

GLOBAL
EDITION

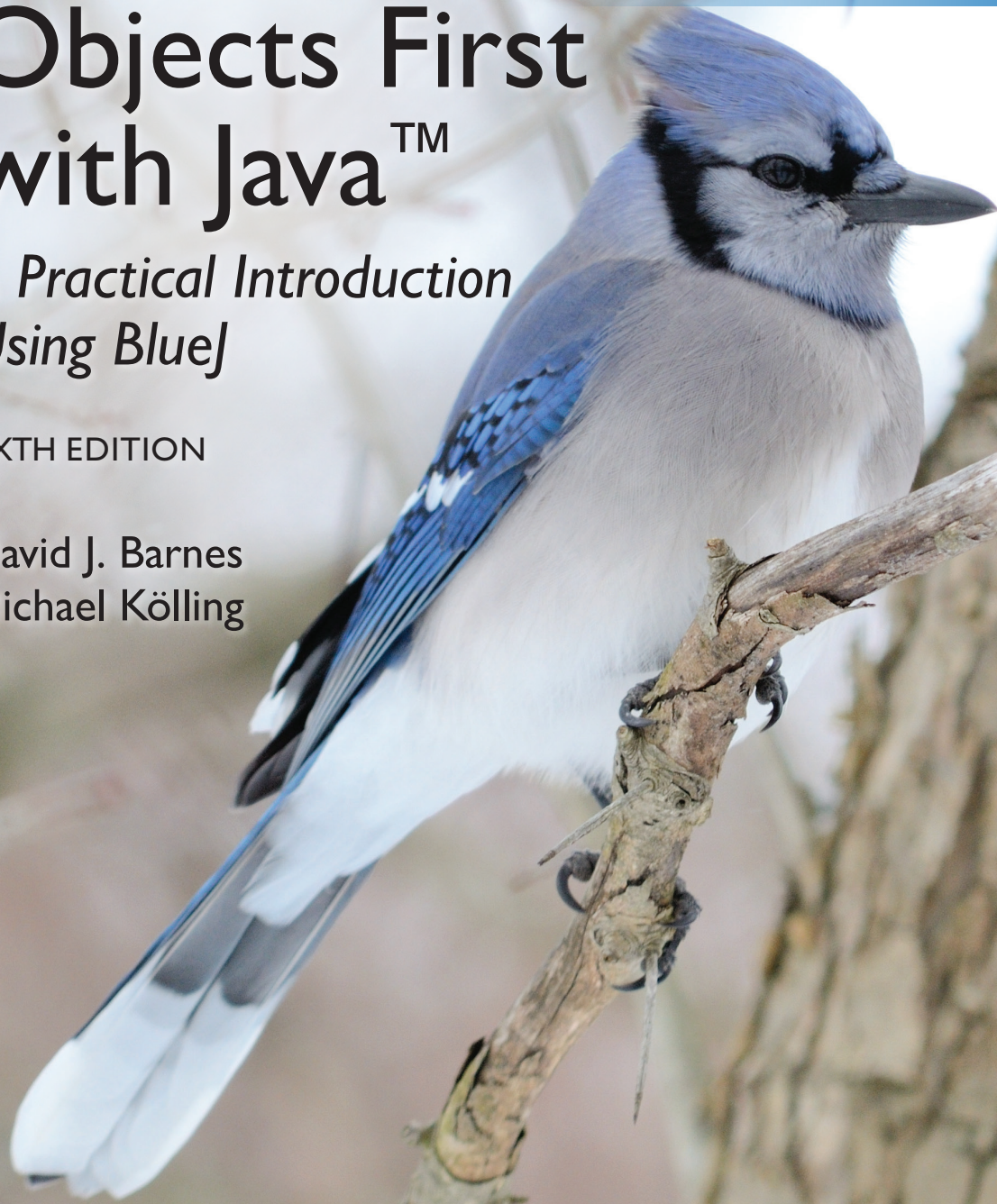


Objects First with JavaTM

*A Practical Introduction
Using BlueJ*

SIXTH EDITION

David J. Barnes
Michael Kölling



ALWAYS LEARNING

PEARSON



Objects First with Java™

A Practical Introduction Using BlueJ

David J. Barnes and Michael Kölling

University of Kent

Sixth Edition

Global Edition

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

write loops to process the elements in a collection. Instead, we apply a *pipeline* of operations to a stream derived from a collection, in order to transform the sequence into the form we want. Each operation in the pipeline defines what is to be done with a single element of the sequence. The most common sequence transformations are performed via the `filter`, `map`, and `reduce` methods.

Operations in a pipeline often take *lambdas* as parameters in order to specify what is to be done with each element of the sequence. A lambda is an anonymous function that takes zero or more parameters and has a block of code that performs the same role as a method body.

Looking at the *animal-monitoring-v1* project that you have been working on, you may have noticed that there are two methods left using for-each loops that we have not rewritten yet using streams. These are the methods that return new collections as method results; we have not yet discussed how to create a new collection from a stream. We will come back to this in Chapter 6, where we shall see how to create a new collection out of the transformed sequence.

Terms introduced in this chapter:

functional programming, lambda, stream, pipeline, filter, map, reduce

Concept summary

- **lambda** A lambda is a segment of code that can be stored and executed later.
- **functional style** In the functional style of collection processing, We do not retrieve each element to operate on it. Instead, we pass a code segment to the collection to be applied to each element.
- **streams** Streams unify the processing of elements of a collection and other sets of data. A stream provides useful methods to manipulate these data sets.
- **filter** We can filter a stream to select only specific elements.
- **map** We can map a stream to a new stream, where each element of the original stream is replaced with a new element derived from the original.
- **reduce** We can reduce a stream; reducing means to apply a function that takes a whole stream and delivers a single result.
- **pipeline** A pipeline is the combination of two or more stream functions in a chain, where each function is applied in turns.

Exercise 5.29 Take a copy of *music-organizer-v5* from Chapter 4. Rewrite the `listAllTracks` and `listByArtist` methods of the `MusicOrganizer` class to use streams and lambdas.

This page intentionally left blank

Main concepts discussed in this chapter:

- using library classes
- reading documentation
- writing documentation

Java constructs discussed in this chapter:

String, **Random**, **HashMap**, **HashSet**, **Iterator**, **static**, **final**, autoboxing, wrapper classes

In Chapter 4, we introduced the class **ArrayList** from the Java class library. We discussed how this enabled us to do something that would otherwise be hard to achieve (in this case, storing an arbitrary number of objects).

This was just a single example of a useful class from the Java library. The library consists of thousands of classes, many of which are generally useful for your work, and many of which you may probably never use.

For a good Java programmer, it is essential to be able to work with the Java library and make informed choices about which classes to use. Once you have started work with the library, you will quickly see that it enables you to perform many tasks more easily than you would otherwise have been able to. Learning to work with library classes is the main topic of this chapter.

The items in the library are not just a set of unrelated, arbitrary classes that we all have to learn individually, but are often arranged in relationships, exploiting common characteristics. Here, we again encounter the concept of abstraction to help us deal with a large amount of information. Some of the most important parts of the library are the collections, of which the **ArrayList** class is one. We will discuss other sorts of collections in this chapter and see that they share many attributes among themselves, so that we can often abstract from the individual details of a specific collection and talk about collection classes in general.

New collection classes, as well as some other useful classes, will be introduced and discussed. Throughout this chapter, we will work on the construction of a single application (the *TechSupport* system), which makes use of various different library classes. A complete

implementation containing all the ideas and source code discussed here, as well as several intermediate versions, is included in the book projects. While this enables you to study the complete solution, you are encouraged to follow the path through the exercises in this chapter. These will, after a brief look at the complete program, start with a very simple initial version of the project, then gradually develop and implement the complete solution.

The application makes use of several new library classes and techniques—each requiring study individually—such as hash maps, sets, string tokenization, and further use of random numbers. You should be aware that this is not a chapter to be read and understood in a single day, but that it contains several sections that deserve a few days of study each. Overall, when you finally reach the end and have managed to undertake the implementation suggested in the exercises, you will have learned about a good variety of important topics.

6.1

Documentation for library classes

Concept

The Java standard class library contains many classes that are very useful. It is important to know how to use the library.

The Java standard class library is extensive. It consists of thousands of classes, each of which has many methods, both with and without parameters, and with and without return types. It is impossible to memorize them all and all of the details that go with them. Instead, a good Java programmer should know:

- some of the most important classes and their methods from the library by name (**ArrayList** is one of those important ones) and
- how to find out about other classes and look up the details (such as methods and parameters).

In this chapter, we will introduce some of the important classes from the class library, and further library classes will be introduced throughout the book. More importantly, we will show you how you can explore and understand the library on your own. This will enable you to write much more interesting programs. Fortunately, the Java library is quite well documented. This documentation is available in HTML format (so that it can be read in a web browser). We shall use this to find out about the library classes.

Reading and understanding the documentation is the first part of our introduction to library classes. We will take this approach a step further, and also discuss how to prepare our own classes so that other people can use them in the same way as they would use standard library classes. This is important for real-world software development, where teams have to deal with large projects and maintenance of software over time.

One thing you may have noted about the **ArrayList** class is that we used it without ever looking at the source code. We did not check how it was implemented. That was not necessary for utilizing its functionality. All we needed to know was the name of the class, the names of the methods, the parameters and return types of those methods, and what exactly these methods do. We did not really care how the work was done. This is typical for the use of library classes.

The same is true for other classes in larger software projects. Typically, several people work together on a project by working on different parts. Each programmer should concentrate on her own area and need not understand the details of all the other parts (we discussed this in Section 3.2 where we talked about abstraction and modularization).

In effect, each programmer should be able to use the classes of other team members as if they were library classes, making informed use of them without the need to know how they work internally.

For this to work, each team member must write documentation about his class similar to the documentation for the Java standard library, which enables other people to use the class without the need to read the code. This topic will also be discussed in this chapter.

6.2 The *TechSupport* system

As always, we shall explore these issues with an example. This time, we shall use the *TechSupport* application. You can find it in the book projects under the name *tech-support1*.

TechSupport is a program intended to provide technical support for customers of the fictitious DodgySoft software company. Some time ago, DodgySoft had a technical support department with people sitting at telephones. Customers could call to get advice and help with their technical problems with the DodgySoft software products. Recently, though, business has not been going so well, and DodgySoft decided to get rid of the technical support department to save money. They now want to develop the *TechSupport* system to give the impression that support is still provided. The system is supposed to mimic the responses a technical-support person might give. Customers can communicate with the technical-support system online.

6.2.1 Exploring the *TechSupport* system

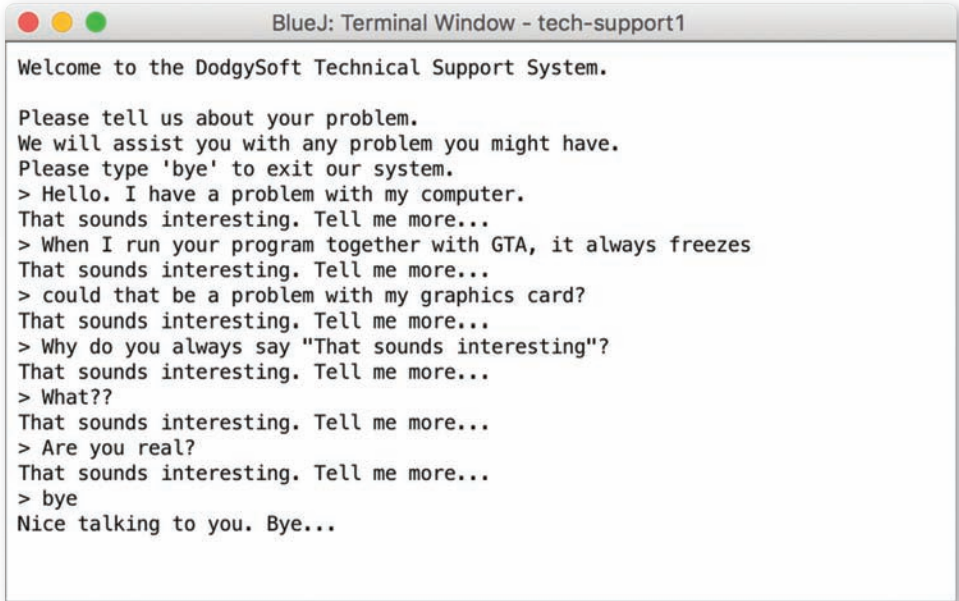
Exercise 6.1 Open and run the project *tech-support-complete*. You run it by creating an object of class **SupportSystem** and calling its **start** method. Enter some problems you might be having with your software to try out the system. See how it behaves. Type “bye” when you are done. You do not need to examine the source code at this stage. This project is the complete solution that we will have developed by the end of this chapter. The purpose of this exercise is only to give you an idea of what we plan to achieve.

Eliza The idea of the *TechSupport* project is based on the ground-breaking artificial intelligence program, Eliza, developed by Joseph Weizenbaum at Massachusetts Institute of Technology in the 1960s. You can find out more about the original program by searching the web for “Eliza” and “Weizenbaum.”

We will now start our more detailed exploration by using the *tech-support1* project. It is a first, rudimentary implementation of our system. We will improve it throughout the chapter. This way, we should arrive at a better understanding of the whole system than we would by just reading the complete solution.

Figure 6.1

A first *TechSupport* dialog



```
BlueJ: Terminal Window - tech-support1

Welcome to the DodgySoft Technical Support System.

Please tell us about your problem.
We will assist you with any problem you might have.
Please type 'bye' to exit our system.
> Hello. I have a problem with my computer.
That sounds interesting. Tell me more...
> When I run your program together with GTA, it always freezes
That sounds interesting. Tell me more...
> could that be a problem with my graphics card?
That sounds interesting. Tell me more...
> Why do you always say "That sounds interesting"?
That sounds interesting. Tell me more...
> What??
That sounds interesting. Tell me more...
> Are you real?
That sounds interesting. Tell me more...
> bye
Nice talking to you. Bye...
```

In Exercise 6.1, you have seen that the program essentially holds a dialog with the user. The user can type in a question, and the system responds. Try the same with our prototype version of the project, *tech-support1*.

In *TechSupport*'s complete version, the system manages to produce reasonably varied responses—sometimes they even seem to make sense! In the prototype version we are using as a starting point, the responses are much more restricted (Figure 6.1). You will notice very quickly that the response is always the same:

"That sounds interesting. Tell me more ..."

This is, in fact, not very interesting at all, and not very convincing when trying to pretend that we have a technical-support person sitting at the other end of the dialog. We will shortly try to improve this. However, before we do this, we shall explore further what we have so far.

The project diagram shows us three classes: **SupportSystem**, **InputReader**, and **Responder** (Figure 6.2). **SupportSystem** is the main class, which uses the **InputReader** to get some input from the terminal and the **Responder** to generate a response.

Examine the **InputReader** further by creating an object of this class and then looking at the object's methods. You will see that it has only a single method available, called **getInput**, which returns a string. Try it out. This method lets you type a line of input in the terminal, then returns whatever you typed as a method result. We will not examine how this works internally at this point, but just note that the **InputReader** has a **getInput** method that returns a string.

Do the same with the **Responder** class. You will find that it has a **generateResponse** method that always returns the string **"That sounds interesting. Tell me more ..."**. This explains what we saw in the dialog earlier.

Now let us look at the **SupportSystem** class a bit more closely.