Simple Arrays

Eric Roberts
CS 106A
February 15, 2017

Once upon a time . . .

A Quick Review of Array Lists

- In Java, an ***array list*** is an abstract type used to store a linearly ordered collection of similar data values.
- When you use an array list, you specify the type **`ArrayList`**, followed by the element type enclosed in angle brackets, as in **`ArrayList<String>`** or **`ArrayList<Integer>`**. In Java, such types are called ***parameterized types***.
- Each element is identified by its position number in the list, which is called its ***index***. In Java, index numbers always begin with 0 and therefore extend up to one less than the size of the array list.
- Operations on array lists are implemented as methods in the **`ArrayList`** class, as shown on the next slide.

Common `ArrayList` Methods

`list.size()` Returns the number of values in the list.
`list.isEmpty()` Returns **`true`** if the list is empty.
`list.set(i, value)` Sets the **`i`**th entry in the list to **`value`**.
`list.get(i)` Returns the **`i`**th entry in the list.
`list.add(value)` Adds a new value to the end of the list.
`list.add(index, value)` Inserts the value before the specified index position.
`list.remove(index)` Removes the value at the specified index position.
`list.clear()` Removes all values from the list.

Arrays in Java

- The Java **`ArrayList`** class is derived from an older, more primitive type called an ***array***, which is a collection of individual data values with two distinguishing characteristics:
  1. *An array is ordered.* You must be able to count off the values: here is the first, here is the second, and so on.
  2. *An array is homogeneous.* Every value in the array must have the same type.
- As with array lists, the individual values in an array are called ***elements***, the type of those elements (which must be the same because arrays are homogeneous) is called the ***element type***, and the number of elements is called the ***length*** of the array. Each element is identified by its position number in the array, which is called its ***index***.

Arrays Have Fewer Capabilities

`list.size()` Returns the number of values in the list.	**`array.length`**
✗ **`list.isEmpty()`** Returns **`true`** if the list is empty.	
`list.set(i, value)` Sets the **`i`**th entry in the list to **`value`**.	**`array[i] = value`**
`list.get(i)` Returns the **`i`**th entry in the list.	**`array[i]`**
✗ **`list.add(value)`** Adds a new value to the end of the list.	
✗ **`list.add(index, value)`** Inserts the value before the specified index position.	
✗ **`list.remove(index)`** Removes the value at the specified index position.	
✗ **`list.clear()`** Removes all values from the list.	

So Why Use Arrays?

- Arrays are built into the Java language and offer a more expressive selection syntax.
- You can create arrays of primitive types like **`int`** and **`double`** and therefore don’t need to use wrapper types like **`Integer`** and **`Double`**.
- It is much easier to create arrays of a fixed, predetermined size.
- Java makes it easy to initialize the elements of an array.
- Many methods in the Java libraries take arrays as parameters or return arrays as a result. You need to understand arrays in order to use those methods.

Declaring an Array Variable

- As with any other variable, array variables must be declared before you use them. In Java, the most common syntax for declaring an array variable looks like this:

  ```
  type[] name = new type[n];
  ```

  where *type* is the element type, *name* is the array name, and *n* is an integer expression indicating the number of elements.

- This declaration syntax combines two operations. The part of the line to the left of the equal sign declares the variable; the part to the right creates an array value with the specified number of elements.

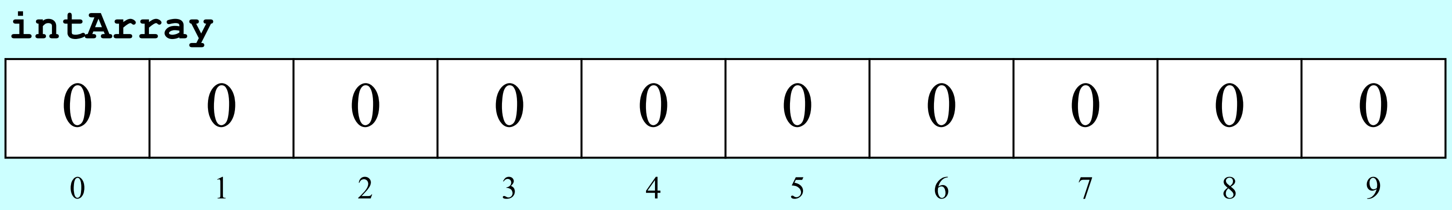
- Even though the two operations are distinct, it will help you avoid errors if you make a habit of initializing your arrays when you declare them.

An Example of Array Declaration

- The following declaration creates an array called **`intArray`** consisting of 10 values of type **`int`**:

```
int[] intArray = new int[10];
```

- This easiest way to visualize arrays is to think of them as a linear collection of boxes, each of which is marked with its index number. You might therefore diagram the **`intArray`** variable by drawing something like this:



- Java automatically initializes each element of a newly created array to its ***default value***, which is zero for numeric types, **`false`** for values of type **`boolean`**, and **`null`** for objects.

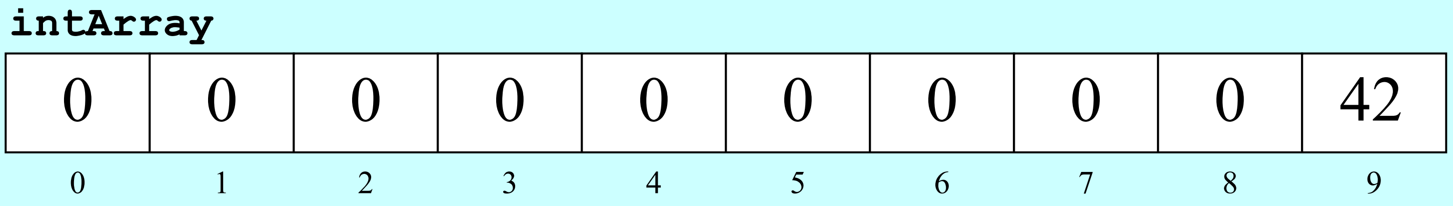
Array Selection

- Given an array such as the **`intArray`** variable at the bottom of this slide, you can get the value of any element by writing the index of that element in brackets after the array name. This operation is called ***selection***.
- You can, for example, select the initial element by writing

```
intArray[0]
```

- The result of a selection operation is essentially a variable. In particular, you can assign it a new value. The following statement changes the value of the last element to 42:

```
intArray[9] = 42;
```



Cycling through Array Elements

- One of the most useful things about array selection is that the index does not have to be a constant. In many cases, it is useful to have the index be the control variable of a **`for`** loop.
- The standard **`for`** loop pattern that cycles through each of the array elements in turn looks like this:

```
for (int i = 0; i < array.length; i++) {
    Operations involving the ith element of the array
}
```

Selecting the **`length`** field returns the number of elements.

- As an example, you can reset every element in **`intArray`** to zero using the following **`for`** loop:

```
for (int i = 0; i < intArray.length; i++) {
    intArray[i] = 0;
}
```

Exercise: Summing an Array

Write a method **`sumArray`** that takes an array of integers and returns the sum of those values.

```
/**
 * Calculates the sum of an integer array.
 * @param array An array of integers
 * @return The sum of the values in the array
 */

private int sumArray(int[] array) {
   int sum = 0;
   for (int i = 0; i < array.length; i++) {
      sum += array[i];
   }
   return sum;
}
```

Static Initialization

- Java makes it easy to initialize the elements of an array as part of a declaration. The syntax is

  ```
  type[] name = { elements };
  ```

  where *elements* is a list of the elements of the array separated by commas. The length of the array is automatically set to be the number of values in the list.

- For example, the following declaration initializes the variable **`powersOfTen`** to the values 10^0, 10^1, 10^2, 10^3, and 10^4:

  ```
  int[] powersOfTen = { 1, 10, 100, 1000, 10000 };
  ```

  This declaration creates an integer array of length 5 and initializes the elements as specified.

Passing Arrays as Parameters

- When you pass an array as a parameter to a method or return a method as a result, only the *reference* to the array is actually passed between the methods.

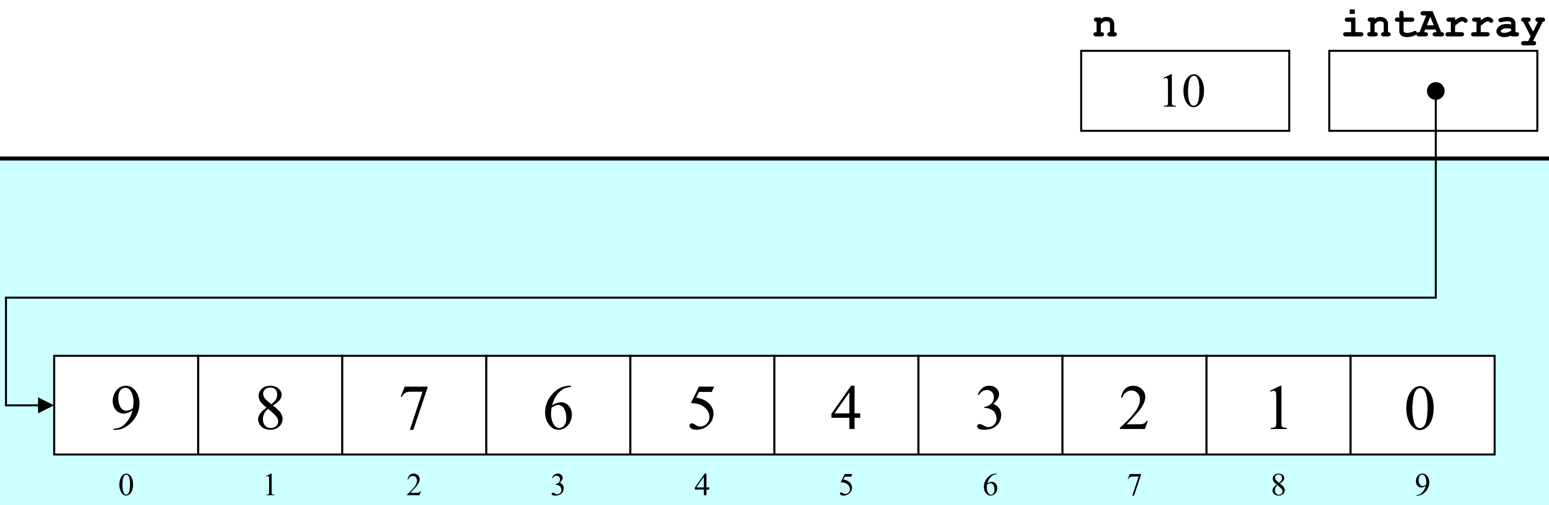
- The effect of Java's strategy for representing arrays as references is that the elements of an array are effectively shared between the caller and callee. If a method changes an element of an array passed as a parameter, that change will persist after the method returns.

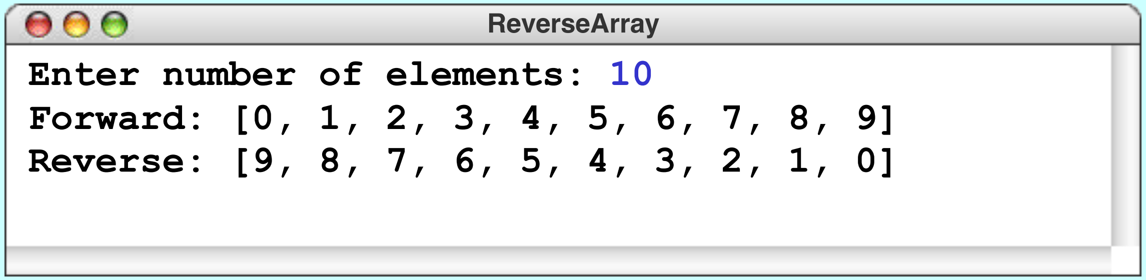
- The next slide contains a simulated version of a program that performs the following actions:

  1. Generates an array containing the integers 0 to $N-1$.
  2. Prints out the elements in the array.
  3. Reverses the elements in the array.
  4. Prints out the reversed array on the console.

The **ReverseArray** Program

```
public void run() {
   int n = readInt("Enter number of elements: ");
   int[] intArray = createIndexArray(n);
   println("Forward: " + arrayToString(intArray));
   reverseArray(intArray);
   println("Reverse: " + arrayToString(intArray));
}
```





```
Enter number of elements: 10
Forward: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Reverse: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

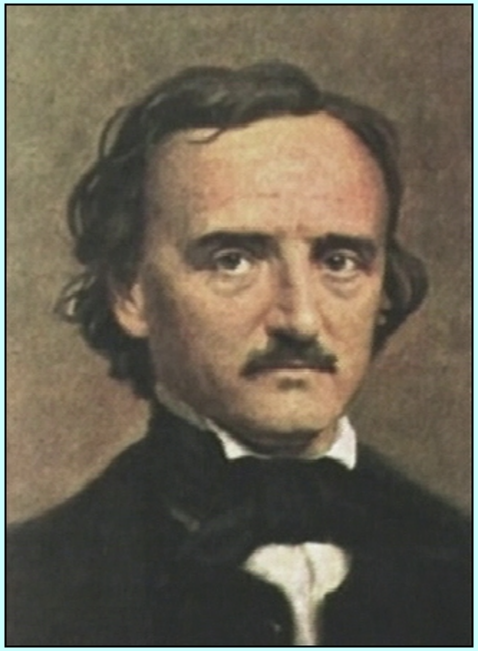
skip simulation

Using Arrays for Tabulation

- Arrays turn out to be useful when you have a set of data values and need to count how many values fall into each of a set of ranges. This process is called ***tabulation***.
- Tabulation uses arrays in a slightly different way from those applications that use them to store a list of data. When you implement a tabulation program, you use each data value to compute an index into an integer array that keeps track of how many values fall into that category.
- The example of tabulation used in the text is a program that counts how many times each of the 26 letters appears in a sequence of text lines. Such a program would be very useful in solving codes and ciphers, as described on the next slide.

Cryptograms

- A ***cryptogram*** is a puzzle in which a message is encoded by replacing each letter in the original text with some other letter. The substitution pattern remains the same throughout the message. Your job in solving a cryptogram is to figure out this correspondence.
- One of the most famous cryptograms was written by Edgar Allan Poe in his short story "The Gold Bug."
- In this story, Poe describes the technique of assuming that the most common letters in the coded message correspond to the most common letters in English, which are E, T, A, O, I, N, S, H, R, D, L, and U.


Edgar Allan Poe (1809-1849)

Poe’s Cryptogram Puzzle

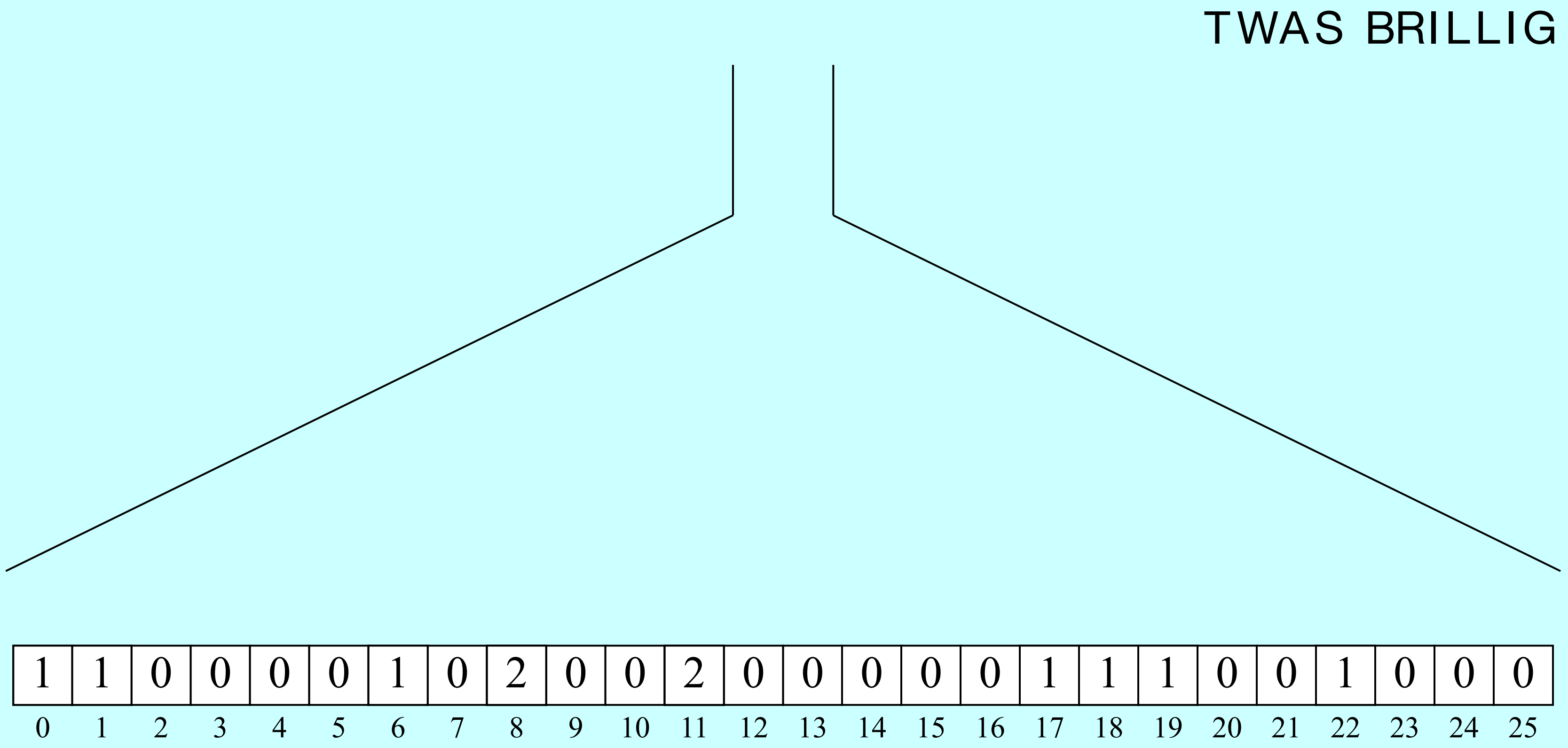
```
53‡‡†305))6*;4826)4‡•)4‡);806*;48†8¶
60))85;1‡(;:‡*8†83(88)5*†;46(;88*96*
?;8)*‡(;485);5*†2:*‡(;4956*2(5*–4)8¶
8*;4069285);)6†8)4‡‡;1(‡9;48081;8:8‡
1;48†85;4)485†528806*81(‡9;48;(88;4(
‡?34;48)4‡;161;:188;‡?;
```

```
AGOODGLASSINTHEBISHOPSHOSTELINTHEDEV
ILSSEATFORTYONEDEGREESANDTHIRTEENMIN
UTESNORTHEASTANDBYNORTHMAINBRANCHSEV
ENTHLIMBEASTSIDESHOOTFROMTHELEFTEYEO
FTHEDEATHSHEADABEELINEFROMTHETREETHR
OUGHTHESHOTFIFTYFEETOUT
```

8	33
;	26
4	19
‡	16
)	16
*	13
5	12
6	11
(	10
†	8
1	8
0	6
9	5
2	5
:	4
3	4
?	3
¶	2
–	1
•	1

Implementation Strategy

The basic idea behind the program to count letter frequencies is to use an array with 26 elements to keep track of how many times each letter appears. As the program reads the text, it increments the array element that corresponds to each letter.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	1	0	0	0	0	1	0	2	0	0	2	0	0	0	0	0	1	1	1	0	0	1	0	0	0

CountLetterFrequencies

```
import acm.program.*;

/**
 * This program creates a table of the letter frequencies in a
 * paragraph of input text terminated by a blank line.
 */
public class CountLetterFrequencies extends ConsoleProgram {

   public void run() {
      println("This program counts letter frequencies.");
      println("Enter a blank line to indicate the end of the text.");
      initFrequencyTable();
      while (true) {
         String line = readLine();
         if (line.length() == 0) break;
         countLetterFrequencies(line);
      }
      printFrequencyTable();
   }

/* Initializes the frequency table to contain zeros */
   private void initFrequencyTable() {
      frequencyTable = new int[26];
      for (int i = 0; i < 26; i++) {
         frequencyTable[i] = 0;
      }
   }
```

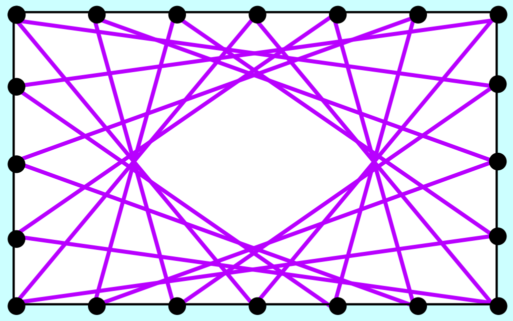
skip code

CountLetterFrequencies

```
/* Counts the letter frequencies in a line of text */
   private void countLetterFrequencies(String line) {
      for (int i = 0; i < line.length(); i++) {
         char ch = line.charAt(i);
         if (Character.isLetter(ch)) {
            int index = Character.toUpperCase(ch) - 'A';
            frequencyTable[index]++;
         }
      }
   }

/* Displays the frequency table */
   private void printFrequencyTable() {
      for (char ch = 'A'; ch <= 'Z'; ch++) {
         int index = ch - 'A';
         println(ch + ": " + frequencyTable[index]);
      }
   }

/* Private instance variables */
   private int[] frequencyTable;

}
```

Arrays and Graphics

- Arrays turn up frequently in graphical programming. Any time that you have repeated collections of similar objects, an array provides a convenient structure for storing them.
- As an aesthetically pleasing illustration of both the use of arrays and the possibility of creating interesting pictures using nothing but straight lines, the text presents the **`YarnPattern`** program, which simulates the following process:
  - Place a set of pegs at regular intervals around a rectangular border.
  - Tie a piece of colored yarn around the peg in the upper left corner.
  - Loop that yarn around the peg a certain distance **`DELTA`** ahead.
  - Continue moving forward **`DELTA`** pegs until you close the loop.



The YarnPattern Program

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;

/**
 * This program creates a pattern that simulates the process of
 * winding a piece of colored yarn around an array of pegs along
 * the edges of the canvas.
 */
public class YarnPattern extends GraphicsProgram {

   public void run() {
      initPegArray();
      int thisPeg = 0;
      int nextPeg = -1;
      while (thisPeg != 0 || nextPeg == -1) {
         nextPeg = (thisPeg + DELTA) % N_PEGS;
         GPoint p0 = pegs[thisPeg];
         GPoint p1 = pegs[nextPeg];
         GLine line = new GLine(p0.getX(), p0.getY(), p1.getX(), p1.getY());
         line.setColor(Color.MAGENTA);
         add(line);
         thisPeg = nextPeg;
      }
   }
```

skip code

The YarnPattern Program

```
/* Initializes the array of pegs */
   private void initPegArray() {
      int pegIndex = 0;
      for (int i = 0; i < N_ACROSS; i++) {
         pegs[pegIndex++] = new GPoint(i * PEG_SEP, 0);
      }
      for (int i = 0; i < N_DOWN; i++) {
         pegs[pegIndex++] = new GPoint(N_ACROSS * PEG_SEP, i * PEG_SEP);
      }
      for (int i = N_ACROSS; i > 0; i--) {
         pegs[pegIndex++] = new GPoint(i * PEG_SEP, N_DOWN * PEG_SEP);
      }
      for (int i = N_DOWN; i > 0; i--) {
         pegs[pegIndex++] = new GPoint(0, i * PEG_SEP);
      }
   }

/* Private constants */
   private static final int DELTA = 67;     /* How many pegs to advance      */
   private static final int PEG_SEP = 10;   /* Pixels separating each peg    */
   private static final int N_ACROSS = 50;  /* Pegs across (minus a corner) */
   private static final int N_DOWN = 30;    /* Pegs down (minus a corner)   */
   private static final int N_PEGS = 2 * N_ACROSS + 2 * N_DOWN;

/* Private instance variables */
   private GPoint[] pegs = new GPoint[N_PEGS];

}
```

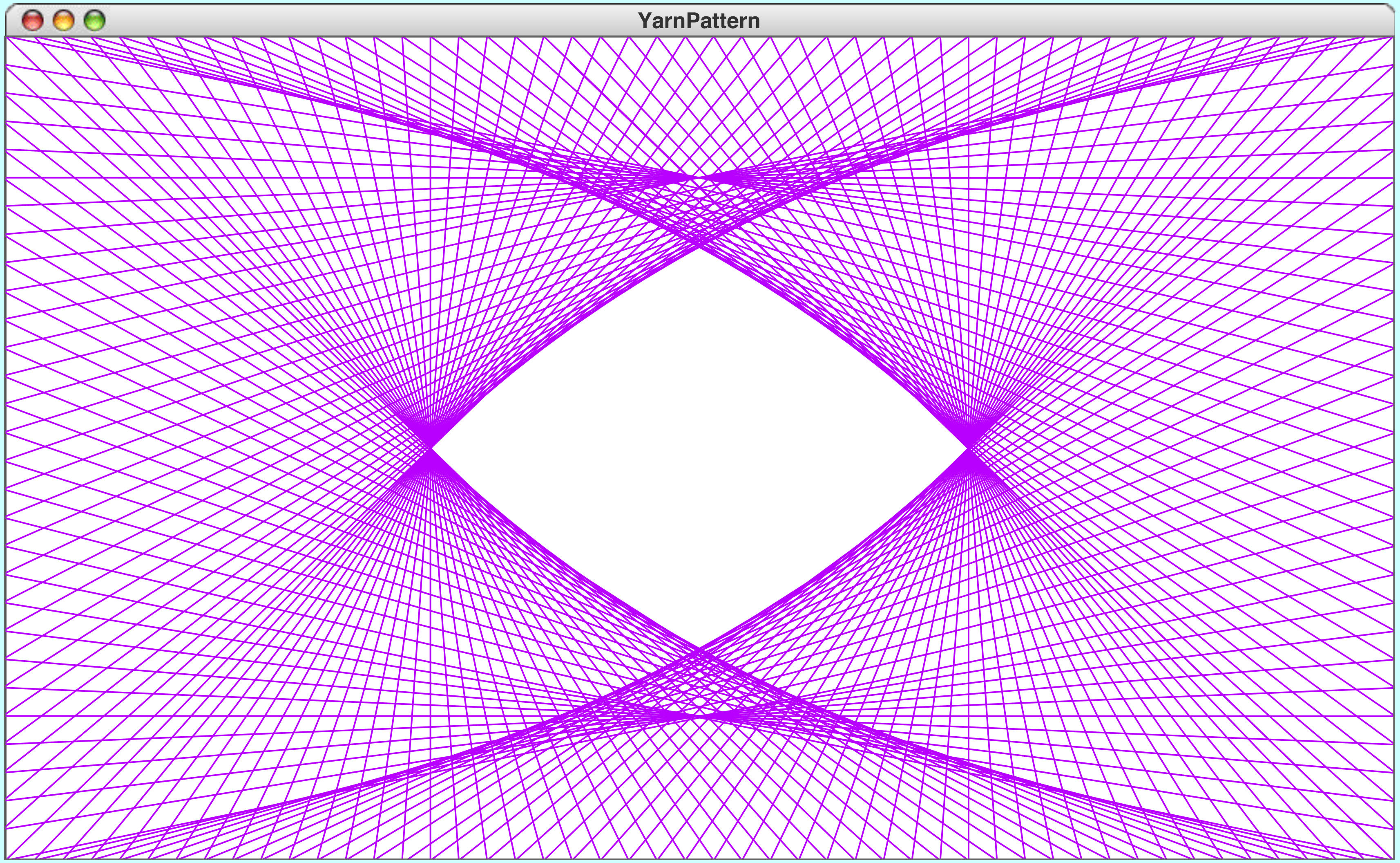
A Digression on the **++** Operator

- The **`YarnPattern`** program illustrates a new form of the **`++`** operator in the various statements with the following form:

```
pegs[pegIndex++] = new GPoint(x, y);
```

- The **`pegIndex++`** expression adds one to **`pegIndex`** just as if has all along. The question is what value is used as the index, which depends on where the **`++`** operator appears:
  - If the **`++`** operator comes *after* a variable, the variable is incremented after the value of the expression is determined. Thus, in this example, the expression **`pegs[pegIndex++]`** therefore selects the element of the array at the *current* value of **`pegIndex`** and then adds one to **`pegIndex`** afterwards, which moves it on to the next index position.
  - If the **`++`** operator comes *before* a variable, the variable is incremented first and the new value is used in the surrounding context.
- The **`--`** operator behaves similarly but subtracts one from the variable instead.

A Larger Sample Run



The End