Lecture 9: Arrays and Index-based For-Loops

Contents

1 Motivating Example: Weather Data 1

2 Accessing Elements Quickly: Arrays 1
   2.1 Declaring an Array: Types and Creation 3
   2.2 Inserting Elements 4
   2.3 Accessing Elements 4
      2.3.1 Dealing with Invalid Dates 4
      2.3.2 Dealing with Missing Dates 5

3 Traversing Arrays 6

4 Summary: Different Kinds of Loops 7

5 Summary: Arrays vs Lists 7

Objectives

By the end of this lecture, you will know:

- What arrays are and how they differ from lists
- Two different kinds of for-loops

By the end of this lecture, you will be able to:

- Create and manipulate arrays
- Use position-based for-loops to traverse arrays

1 Motivating Example: Weather Data

Imagine that you’re creating a class to store weather data for each day in a month. In your initial design, the class maintains a list of reports, one for each day of the month. The initial code is in Figure 1: it includes an inner class for storing data about one element, and a method recordDate that adds information about a date into the list.

Note: this code has switched over to using Java’s built-in classes for Linked Lists (which extends on what we’ve been implementing in lecture). You use the built-in version by adding the following line to the top of your file:
import java.util.LinkedList;

public class MonthWeatherL {
    int numDays; // number of days in month
    LinkedList<DailyWeather> dailyW = new LinkedList<DailyWeather>();

    public MonthWeatherL(int numDays) {
        this.numDays = numDays;
    }

    // an inner class for the weather for one day
    class DailyWeather {
        int date;
        double temp;

        DailyWeather(int d, double t) {
            this.date = d;
            this.temp = t;
        }

        // return the temperature for this date
        double getTemp() {
            return this.temp;
        }
    }

    // store the weather data for the given date
    public void recordDate(int forDate, double temp) {
        dailyW.addLast(new DailyWeather(forDate, temp));
    }

    // retrieve the temperature for the given date
    public double tempForDate(int forDate) {
        for (DailyWeather dw : dailyW) {
            if (dw.date == forDate) {
                return dw.getTemp();
            }
        }
        throw new RuntimeException("No data for date: " + forDate);
    }

    public static void main(String[] args) {
        // set up a month with data for three dates
        MonthWeatherL janW = new MonthWeatherL(31);
        janW.recordDate(1, 20);
        janW.recordDate(2, 15);
        janW.recordDate(3, 50);
        System.out.println("Temp on Jan 2: " + janW.tempForDate(2));
    }
}

Figure 1: A list-based implementation of managing monthly weather data
import java.util.LinkedList;

The meteorologists you support with this code end up using tempForDate a lot (as part of weather predictions). You start to worry about the time required to keep traversing the list with the loop in that method. Fortunately, there is a common data structure that addresses that problem ...

2 Accessing Elements Quickly: Arrays

Think about how a LinkedList is laid out in memory: we have a bunch of Node objects linked together through their next fields. To search for a specific date, we have to traverse all those arrows, which gets expensive.

The weather problem has a special feature, however – we are accessing DailyWeather objects based on a number (the date), and the numbers are consecutive. This suggests leveraging the numbers to place the objects in memory in such a way that we can find them faster. For example, what if they were directly adjacent in memory, as shown in the lower bit of memory (labeled “Array”).

With the array layout, once you know the memory location of the array object, you (or Java) can easily compute the memory location of any item given its position: roughly, the location is

\[
arrayLocation + (posWanted * perElementSpace)
\]

where perElementSpace is the amount of space Java needs to store the reference to each DailyWeather object (those objects can be anywhere in memory).

The data structure which yields the second picture is called an array. Arrays exist in nearly all programming languages. To illustrate arrays, we’ll migrate our MonthWeather class over to use arrays instead.

2.1 Declaring an Array: Types and Creation

Array Types To say “dailyW should be an array of DailyWeather objects”, we write the following:
The pair of square brackets is the “array” part.

No import statement is needed: arrays are available in Java as are basic data types like `int` and `String`.

**Creating Arrays**  As we saw in the picture, the whole idea of an array is to store many objects consecutively in memory. For Java to set aside the right amount of space, it needs to know how many elements we want to store in the array. Here’s the revised constructor:

```java
public MonthWeatherA(int numDays) {
    this.numDays = numDays;
    this.dailyW = new DailyWeather[numDays];
}
```

Note that when we create the actual `dailyW` array, we put the size of the array in the square brackets. For this example, we had to wait until we had the number of days to create the array, so the array creation has to occur inside the constructor.

**2.2 Inserting Elements**

How to we add something to an array? We use a combination of square-brackets notation and the `=` sign (in mutation/assignment mode), as follows:

```java
// almost right, but not quite ...
public void recordDate(int forDate, double temp) {
    dailyW[forDate] = new DailyWeather(forDate, temp);
}
```

This is fine notationally, but not quite what we wanted to do. Arrays number positions starting with 0, not with 1 (this is true across languages). So the first day of the month is in position 0, the second day in position 1, and so on.

```java
// correct version that adjusts for array positions starting at 0
public void recordDate(int forDate, double temp) {
    dailyW[forDate-1] = new DailyWeather(forDate, temp);
}
```

You could also deal with this position-starts-at-0 issue for the weather application by creating an array of size `numDays+1` and never using position 0. In a real implementation, we would probably do something like that just to avoid the inevitable errors from forgetting to adjust the date. In these notes, we will stick to the -1 version to help drive home the point that arrays are indexed starting from 0.
2.3 Accessing Elements

What if we now want to know the temperature on the 2nd day of a month? We again use the square-bracket notation, this time providing the position that we want to access within the array. Our first instinct is to try the following:

```java
// first attempt to retrieve the temp for a given date from the array
public double tempForDate(int forDate) {
    return dailyW[forDate].getTemp();
}
```

But remember that arrays start at position 0, so we have to make the same-1 adjustment as we did when storing elements:

```java
// correct version that adjusts date around array positions starting at 0
public double tempForDate(int forDate) {
    return dailyW[forDate-1].getTemp();
}
```

2.3.1 Dealing with Invalid Dates

In our lists implementation, if someone requested data for a date that didn’t exist in the dailyW list, we threw a RuntimeException after we ran off the end of the list. With arrays, Java itself will throw an error if we try to access a position that does not fall within the array (this is called an array out-of-bounds error).

Ideally, we should check that the requested date is valid before accessing the array. Here’s the code, again using a RuntimeException:

```java
// adding checking for valid dates
public double tempForDate(int forDate) {
    if (forDate > numDays || forDate < 1) {
        throw new RuntimeException("Invalid date: " + forDate);
    }
    return dailyW[forDate-1].getTemp(); // -1 because arrays start at 0
}
```

2.3.2 Dealing with Missing Dates

In our list implementation, if no data had been recorded for a date, Java would run off the end of the list (in the for-loop in tempForDate) and we’d throw the RuntimeException. With the array version, we create a position for every date in the month (in the constructor) by default. This means someone can try to access the temperature for a date that exists in the array, but for which there is no data. What should happen in this case?

If we are programming carefully, we should put some object in every position of the array when we create it. That object should be capable of responding appropriately to every method we might want to call on valid DailyWeather objects.

There are two approaches to this problem:
Use null to signal no weather data

Create a NoWeatherData class to signal no weather data.

Using null If you want to set each position in the array to null, use the following constructor
(the Arrays part comes from an import of an array class that you’ll see in the source file).

```java
public MonthWeatherA(int numDays) {
    this.numDays = numDays;
    this.dailyW = new IWeatherData[numDays];
    Arrays.fill(dailyW, null);
}
```

Using A Special Class In contrast, if you want to define a class for no weather data, the constructor would look as follows (assuming you had already defined the class elsewhere):

```java
public MonthWeatherA(int numDays) {
    this.numDays = numDays;
    this.dailyW = new IWeatherData[numDays];
    Arrays.fill(dailyW, new NoWeatherData());
}
```

Either way, whenever you have code that accesses elements of the array, it must include some code
that handles the case when no useful data is in that position. We’ll see an example of this shortly.

3 Traversing Arrays

We have seen how to access individual elements of an array, but sometimes we still want to traverse
over all elements of an array. For example, we might want the average temperature for a month.

Arrays do not implement the iterator interface, so you cannot use the iterator-based for-loops that
we used with the list-based MonthWeather class. For arrays, we use a different for-loop notation
that explicitly manages the positions to access. Here’s an example:

```java
/** produce the average temperature for this month
 * @return average temperature across all days
 */
public double avgTemp() {
    int pos; // local variable for the date
    double totalTemp = 0; // running sum of temps
    for (pos = 0; pos < numDays; pos=pos+1) {
        totalTemp = totalTemp + dailyW[pos].getTemp();
    }
    return (totalTemp / this.numDays);
}
```

This style of for loop needs:
Let's start by seeing what the code looks like if we used null to indicate a lack of data. The version for when you used a class will be in the source code posted with today’s class.

```java
/** produce the average temperature for this month  
 * @return average temperature across all days 
 */
public double avgTemp() {
    int pos; // local variable for the date positions
    double totalTemp = 0; // running sum of temps
    for (pos = 0; pos < numDays; pos=pos+1) {
        if (dailyW[pos] != null) {
            totalTemp = totalTemp + dailyW[pos].getTemp();
        }
    }
    return (totalTemp / this.numDays);
}
```

Ask yourself: Does this look right now? Any other issues we need to consider?

## 4 Summary: Different Kinds of Loops

We have seen three different kinds of loops across the last two lectures:

- a variable of type int (here, `pos`) to hold the current position to be processed
- an initial position to consider (here, `pos=0`)
- a condition indicating when to keep looping (here, `pos < numDays`)
- instructions on how to update the position for the next iteration of the loop (here, `pos=pos+1`)

Inside the loop, we perform whatever computation we had in mind (here, updating the running `totalTemp` across the array).

These index-based loops give you more flexibility than the iterator loops. For example, you could sample the temperature on the same day of every week by using `pos=pos+7` as the position-update statement; you could look at even numbered days by using `pos=pos+2`, and so on. You could even start at the end of the array and count down (that might matter for printing values in a particular way).

These index-based loops are also more error-prone because you have to manage the position updates yourself. People also often forget that the position variable (here, `pos`) is the position and not the data itself (they might accidentally write `totalTemp = totalTemp + pos`, for example). Iterator-based loops are considered better practice unless your data or computation actually needs the position-based ones.

### Handling positions with no data

The version of `avgTemp` that we just saw assumes that there is a `DailyWeather` object in every position of the array. This might not be true (as in the case of our main example). In practice, you need to put some code in the loop body to handle such cases if they arise in your situation. Here’s what that code would look like if you used `null` to indicate a lack of data. The version for when you used a class will be in the source code posted with today’s class.
• **while** loops have you continue a computation until some condition has occurred. Here, we saw **while** loops for traversing a list, but we could have also been repeating other tasks (such as asking a user to type in numbers, stopping when they enter the word “done”).

• **Iterator-style** for loops visit each element in a list (or other collection of data) exactly once, letting you perform some aggregating computation over each element.

• **Position-based** for loops let you repeat a computation on each item in some formula-defined sequence, such as all the numbers from 0 to the size of an array, or all even numbers.

Usually, which type of loop you use is determined by a combination of the data structure you are using (which style of for-loop) and the computation to perform (for-vs-while).

## 5 Summary: Arrays vs Lists

This lecture introduce arrays, contrasting them to lists for our problem of managing weather data. Let’s review and contrast the features of both:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Array</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory layout</td>
<td>consecutive elements</td>
<td>elements scattered in memory</td>
</tr>
<tr>
<td>Time to access an element</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>Time to add an element</td>
<td>constant if position exists (else must extend array)</td>
<td>constant if inserting at an end</td>
</tr>
<tr>
<td>Space used</td>
<td>proportional to number of elements declared up front</td>
<td>proportional to number of elements in list; also needs space for next references</td>
</tr>
<tr>
<td>Time to remove element</td>
<td>depends on position, since must shift remaining elements over to keep items consecutive</td>
<td>linear</td>
</tr>
<tr>
<td>Iteration</td>
<td>position-based for loop</td>
<td>iterator-based for or while</td>
</tr>
</tbody>
</table>

Sometimes, you want both the convenience of iterators and constant-time access to elements. In that case, you can use a Java class called `ArrayList`. They take a bit more space than arrays, and they can only be used with objects (as opposed to plain int or bools). They are often a reasonable choice if you use each of insert, delete, and access with high frequency.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)