# Fixed-size Collections

### Java version

# Objectives

To investigate how to store and access information in objects by introducing collections, arrays, and loops.

## Main concepts discussed in this chapter

- collections
- iterators
- loops
- arrays

# Resources

Classes needed for this lab – *chapter04.jar*.

# To do

## Installing the lab session

Just like in the previous lab sessions.

## Building on themes from previous sections

As well as introducing new material on collections and iteration, we will also be revisiting two of the key themes that were already introduced: *abstraction* and *object interaction*. We have seen that abstraction allows us to simplify a problem by identifying discrete components that can be viewed as a whole, rather than being concerned with their detail. We will see this principle in action when we start making use of the library classes that are available in Java. While these classes are not, strictly speaking, a part of the language, some of them are so closely associated with writing most Java programs that they are often thought of in that way. Most people writing Java programs will constantly check the libraries to see if someone has already written a class that they can make use of. That way, they save a huge amount of effort that can be better used in working on other parts of the program. The same principle applies with most other programming languages, which also tend to have libraries of useful classes. So it pays to become familiar with the contents of the library and how to use the most common classes. The power of abstraction is that we don't usually need to know much (if anything, indeed!) about what the class looks like inside to be able to use it effectively.

If we use a library class, it follows that we will be writing code that creates instances of those classes, and then our objects will be interacting with the library objects. Object interaction will also figure highly in this chapter,

therefore.

In this chapter, we also extend our understanding of abstraction to see that it does not just mean hiding detail but also means seeing the common features and patterns that recur again and again in programs. Recognizing these patterns means that we can often reuse in a new situation part or all of a method or class we have previously written. This particular applies when looking at collections and iteration.

# Fixed-size collections

- Flexible collections have the advantage of storing a number of objects which is not known when the code is written. In some cases this number can be known in advance, in which case we can use fixed-size collections (called *arrays*).
- Arrays have the advantage that access to their elements is sometimes faster.
- Arrays *can* store primitive types as well as objects.

## A log-file analyer

Web servers generally keep track of events in a log file from which the following sorts of information can be extracted

- which are the most popular pages they provide
- which sites brought users to this one
- whether other sites appear to have broken links to this site's pages
- how much data is being delivered to clients
- the busiest periods over the course of a day, a week, or a month

The *webloganalyzer* project contains the **`LogAnalyzer`** class

```
package webloganalyzer;

/**
 * Read web server data and analyse hourly access patterns.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version    2016.02.29
 */
class LogAnalyzer {
    // Where to calculate the hourly access counts.
    private final int[] hourCounts;
    // Use a LogfileReader to access the data.
    private final LogfileReader reader;

    /**
     * Create an object to analyze hourly web accesses.
     */
    private LogAnalyzer() {
        // Create the array object to hold the hourly
        // access counts.
        hourCounts = new int[24];
        // Create the reader to obtain the data.
        reader = new LogfileReader();
    }
```

```
    /**
     * Analyze the hourly access data from the log file.
     */
    private void analyzeHourlyData() {
        while (reader.hasNext()) {
            LogEntry entry = reader.next();
            int hour = entry.getHour();
            hourCounts[hour]++;
        }
    }

    /**
     * Print the hourly counts.
     * These should have been set with a prior
     * call to analyzeHourlyData.
     */
    private void printHourlyCounts() {
        System.out.println("Hr: Count");
        for (int hour = 0; hour < hourCounts.length; hour++) {
            System.out.println(hour + ": " + hourCounts[hour]);
        }
    }

    /**
     * Print the lines of data read by the LogfileReader
     */
    private void printData() {
        reader.printData();
    }
}
```

which reads data from a file called *weblog.txt*. This file stores the following information

```
year month day hour minute
```

eg, the line

```
2011 05 01 00 10
```

indicates an access at 00h10 on 1 May, 2011.

Exercise

1. Explore the *webloganalyzer* project by creating a **`LogAnalyzer`** object and calling its **`analyzeHourlyData`** method. Follow that with a call to its **`printHourlyCounts`** method, which will print the results of the analysis. What are the busiest times of day?

We'll look at how to use arrays through this project.

## Declaring array variables

An array is declared in the following line from **`LogAnalyzer`**

```
    private final int[] hourCounts;
```

The square brackets **int[]** indicate that the **hourCounts** variable is an array containing items of type **int**. The above expression only *declares* an array, it does not create it. An array can only store items of its *base type*, eg, **int**s. Why do array declarations use square brackets, **[** and **]**, whereas type declarations use angle brackets, **<** and **>**? Answer: for, as they say, *historical reasons*!

Exercises

2. Write a declaration of an array variable **people** that could store objects of type **Person**.
3. Write a declaration for an array variable **vacant** that could be used to refer to an array of **boolean** variables.
4. Read through the **LogAnalyzer** class and identify all the places where the **hourCounts** variable is used. Note how often a pair of square brackets is used with the variable.
5. What is wrong with the following array declarations? Correct them.

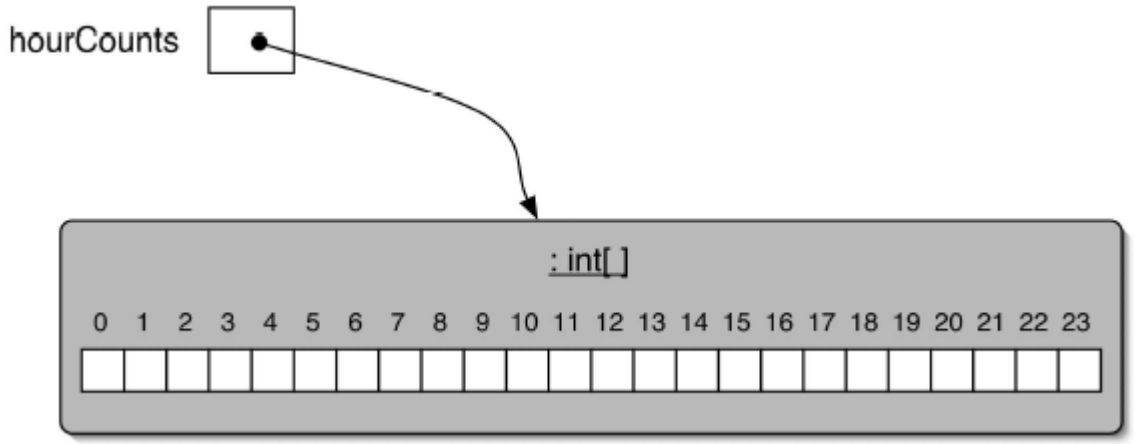
```
[]int counts;
boolean[5000] occupied;
```

## Creating array objects

The constructor of **LogAnalyzer** contains the line

```
hourCounts = new int[24];
```

which creates an array object capable of storing 24 **int**s and assigns its reference to the variable **hourCounts**.




The above expression just creates the array, not the elements stored in the array.

The general form of array construction is

```
new type[integer expression]
```

eg,

```
String[] names = new String[10];
```

Exercises

6. Given the following variable declarations:

```
double[] readings;
String[] urls;
TicketMachine[] machines;
```

write assignments that accomplish the following tasks:

a. make the **readings** variable refer to an array that is able to hold 60 **double** values,

b. make the **urls** variable refer to an array that is able to hold 90 **String** objects,

c. make the **machines** variable refer to an array that is able to hold five **TicketMachine** objects.

7. How many **String** objects are created by the following declaration?

```
String[] labels = new String[20];
```

8. What is wrong with the following array creation?

```
double prices[] = new prices(50);
```

Correct it.

## Using array objects

Elements of an array are referred to by their *index*, ie, their position in the array.

```
labels[6]
machines[0]
people[x + 10 - y]
```

The valid range of indices for an array of **n** elements is **0..(n - 1)**.

Indices can be used on the left-hand side or right-hand side of an expression, eg,

```
labels[5] = "Quit";
double half = readings[0] / 2;
System.out.println(people[3].getName());
machines[0] = new TicketMachine(500);
```

## Analyzing the log file

The **LogfileReader** class gets information from the log file. We don't really care about the details. All we need to know in the **LogAnalyzer** class is that **LogfileReader** has methods

```
public boolean hasMoreEntries()
public LogEntry nextEntry()
```

which look and work like the **hasNext** and **next** methods of an iterator. The **LogAnalyzer** has the following expression

```
hourCounts[hour]++;
```

which increments the value of the **hourCounts** array at index **hour**. This is actually a short form for either of the following equivalent expressions

```
hourCounts[hour] = hourCounts[hour] + 1;
hourCounts[hour] += 1;
```

## The for loop

The for loop can be easier to use than the while under dertain circumstances

- we wish to execute a set of statements a fixed number of times,
- the iteration variable changes by a fixed amount at each iteration.

The general form is

```
for (intialization; condition; post-body action) {
    statements
}
```

eg, in the **LogAnalyzer** class

```
for (int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

The **for** is really just a convenient form of **while** loop, since the above could have been equivalently written

```
initialization
while (condition) {
    statements
    post-body action
}
```

eg,

```
int hour = 0;
while (hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

Note in the *condition* **hour < hourCounts.length** that the array **hourCount** has a *variable* (not a method) **length** storing the number of elements in the array. Why? Because.

Note also that for-each loops also work with arrays.

Exercises

9. What happens if line 47 in the code above (in the **printHourlyCounts** method) is changed to

```
for (int hour = 0; hour <= hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

   What would you need to change to make the code work keeping the **<=** operator?

10. Rewrite the body of **printHourlyCounts** so that the for loop is replaced by an equivalent while loop. Call the rewritten method to check that it prints the same result as before.

11. Correct all the errors in the following method.

```
/**
 * Print all the values in the marks array that are greater
 * than mean.
 * @param marks An array of mark values.
 * @param mean The mean (average) mark.
 */
void printGreater(double marks, double mean) {
    for (index = 0; index <= marks.length; index++) {
        if (marks[index] > mean) {
```

```
                System.out.println(marks[index]);
            }
        }
    }
```

Re-write the above with a for-each loop.

12. Modify the **LogAnalyzer** class so that the constructor has an argument for the name of the log file to be analyzed. Have the constructor pass the file name on to the constructor of the **LogFileReader** class. Create some random data with the **LogFileCreator** and then analyze it.

13. Complete the **numberOfAccesses** method, below, to count the total number of accesses recorded in the log file. Complete it by using a for loop to iterate over **hourCounts**:

```
/**
 * Return the number of accesses recorded in the log
 * file.
 */
int numberOfAccesses() {
    int total = 0;
    // Add the value of each element in hourCounts
    // to total.
    ...
    return total;
}
```

Write a second version using a for-each loop;

14. Add your **numberOfAccesses** method to the **LogAnalyzer** class and check that it gives the correct result. *Hint:* You can simplify your checking by having the analyzer read log files containing just a few lines of data. The **LogFileReader** class has a constructor with the following signature to read from a particular file:

```
/**
 * Create a LogFileReader that will supply data
 * from a particular log file.
 * @param filename The file of log data.
 */
LogFileReader(String filename)
```

15. Add a method **busiestHour** to **LogAnalyzer** that returns the busiest hour and its access count. You can do this by looking through the **hourCounts** array to find the element with the biggest count. *Hint*: Do you need to check every element to see if you have found the busiest hour? If so, use a for loop.

16. Add a method **quietestHour** to **LogAnalyzer** that returns the number of the least busy hour and its access count.

17. Which hour is returned by your **busiestHour** method if more than one hour has the biggest count?

18. Add a method **busiestTwoHours** to **LogAnalyzer** that finds which two-hour period is the busiest. Return the value of the first hour of this period and their total access count.

19. Save the weblog-analyzer project under a different name so that you can develop a new version that performs a more extensive analysis of the available data. For instance, it would be useful to know which days tend to be quieter than others, Are there any seven-day cyclical patterns, for instance? In order to perform analysis of daily, monthly, or yearly data, you will need to make some changes to the LogEntry

class. This already stores all the values from a single log line, but only the hour and minute values are available via accessors. Add further methods that make the remaining fields available in a similar way. Then add a range of additional analysis methods to the analyzer.

20. If you have completed the previous exercise, you could extend the log-file format with additional numerical fields. For instance, servers commonly store a numerical code that indicates whether an access was successful or not. The value 200 stands for a successful access, 403 means that access to the document was forbidden, and 404 means that the document could not be found. Have the analyzer provide information on the number of successful and unsuccessful accesses. This exercise is likely to be very challenging, as it will require you to make changes to every class in the project.

# Summary

## Concept summary

### collections

Collection objects are objects that can store an arbitary number of other objects.

### loop

A loop can be used to execute a block of statements repeatedly without having to write them multiple times.

### iterator

An iterator is an object that provides functionality to iterate over all elements of a collection.

### null

The Java reserved word **`null`** is used to mean *no object* when an object variable is not currently referring to a particular object. A field that has not explicitly been initialized will contain the value **`null`** by default.

### array

An array is a special type of collection that can store a fixed number of elements.

## Additional exercises

Exercises

21. In the *labclasses* project from the *Objects and classes* lab session, the **`LabClass`** class includes a **`students`** field to maintain a collection of **`Student`** objects. Read through the **`LabClass`** class in order to reinforce some of the concepts we have discussed in this lab session.

22. The **`LabClass`** class enforces a limit to the number of students who may be enrolled in a particular tutorial group. In view of this, do you think it would be more appropriate to use a fixed-size array rather than a flexible-size collection for the **`students`** field? Give reasons both for and against the alternatives.

23. Rewrite the **`listAllFiles`** method in the **`MusicOrganizer`** class from *musicorganizer.v3* by using a for loop rather than a for-each loop.