CMSC131

Data Structures: The Array

Arrays: The Concept

There are a wide variety of data structures that we can use or create to attempt to hold data in useful, organized, efficient ways.

The `MaritanPolynomial` class you created as part of Project 4 is a simple data structure in some ways. It holds three values; one coefficient for each of the three terms.

An **array** is a linear, contiguous, homogenous structure that can hold an arbitrary number of elements (as long as we know how many we want it to hold in advance).

Arrays: Some Properties

- Elements in an array of size ***n*** are numbered (some languages do 1..***n***, many like Java use 0..***n***-1).
- We can use those numbers to directly access an array position to read from or write into it.
- The data structure has a "first" and "last" position.
- In an array with more than 1 element, any position other than the first has a "previous" position and any position other than the last has a "next" position.
- We generally need to specify how many elements an array will have room for when we declare it.

index	0	1	2	3	4
values					

Arrays in Java

- In Java, an array is an object and we will have a variable serve as a reference to that object.
- We will indicate the type of elements that will be stored when we declare it.

```
datatype[] arrayName;
```

- We will allocate the actual array.

```
arrayName = new datatype[size];
```

- We will access a position within an array by using the name of the reference variable, square brackets, with the position inside the brackets.

```
arrayName[position]= value;
```

Some Motivation

We have actually been using arrays quite a bit, they've just been behind the scenes.

The **`String`** and **`StringBuffer`** classes both hold characters in an array.

To better understand the efficiency example we saw, we need to consider how arrays are allocated and used.

A Simple Problem

What if I asked you to write a program that would allow the user to specify how many numbers they wanted to enter, then read them in, and print them out backwards?

What might we need to do?

- Declare some variables like an integer to store the answer when we ask the user how many numbers they want to enter and a variable to act as a reference to an array.
- Get the size request from the user.
- Allocate an array big enough to hold them all.
- Allow the user to enter those numbers one at a time and store them in the array.
- Traverse through the array backwards and print them out.

What might we need to do?

```
int[] dataArray;
int numberOfElements;
```

- Get the size request from the user.
- Allocate an array big enough to hold them all.
- Allow the user to enter those numbers one at a time and store them in the array.
- Traverse through the array backwards and print them out.

What might we need to do?

```
int[] dataArray;
int numberOfElements;
numberOfElements = sc.nextInt();
```

- Allocate an array big enough to hold them all.
- Allow the user to enter those numbers one at a time and store them in the array.
- Traverse through the array backwards and print them out.

What might we need to do?

```
int[] dataArray;
int numberOfElements;
numberOfElements = sc.nextInt();
dataArray = new int[numberOfElements];
```

- Allow the user to enter those numbers one at a time and store them in the array.
- Traverse through the array backwards and print them out.

What might we need to do?

```
int[] dataArray;
int numberOfElements;
numberOfElements = sc.nextInt();
dataArray = new int[numberOfElements];

for (int i=0; i<numberOfElements; i++) {
     dataArray[i] = sc.nextInt();
} //I am using “i” to make this fit page
```

- Traverse through the array backwards and print them out.

What might we need to do?

```
int[] dataArray;
int numberOfElements;
numberOfElements = sc.nextInt();
dataArray = new int[numberOfElements];

for (int i=0; i<numberOfElements; i++) {
     dataArray[i] = sc.nextInt();
} //I am using “i” to make this fit page

for (int i=numberOfElements-1; i>=0; i--){
     System.out.print(dataArray[i] + " ");
} //I am using “i” to make this fit page
```

Length of an Array

- After allocating an array, we could have a variable that contains the size we requested.
- In some languages (including Java) an array will have an *instance variable* within the object that stores the size, but some other languages (like C++) do not, so we ***need*** to keep track of it.
- In Java, if you can access the size of the array via the instance variable **`length`**.

```
arrayName.length
```

- The **`.length`** field is a read-only value, so you cannot grow an array by changing this number!

Operations on an Array

- You can allocate and initialize an array at the same time if you know all of the information in advance.

```
int[] firstFivePrimes = {2,3,5,7,11};
```

- Remember that the variable we are using is a reference to an object. The following just makes a copy of the reference.

```
int[] notCopied = firstFivePrimes;
```

- We'll look at three types of copies a little later in our discussion.

"Growing" an Array

- We can't really grow an array once it has been allocated.
- What we ***can*** do is allocate a new array, copy the information from the old one into the new one, and move the array reference to point to this new array object.

Imagine we have an array named **myData** and we want to make it twice as large…

```
int[] tempName = new int[myData.length*2];
for (int i=0; i<myData.length; i++){
  tempName[i] = myData[i];
} //I am using “i” to make this fit page
myData = tempName;
```

Time consuming, isn't it…

Why **`StringBuffer`** was much faster!

- Every time we appended something onto our **`String`**, a whole new **`String`** object was created and the characters copied into it.
- With the **`StringBuffer`**, an initial size array is allocated, and only when that size is going to be exceeded by an operation is a new array allocated and all of the information copied in.
- In the current Java implementation, each time the size is going to be exceeded, the new array allocated is *twice as large*!

- What are the pros and cons of this?

Arrays of Objects

Arrays elements can be references to objects as well. If so, the array will hold the references to those objects, not the actual objects.

```
String[] cands = {"Obama", "Romney"};

StringBuffer[] cands = new StringBuffer[2];
    cands[0] = new StringBuffer("Obama");
    cands[1] = new StringBuffer("Romney");

String[] voters = new String[15000000];
```

Reference, Shallow, Deep Copies

Reference Copy

```
Student[] array1 = new Student[10];
//some code here which fills in the array with data
Student[] array2 = array1;
```

Shallow Copy

```
Student[] array1 = new Student[10];
//some code here which fills in the array with data
Student[] array2 = new Student[array1.length];
for (int i=0; i<array1.length; i++) {
   array2[i] = array1[i];
} //I am using “i” to make this fit page
```

Reference, Shallow, Deep Copies

Deep Copy

```
Student[] array1 = new Student[10];
//some code here which fills in the array with data
Student[] array2 = new Student[array1.length];
for (int i=0; i<array1.length; i++) {
   array2[i] = new Student(array1[i]);
} //I am using “i” to make this fit page
```

What is the danger in the above approach that neither reference nor shallow copies faced?

To what "value" are the elements of an array of references automatically initialized by default in Java?

1. zero
2. null
3. it depends on the data type
4. they aren’t initialized

`Arrays` class

There is a useful library class **`Arrays`** which contains a variety of static methods.

One subset of these that can be useful when exploring arrays is the group of **`toString()`** methods which take an array and generates an ASCII visualization of that array.

Another useful ability it provides is the ability to sort the contents of an array. This works for arrays of primitives and certain types of objects (we will see more of this later).

Arrays as Arguments

A reference to an array can be passed as an argument into a method.

You do NOT specify the size of the array since the array itself isn't really being passed into the method, just the reference to it.

Once the reference to the array is passed into a method, that method can access and alter the elements stored within the array.

Let's look at **`ArrayParameter.java`** and **`ArrayParameterDriver.java`**

initArray1

1. 999999999
2. 012345678
3. Not Sure

initArray2

1. 999999999
2. 012345678
3. Not Sure

"Privacy" Issues

- Even without arrays, object references create certain "privacy" issues.
- Consider how references to objects work and what it means when a method returns a reference to an object.
  - If it returns a reference to (for example) a private instance variable then the "outside world" now has direct access to it even though it is marked as private.
  - This doesn't "really" matter for immutable objects, but for mutable ones it would allow code from outside of the class to alter the contents of private data.
- With arrays, even though the size of the array is immutable, the ***contents*** aren't.

Multi-Dimensional Arrays

It is sometimes useful to have a multi-dimensional structure for data storage.

- Consider representing things such as a chess board or storage warehouse.

Declaring them as an "Array of Arrays"

- Allows us to have "ragged edged" arrays.

Declaring them as a "xD Array"

- Doesn't give any real advantage in Java as far as I can tell but might depending on the language.

2D Arrays – Syntax Examples

Rectangular

```
int[][] arr = new int[rows][cols];
```

Ragged

```
int[][] arr = new int[rows][];
arr[0] = new int[colsInRow0];
arr[1] = new int[colsInRow1];
arr[2] = new int[colsInRow2];
arr[3] = new int[colsInRow3];
  :
  :
```

Upper Left-Hand Right Triangle

We want to create a structure in which we store the distance from the "origin" to each cell in the structure, but only for the upper left-hand right triangle starting at that origin.

Rectangular array or Ragged array?

How do we make it work for a triangle with sides of length **`triangleSize`**?

Dealing with Ragged Arrays Generically

```
int[][] reallyRagged;
  reallyRagged = new int[5][];
  reallyRagged[0] = new int[9];
  reallyRagged[1] = new int[3];
  reallyRagged[2] = new int[7];
  reallyRagged[3] = new int[0];  //NOTE
  reallyRagged[4] = new int[12];
```

How could you go in and perform an operation on each element without having to hard-code the different lengths?

Self-awareness

Recall that arrays know their own length!

```
counter = 0;
for (row=0; row<reallyRagged.length; row++) {
  for (col=0; col<reallyRagged[row].length; col++) {
    reallyRagged[row][col] = counter++;
  }
}
```

What would happen if we had used this?

```
reallyRagged[3] = null;
```