better than any other.

Array Access and Modification

The following code showcases all of the key syntax we'll use to work with arrays. Try stepping through the code below and making sure you understand what happens when each line executes. To do so, click here for an interactive visualization. With the exception of the final line of code, we've seen all of this syntax before.

```
int[] z = null;
int[] x, y;

x = new int[]{1, 2, 3, 4, 5};
y = x;
x = new int[]{-1, 2, 5, 4, 99};
y = new int[3];
z = new int[0];
int xL = x.length;

String[] s = new String[6];
s[4] = "ketchup";
s[x[3] - x[1]] = "muffins";

int[] b = {9, 10, 11};
System.arraycopy(b, 0, x, 3, 2);
```

The final line demonstrates one way to copy information from one array to another. `System.arraycopy` takes five parameters:

- The array to use as a source
- Where to start in the source array
- The array to use as a destination
- Where to start in the destination array
- How many items to copy

For Python veterans, `System.arraycopy(b, 0,x, 3, 2)` is the equivalent of `x[3:5] = b[0:2]` in Python.

An alternate approach to copying arrays would be to use a loop. `arraycopy` is usually faster than a loop, and results in more compact code. The only downside is that `arraycopy` is (arguably) harder to read. Note that Java arrays only perform bounds checking at runtime. That is, the following code compiles just fine, but will crash at runtime.

```
int[] x = {9, 10, 11, 12, 13};
int[] y = new int[2];
int i = 0;
while (i < x.length) {
    y[i] = x[i];
    i += 1;
}
```

Try running this code locally in a java file or in the visualizer. What is the name of the error that you encounter when it crashes? Does the name of the error make sense?

2D Arrays in Java

What one might call a 2D array in Java is actually just an array of arrays. They follow the same rules for objects that we've already learned, but let's review them to make sure we understand how they work.

Syntax for arrays of arrays can be a bit confusing. Consider the code `int[][] bamboozle = new int[4][]`. This creates an array of integer arrays called `bamboozle`. Specifically, this creates exactly four memory boxes, each of which can point to an array of integers (of unspecified length).

Try running the code below line-by-lines, and see if the results match your intuition. For an interactive visualization, click here.

```
int[][] pascalsTriangle;
pascalsTriangle = new int[4][];
int[] rowZero = pascalsTriangle[0];

pascalsTriangle[0] = new int[]{1};
pascalsTriangle[1] = new int[]{1, 1};
pascalsTriangle[2] = new int[]{1, 2, 1};
pascalsTriangle[3] = new int[]{1, 3, 3, 1};
int[] rowTwo = pascalsTriangle[2];
rowTwo[1] = -5;

int[][] matrix;
matrix = new int[4][];
matrix = new int[4][4];

int[][] pascalAgain = new int[][]{{1}, {1, 1},
                                 {1, 2, 1}, {1, 3, 3, 1}};
```

Exercise 2.4.1: After running the code below, what will be the values of x[0][0] and w[0][0]? Check your work by clicking here.

```java
int[][] x = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[][] z = new int[3][];
z[0] = x[0];
z[1] = x[1];
z[2] = x[2];
z[0][0] = -z[0][0];

int[][] w = new int[3][3];
System.arraycopy(x[0], 0, w[0], 0, 3);
System.arraycopy(x[1], 0, w[1], 0, 3);
System.arraycopy(x[2], 0, w[2], 0, 3);
w[0][0] = -w[0][0];
```

Arrays vs. Classes

Both arrays and classes can be used to organize a bunch of memory boxes. In both cases, the number of memory boxes is fixed, i.e. the length of an array cannot be changed, just as class fields cannot be added or removed.

The key differences between memory boxes in arrays and classes:

- Array boxes are numbered and accessed using `[]` notation, and class boxes are named and accessed using dot notation.
- Array boxes must all be the same type. Class boxes can be different types.

One particularly notable impact of these difference is that `[]` notation allows us to specify which index we'd like at runtime. For example, consider the code below:

```java
int indexOfInterest = askUserForInteger();
int[] x = {100, 101, 102, 103};
int k = x[indexOfInterest];
System.out.println(k);
```

If we run this code, we might get something like:

```
$ javac arrayDemo
$ java arrayDemo
What index do you want? 2
102
```

By contrast, specifying fields in a class is not something we do at runtime. For example, consider the code below:

```
String fieldOfInterest = "mass";
Planet p = new Planet(6e24, "earth");
double mass = p[fieldOfInterest];
```

If we tried compiling this, we'd get a syntax error.

```
$ javac classDemo
FieldDemo.java:5: error: array required, but Planet found
        double mass = earth[fieldOfInterest];
                           ^
```

The same problem occurs if we try to use dot notation:

```
String fieldOfInterest = "mass";
Planet p = new Planet(6e24, "earth");
double mass = p.fieldOfInterest;
```

Compiling, we'd get:

```
$ javac classDemo
FieldDemo.java:5: error: cannot find symbol
        double mass = earth.fieldOfInterest;
                           ^
  symbol:   variable fieldOfInterest
  location: variable earth of type Planet
```

This isn't a limitation you'll face often, but it's worth pointing out, just for the sake of good scholarship. For what it's worth, there is a way to specify desired fields at runtime called *reflection*, but it is considered very bad coding style for typical programs. You can read more about reflection here. **You should never use reflection in any 61B program**, and we won't discuss it in our course.

In general, programming languages are partially designed to limit the choices of programmers to make code simpler to reason about. By restricting these sorts of features to the special Reflections API, we make typical Java programs easier to read and interpret.

Appendix: Java Arrays vs. Other Languages

Compared to arrays in other languages, Java arrays:

- Have no special syntax for "slicing" (such as in Python).
- Cannot be shrunk or expanded (such as in Ruby).
- Do not have member methods (such as in Javascript).
- Must contain values only of the same type (unlike Python).

Previous
5. DLLists

Next
7. Testing

Last updated 5 months ago