More Sophisticated Behaviour

Java version

Objectives

To start developing professional code.

Writing human-understandable code

This is not an option for this course, this is obligatory! Adopt one of the coding conventions and stick to it; we'll follow the code conventions from Javasoft.

Main concepts discussed in this chapter

- using library classes
- writing documentation
- reading documentation

Resources

- Classes needed for this lab - *chapter06.jar*.

To do

Documentation for library classes

Java, like most object-oriented languages, has an extensive class library. This is definitely a Good Thing™ as it means that we don't need to write our applications completely from scratch - we can rely on the thousands of existing classes in the library. Even better, we don't really need to know how the library classes work internally, we just need to know what they do. For example, we've used the **`ArrayList`** successfully without knowing anything at all about its implementation, eg, that deep down it stores its objects in an array.

However, we do need to know where to find information about the library classes. Similarly, for our professional applications, we need to be able to write helpful documentation for our classes. Thus others can eventually maintain and extend our applications. This would be very hard without proper documentation.

The TechSupport system

The example is that of an automated technical support system for a company called *DodgySoft*, which has just laid off all its technical support staff in order to enhance shareholder value.

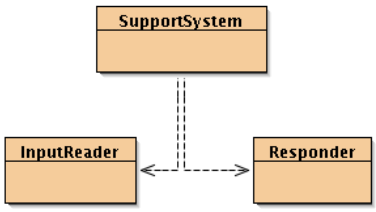
The system is based on a pioneering artificial intelligence (AI) program called *Eliza* written by Joseph Weizenbaum at MIT in the 1960's.

Exploring the TechSupport system

Exercise

1. Open and run the *techsupport.complete* package (run the **`Main`** class). To try out the system, enter some problems you might be having with your software. See how it behaves. Type **`bye`** when you are done (unfortunately the system is monolingual English).

We will actually start with a very simplified version of tech support, in the package *techsupport.v1*. Open this project now and see what it does. Not very interesting is it? In fact, this system consists of three classes



The **`Responder`** class prepares a reply to clients' questions, and in this version it always gives the same reply to every question!

```
package techsupport.v1;

/**
 * The responder class represents a response generator object.
 * It is used to generate an automatic response to an input string.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    0.1 (2016.02.29)
 */
class Responder {
    /**
     * Construct a Responder - nothing to do
     */
    Responder() {
    }

    /**
     * Generate a response.
     * @return   A string that should be displayed as the response
     */
    String generateResponse() {
        return "That sounds interesting. Tell me more...";
    }
```

```
}
```

Reading the code

```
package techsupport.v1;

/**
 * This class implements a technical support system. It is the top level
class
 * in this project. The support system communicates via text input/output
 * in the text terminal.
 *
 * This class uses an object of class InputReader to read input from the
user,
 * and an object of class Responder to generate responses. It contains a
loop
 * that repeatedly reads input and generates output until the users wants
to
 * leave.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    0.1 (2016.02.29)
 */
public class SupportSystem {
    private final InputReader reader;
    private final Responder responder;

    /**
     * Creates a technical support system.
     */
    public SupportSystem() {
        reader = new InputReader();
        responder = new Responder();
    }

    /**
     * Start the technical support system. This will print a welcome
     * message and enter into a dialog with the user, until the user
     * ends the dialog.
     */
    public void start() {
        boolean finished = false;

        printWelcome();

        while (!finished) {
            String input = reader.getInput();

            if (input.startsWith("bye")) {
                finished = true;
```

```
            } else {
                String response = responder.generateResponse();
                System.out.println(response);
            }
        }

        printGoodbye();
    }

    /**
     * Print a welcome message to the screen.
     */
    private void printWelcome() {
        System.out.println("Welcome to the DodgySoft Technical Support
System.");
        System.out.println();
        System.out.println("Please tell us about your problem.");
        System.out.println("We will assist you with any problem you might
have.");
        System.out.println("Please type 'bye' to exit our system.");
    }

    /**
     * Print a good-bye message to the screen.
     */
    private void printGoodbye() {
        System.out.println("Nice talking to you. Bye...");
    }
}
```

The **start** method of the **SupportSystem** class above contains some interesting code whose general structure is as follows

```
boolean finished = false;

while (!finished) {
    // do stuff

    if (exit condition) {
        finished = true;
    } else {
        // do more stuff
    }
}
```

Here the variable **finished** serves as a flag to indicate whether control should exit the loop. Inside the loop,

```
// do stuff
```

could be replaced by the code

```
String input = reader.getInput();
// code omitted
String response = responder.generateResponse();
System.out.println(response);
```

which repeatedly gets the client's input, prepares a reply, and prints the reply. The code

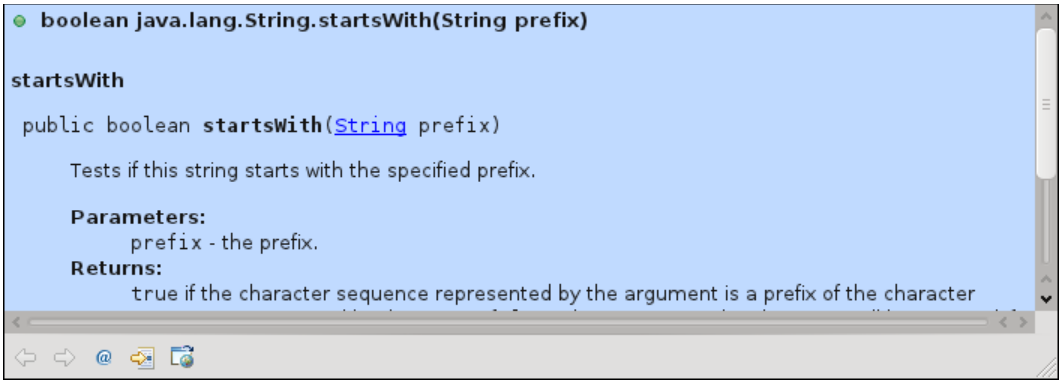
```
String input = reader.getInput();

if (input.startsWith("bye")) {
    finished = true;
}
```

determines whether or not to exit the loop. But...where does the method **`startsWith`** come from and what does it do? Read on...

Reading class documentation

Generally if your software development system is configured properly, you should be able to access some Java class documentation by hovering the cursor over a method or class name, eg, in Eclipse:



For more information, look at the online documentation for the class **`java.lang.String`**, ie, the class **`String`** in the package **`java.lang`**.

Exercises

2. Investigate the **`String`** documentation. Then look at the documentation for some other classes. What is the structure of class documentation? Which sections are common to all class descriptions? What is their purpose?
3. Look up the **`startsWith`** method in the documentation for **`String`**. There are two versions. Describe in your own words what they do and the differences between them.
4. Is there a method in the **`String`** class that tests whether a string ends with a given suffix? If so, what is it called and what are its parameters and return type?

5. Is there a method in the **`String`** class that returns the number of characters in a string? If so, what is it called and what are its parameters?

Interfaces versus implementation

The word *interface* has several meanings in Java. When we talk of the *interface* of a class here, we mean its publicly visible parts, ie, the information in the class documentation. (There is also a structure called an **`interface`** which we will see later. The interface describes what a class does. The implementation of a class refers to its source code. The implementation details how a class does what it does.

Using library-class methods

The loop exit condition of this first version of tech support is not very tolerant, only "bye" will work, not "Bye" nor " bye". We will now fix that by using appropriate methods from the library.

Exercises

7. Find the **`trim`** method in the **`String`** class's documentation. Write down the signature of that method. What does the documentation say about different characters at the beginning and end of the string?
8. Using the **`trim`** method of the **`String`** class, improve the code of the **`SupportSystem`** class in the *techsupport.v1* package so that it accepts, eg, the input strings " bye", " bye ", etc.
9. Improve the code of the **`SupportSystem`** class so that letter-case in the input is ignored, eg, both "Bye" and "BYE" will be accepted.

The code

```
String input = reader.getInput();
input = input.trim();
```

can be condensed into a single line

```
String input = reader.getInput().trim();
```

Checking string equality

Checking string equality in Java contains some traps. For example, it might seem reasonable to write the exit condition check as follows

```
if (input == "bye") {// does not always work!}
```

Unfortunately this would not work. The **`==`** relation checks for identity rather than equality - whether two objects are identical, ie, stored in the same memory location. Thus for example

```
new String("bye") == new String("bye")
```

evaluates to **`false`** since we are comparing two distinct objects that just happen to both contain the same characters. What we want is the **`equals`** method of **`String`** which compares two strings character by character and returns **`true`** iff all the corresponding characters are the same, so that for example

```
new String("bye").equals(new String("bye"))
```

evaluates to **`true`**. Our exit condition could thus be

```
if ("bye".equals(input)) {// …}
```

Exercises

10. Find the **`equals`** method in the documentation for the class **`String`**. What is the return type of this method?
11. Change the implementation to use the **`equals`** method of the class **`String`** instead of **`startsWith`**.

Adding random behaviour

We've slightly improved the input to the tech support system, now we'll try to make the output a bit more interesting by making the system's replies random.

The `Random` class

Exercises

12. Find the **`Random`** class in the Java library documentation. Which package is it in? What does it do? How do you construct an instance? How do you generate a random number? Note that you will probably not understand everything that is stated in the documentation. Just try to find out what you need to know.
13. Try to write a small code fragment (on paper) that generates a random integer number using this class.
14. Write some code to test the generation of random numbers. To do this, create a new class called **`RandomTester`**. You can create this class in the *techsupport1* package, or you can create a new package for it. In class **`RandomTester`** implement two methods: **`printOneRandom`** (which prints out one random number) and **`printMultiRandom(int howMany)`** (which has a parameter to specify how many random numbers you want, and then prints out the appropriate number of random numbers).

    Note: Your class should create only one single instance of the class **`Random`** (in its constructor) and store it in a field. *Do not create a new **`Random`** instance every time you want a new number.* The number might not be very random otherwise.

Random numbers with limited range

Exercises

15. Find the **`nextInt`** method in class **`Random`** that allows the target range of random numbers to be specified. What are the possible random numbers that are generated when you call this method with 100 as its parameter?
16. Write a method in your **`RandomTester`** class called **`throwDice`** that returns a random number between 1 and 6 (inclusive).
17. Write a method called **`getResponse`** that randomly returns one of the strings "yes", "no", or "maybe".

18. Expand your **getResponse** method so that it uses an **ArrayList** to store an arbitrary number of responses and randomly returns one of them.
19. Add a method to your **RandomTester** class that takes a parameter **max** and generates a random number **n** in the range 1 ≤ n ≤ max.
20. Add a method to your **RandomTester** class that takes two parameters, **min** and **max**, and generates a random number **n** in the range min ≤ n ≤ max. Rewrite the body of the method you wrote for the previous exercise so that it now calls this new method to generate its result.

Generating random responses

To improve the responses from the tech support system, we will modify the **Responder** class as follows:

- declare a field of type **Random** to hold the random number generator;
- declare a field of type **ArrayList** to hold our possible responses;
- create the **Random** and **ArrayList** objects in the **Responder** constructor;
- fill the responses list with some phrases;
- select and return a random phrase when **generateResponse** is called.

These improvements to the **Responder** class might look like the following code.

```
package techsupport.v2;

import java.util.ArrayList;
import java.util.Random;

/**
 * The responder class represents a response generator object. It is used
 * to generate an automatic response. This is the second version of this
 * class. This time, we generate some random behavior by randomly selecting
 * a phrase from a predefined list of responses.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    0.2 (2016.02.29)
 */
class Responder {
    private final Random randomGenerator;
    private final ArrayList<String> responses;

    /**
     * Construct a Responder
     */
    Responder() {
        randomGenerator = new Random();
        responses = new ArrayList<>();
        fillResponses();
    }

    /**
     * Generate a response.
```

```
     *
     * @return  A string that should be displayed as the response
     */
    String generateResponse() {
        // Pick a random number for the index in the default response
        // list. The number will be between 0 (inclusive) and the size
        // of the list (exclusive).
        int index = randomGenerator.nextInt(responses.size());
        return responses.get(index);
    }

    /**
     * Build up a list of default responses from which we can pick one
     * if we don't know what else to say.
     */
    private void fillResponses() {
        responses.add("That sounds odd. Could you describe this in more
detail?");
        responses.add("No other customer has ever complained about this \n"
+
                      "before. What is your system configuration?");
        responses.add("I need a bit more information on that.");
        responses.add("Have you checked that you do not have a dll
conflict?");
        responses.add("That is covered in the manual. Have you read the
manual?");
        responses.add("Your description is a bit wishy-washy. Have you
got \n" +
                      "an expert there with you who could describe this
better?");
        responses.add("That's not a bug, it's a feature!");
        responses.add("Could you elaborate on that?");
        responses.add("Have you tried running the app on your phone?");
        responses.add("I just checked StackOverflow - they don't know
either.");
    }
}
```

Looking more closely at the **generateResponse** method

```
     String generateResponse() {
         int index = randomGenerator.nextInt(responses.size());
         return responses.get(index);
     }
```

The first expression does the following:

- gets the size of the response list by calling its **size** method;
- generates a 0 ≤ random number < size;
- stores the random number in the local variable **index**.

The second expression:

- retrieves the response at position **`index`** using the **`get`** method;
- returns the selected **`String`**.

Exercises

21. Implement the random-response improvements discussed above to the **`Responder`** class from the *techsupport.v1* package.
22. What happens if you add more (or fewer) possible responses to the responses list? Will the selection of a random response still work properly? Why or why not?

Reading documentation for parametrized classes

When the documentation for a class starts with **`Class ArrayList<E>`**, this means that **`ArrayList`** is a *parametrized* class, or a *generic* class. The **`<E>`** indicates that the class takes a parameter that must be passed when the class is used, eg, as we've already seen

```
private final ArrayList<String> notes;  // line 1
private final ArrayList<Lot> lots;      // line 2
```

In the documentation for **`ArrayList<E>`**'s methods,

```
boolean add(E o)
E get(int index)
```

When **`ArrayList<E>`** is declared as in line 1 above, this is as if the methods were actually

```
boolean add(String o)
String get(int index)
```

and if it's declared as in line 2 then the methods correspond to

```
boolean add(Lot o)
Lot get(int index)
```

The parameter is replaced by the actual type used.

Packages and Import

Java library classes are structured into packages, eg, **`Random`** is in the package **`java.util`**; **`String`** is in **`java.lang`**, etc. To make library classes available to your code, they must be imported. The first two lines of **`Responder`**

```
import java.util.ArrayList;
import java.util.Random;
```

allow us to refer to these classes as just **`ArrayList`** and **`Random`**. Classes in the package **`java.lang`** don't

need to be imported.
All classes in a package can also be imported with a single statement

```
import java.util.*;
```

However, it is considered to be better style to explicitely import classes separately as this contributes to the documentation of the code.

Using maps for associations

We now have a solution to our technical-support system that generates random responses. This is better than our first version, but is still not very convincing. In particular, the input of the user does not influence the response in any way. It is this area that we now want to improve.

The plan is that we shall have a set of words that are likely to occur in typical questions and we will associate these words with particular responses. If the input from the user contains one of our known words, we can generate a related response. This is still a very crude method, because it does not pick up any of the meaning of the user's input, nor does it recognize a context, but it can be surprisingly effective. And it is a good next step.

To do this, we will use a **`HashMap`.** You will find the documentation for the class **`HashMap`** in the Java library documentation. **`HashMap`** is a specialization of a **`Map`**, which you will also find documented. You will see that you need to read the documentation of both to understand what a **`HashMap`** is and how it works.

Exercises

23. What is a **`HashMap`**? What is its purpose and how do you use it? Answer these questions using the Java library documentation of **`Map`** and **`HashMap`** for your responses. Note that you will find it hard to understand everything, as the documentation for these classes is not very good. We will discuss the details later in this chapter, but see what you can find out on your own before reading on.
24. **`HashMap`** is a parameterized class. List those of its methods that depend on the types used to parameterize it. Do you think the same type could be used for both of its parameters?

The concept of a map

A map is a collection of key/value pairs of objects. As with the **`ArrayList`**, a map can store a flexible number of entries. One difference between the **`ArrayList`** and a **`Map`** is that with a **`Map`** each entry is not an object, but a pair of objects. This pair consists of a key object and a value object.

Instead of looking up entries in this collection using an integer index (as we did with the **`ArrayList`**), we use the key object to look up the value object.

An everyday example of a map is a telephone directory. A telephone directory contains entries, and each entry is a pair: a name and a phone number. You use a phone book by looking up a name and getting a phone number. We do not use an index—the position of the entry in the phone book—to find it.

A map can be organized in such a way that looking up a value for a key is easy. In the case of a phone book, this is done using alphabetical sorting. By storing the entries in the alphabetical order of their keys, finding the key and looking up the value is easy. Reverse lookup (finding the key for a value—i.e., finding the name for a given phone number) is not so easy with a map. As with a phone book, reverse lookup in a map is possible, but it takes a comparatively long time. Thus, maps are ideal for a one-way lookup, where we know the lookup key and need to know a value associated with this key.

Using a HashMap

HashMap is a particular implementation of **Map.** The most important methods of the **HashMap** class are **put** and **get.**

The **put** method inserts an entry into the map, and **get** retrieves the value for a given key. The following code fragment creates a **HashMap** and inserts three entries into it. Each entry is a key/value pair consisting of a name and a telephone number.

```
HashMap<String, String> phoneBook = new HashMap<>();
phoneBook.put("Charles Nguyen", "(531) 9392 4587");
phoneBook.put("Lisa Jones", "(402) 4536 4674");
phoneBook.put("William H. Smith", "(998) 5488 0123");
```

As we saw with **ArrayList**, when declaring a **HashMap** variable and creating a **HashMap** object, we have to say what type of objects will be stored in the map and, additionally, what type of objects will be used for the key. For the phone book, we will use strings for both the keys and the values, but the two types will sometimes be different.

The following code will find the phone number for Lisa Jones and print it out.

```
String number = phoneBook.get("Lisa Jones");
System.out.println(number);
```

Note that you pass the key (the name “Lisa Jones”) to the get method in order to receive the value (the phone number).

Exercises

25. How do you check how many entries are contained in a map?
26. Create a class **MapTester** (either in your current project or in a new project). In it, use a **HashMap** to implement a phone book similar to the one in the example above. (Remember that you must import **java.util.HashMap**.) In this class, implement two methods:

    public void enterNumber(String name, String number)
    and

    public String lookupNumber(String name)
    The methods should use the **put** and **get** methods of the **HashMap** class to implement their functionality.
27. What happens when you add an entry to a map with a key that already exists in the map?
28. What happens when you add an entry to a map with two different keys?
29. How do you check whether a given key is contained in a map? (Give a Java code example.)
30. What happens when you try to look up a value and the key does not exist in the map?
31. How do you check how many entries are contained in a map?
32. How do you print out all keys currently stored in a map?

Using a map for the TechSupport system

In the **tech support** system, we can make good use of a map by using known words as keys and associated responses as values. The code below shows an example in which a **HashMap** named **responseMap** is created

and three entries are made. For example, the word "slow" is associated with the text

> *"I think this has to do with your hardware. Upgrading your processor should solve all performance problems. Have you got a problem with our software?"*

Now, whenever somebody enters a question containing the word "slow," we can look up and print out this response. Note that the response string in the source code spans several lines but is concatenated with the + operator, so a single string is entered as a value into the **HashMap.**

```
private final HashMap<String, String> responseMap;
...
Responder() {
    responseMap = new HashMap<>();
    fillResponseMap();
}

/**
* Enter all the known keywords and their associated
* responses into our response map.
*/
private void fillResponseMap() {
    responseMap.put("slow",
        "I think this has to do with your hardware. \n" +
        "Upgrading your processor should solve all " +
        "performance problems. \n" +
        "Have you got a problem with our software?");
    responseMap.put("bug",}
        "Well, you know, all software has some bugs. \n" +
        "But our software engineers are working very " +
        "hard to fix them. \n" +
        "Can you describe the problem a bit further?");
    responseMap.put("expensive",
        "The cost of our product is quite competitive. \n" +
        "Have you looked around and " +
        "really compared our features?");
}
```

A first attempt at writing a method to generate the responses could now look like the **generateResponse** method below. Here, to simplify things for the moment, we assume that only a single word (for example, "slow") is entered by the user.

```
String generateResponse(String word) {
    String response = responseMap.get(word);
    if(response != null) {
        return response;
    } else {
        // If we get here, the word was not recognized. In
        // this case, we pick one of our default responses.
        return pickDefaultResponse();
    }
}
```

In this code fragment, we look up the word entered by the user in our response map. If we find an entry, we use this entry as the response. If we don't find an entry for that word, we call a method called **pickDefaultResponse.** This method can now contain the code of our previous version of **generateResponse**, which randomly picks one of the default responses. The new logic, then, is that we pick an appropriate response if we recognize a word, or a random response out of our list of default responses if we don't.

Exercise

33. Implement the changes discussed here in your own version of the **tech support** system. Test it to get a feel for how well it works.

This approach of associating keywords with responses works quite well as long as the user does not enter complete questions, but only single words. The final improvement to complete the application is to let the user enter complete questions again and then pick matching responses if we recognize any of the words in the questions.

This poses the problem of recognizing the keywords in the sentence that was entered by the user. In the current version, the user input is returned by the **InputReader** as a single string. We shall now change this to a new version in which the **InputReader** returns the input as a set of words. Technically, this will be a set of strings, where each string in the set represents a single word that was entered by the user.

If we can do that, then we can pass the whole set to the Responder, which can then check every word in the set to see whether it is known and has an associated response.

To achieve this in Java, we need to know about two things: how to cut a single string containing a whole sentence into words and how to use sets. These two issues are discussed in the next two sections.

Using sets

The Java standard library includes different variations of sets implemented in different classes. The class we shall use is called **HashSet.**

Exercise

34. What are the similarities and differences between a **HashSet** and an **ArrayList**? Use the descriptions of **Set**, **HashSet**, **List**, and **ArrayList** in the library documentation to find out, because **HashSet** is a special case of a **Set** and **ArrayList** is a special case of a **List.**

The two types of functionality that we need are the ability to enter elements into the set and retrieve the elements later. Fortunately, these tasks contain hardly anything new for us. Consider the following code fragment:

```
import java.util.HashSet;
      ...
      HashSet<String> mySet = new HashSet<>();
      mySet.add("one");
      mySet.add("two");
      mySet.add("three");
```

Compare this code with the statements needed to enter elements into an **ArrayList.** There is almost no difference, except that we create a **HashSet** this time instead of an **ArrayList.** Now let us look at iterating over all elements:

```
      for(String item : mySet) {
          Do something with that item.
      }
```

Again, these statements are the same as the ones we used to iterate over an **`ArrayList`**.

In short, using collections in Java is quite similar for different types of collections. Once you understand how to use one of them, you can use them all. The differences really lie in the behavior of each collection. A list, for example, will keep all elements entered in the desired order, provides access to elements by index, and can contain the same element multiple times.

A set, on the other hand, does not maintain any specific order (the elements may be returned in a for-each loop in a different order from that in which they were entered) and ensures that each element is in the set at most once. Entering an element a second time simply has no effect.

`List`, `Map`, and `Set`

It is tempting to assume that a **`HashSet`** must be used in a similar way to a **`HashMap`**. In fact, as we have illustrated, a **`HashSet`** is actually much closer in usage to an **`ArrayList`**.

When trying to understand how the various collection classes are used, it helps to pay close attention to their names. The names consist of two parts, e.g.: "**`Array`**" "**`List`**." The second half tells us what kind of collection we are dealing with (**`List`**, **`Map`**, **`Set`**), and the first tells us how it is implemented (for instance, using an array).

For using collections, the type of the collection (the second part) is the more important. We have discussed before that we can often abstract from the implementation; we do not need to think about it much. Thus, for our purposes, a **`HashSet`** and a **`TreeSet`** are very similar. They are both sets, so they behave in the same way. The difference is only in their implementation, which is important only when we start thinking about efficiency: one implementation will perform some operations much faster than another. However, efficiency concerns come much later, and only when we have either very large collections or applications in which performance is critical.

Dividing strings

Now that we have seen how to use a set, we can investigate how we can cut the input string into separate words to be stored in a set of words. The solution is shown in a new version of the **`InputReader`**'s **`getInput`** method.

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @return A set of Strings, where each String is one of the
 *         words typed by the user
*/
HashSet<String> getInput() {
    System.out.print("> ");  // print prompt
    String inputLine = reader.nextLine().trim().toLowerCase();
    String[] wordArray = inputLine.split(" ");

    // add words from array into hashset
    HashSet<String> words = new HashSet<>();
    for(String word : wordArray) {
        words.add(word);
    }
    return words;
```

```
}
```

Here, in addition to using a **HashSet**, we also use the **split** method, which is a standard method of the **String** class.

The **split** method can divide a string into separate substrings and return those in an array of strings. The parameter to the **split** method defines at what kind of characters the original string should be split. We have defined that we want to cut our string at every space character.

The next few lines of code create a **HashSet** and copy the words from the array into the set before returning the set.

Exercise

35. The **split** method is more powerful than it first seems from our example. How can you define exactly how a string should be split? Give some examples.
36. How would you call the **split** method if you wanted to split a string at either space or tab characters? How might you break up a string in which the words are separated by colon characters (:)?
37. What is the difference in the result of returning the words in a **HashSet** compared with returning them in an **ArrayList**?
38. What happens if there is more than one space between two words (e.g., two or three spaces)? Is there a problem?
39. Read the footnote above about the **Arrays.asList** method. Find and read the sections in these notes about class variables and class methods. Explain in your own words how this works.

    What are examples of other methods that the **Arrays** class provides?

    Create a class called **SortingTest.** In it, create a method that accepts an array of **int** values as a parameter and prints out to the terminal the elements sorted (smallest element first).

Finishing the TechSupport system

To put everything together, we also have to adjust the **SupportSystem** and **Responder** classes to deal correctly with a set of words rather than a single string. The code below shows the new version of the start method from the **SupportSystem** class. It has not changed a great deal. The changes are:

- The input variable receiving the result from **reader.getInput()** is now of type **HashSet** .
- The check for ending the application is done using the contains method of the **HashSet** class, rather than a **String** method. (Look this method up in the documentation.)

```
public void start() {
    boolean finished = false;
    printWelcome();

    while(!finished) {
        HashSet<String> input = reader.getInput();
        if(input.contains("bye")) {
            finished = true;
        } else {
```

```
            String response = responder.generateResponse(input);
            System.out.println(response);
        }
    }
    printGoodbye();
}
```

Finally, we have to extend the **generateResponse** method in the **Responder** class to accept a set of words as a parameter. It then has to iterate over these words and check each of them with our map of known words. If any of the words is recognized, we immediately return the associated response. If we do not recognize any of the words, as before, we pick one of our default responses.

```
String generateResponse(HashSet<String> words) {
    for(String word : words) {
        String response = responseMap.get(word);
        if(response != null) {
            return response;
        }
    }
    // If we get here, none of the words from the input line was
    // recognized. In this case, we pick one of our default
    // responses.

    return pickDefaultResponse();
}
```

This is the last change to the application discussed here in this chapter. The solution in the package *techsupport.complete* contains all these changes. It also contains more associations of words to responses than are shown in this chapter.

Many more improvements to this application are possible. We shall not discuss them here. Instead, we suggest some, in the form of exercises, left to the reader. Some of these are quite challenging programming exercises.

Exercises

40. Implement the final changes discussed above in your own version of the program.
41. Add more word/response mappings into your application. You could copy some out of the solutions provided and add some yourself.
42. Ensure that the same default response is never repeated twice in a row.
43. Sometimes two words (or variations of a word) are mapped to the same response. Deal with this by mapping synonyms or related expressions to the same string so that you do not need multiple entries in the response map for the same response.
44. Identify multiple matching words in the user's input, and respond with a more appropriate answer in that case.
45. When no word is recognized, use other words from the user's input to pick a well-fitting default response: for example, words such as "why," "how," and "who."

Writing class documentation

When you used library classes, eg, **`ArrayList`**, you used the class's documentation (its interface) to figure out how it worked. You didn't have to go and read the code (its implementation)! The same should apply to your own classes, they should be sufficiently well documented so that they can be used without having to read the actual code. This is especially important in a professional context where you may be part of team working on a project, and each team member is responsible for only part of the code development. Java provides a tool, javadoc, to help in writing documentation.

Using javadoc in Eclipse

When editing a class with the Eclipse development environment, up to now you've been looking at its code. You can generate and preview the documentation by selecting *Project / Generate Javadoc...* which brings up a window with options for generating the documentation. Try generating the documentation into the *build/doc* directory with different options.

Elements of class documentation

The documentation of a class should include at least:

- the class name;
- a comment describing the overall purpose and characteristics of the class;
- the author's name (or authors' names);
- documentation for each public constructor and each public method.

The documentation for each constructor and method should include:

- the name of the method;
- the return type;
- the parameter names and types;
- a description of the purpose and function of the method;
- a description of each parameter;
- a description of the value returned.

Eclipse can be configured to generate boiler-plate documentation templates where you just have to fill in the details.

Exercise

46. Java can also generate documentation independently of Eclipse. Consult the help of the javadoc commandline and use it to generate the documentation for your tech support project.

In the source code, comments to be included in the javadoc documentation are written in the format

```
/**
   This is a javadoc comment.
 */
```

Javadoc class comments appear just before the class declaration and javadoc method comments immediately

precede a method signature. Special tags in the javadoc comments get formatted specially, eg,

- @author
- @version
- @param
- @return

Exercises

47. Find examples of javadoc tags in the source code of the tech support project. How do they influence the formatting of the documentation?
48. Find out about other javadoc tags by looking at the document from ~~Sun Microsystems~~ Oracle javadoc - *How to Write Doc Comments for the Javadoc Tool*.
49. Properly document all classes and methods in your version of the tech support project.

`public` versus `private`

Access modifiers are the keywords **`public`** and **`private`** occurring in the declarations of fields and methods

```
// field declarations
private int numberOfSeats;

// methods
public void setAge(int replacementAge) {
    ....
}

private int computeAverage() {
    ....
}
```

The **`public`** keyword means that the field or method is accessible from any class. The **`private`** keyword means that the field or method is accessible only from within the class in which it is defined. Public methods are intended to be used as library methods - the user only has to know what they do but not how they work. Public methods are part of the class documentation generated by javadoc. Private elements are intended to be visible only to the developer of the class in which they are defined - the user in this case has to know details of how these methods work. In general, fields should be private whereas methods are often public, but there are exceptions of course. Note that by default private elements don't show up in the documentation generated by javadoc, since this is intended for users more than for developers.

Access keyword	Visibility	Element used by
`public`	From any class	User using its interface
`private`	Only from class where defined	Developer using its implementation

Information hiding

Making an element of a class private means that it is strictly internal to the class. From outside the class, such internal details are completely hidden. This means that someone using the class *cannot* depend on these internal implementation details. What's the point? Hiding such details makes the class much more easily maintainable. No other code can rely on these details since they are hidden, meaning that the class maintainer can change them without the risk of breaking other code. Imagine if you had to change all of your classes which use the **`ArrayList`** class every time that ~~Sun~~ Oracle changed some internal detail of the library class! Classes are called *loosely coupled* when they only rely on each others' public interfaces, and loose coupling is definitely a Good Idea™. On the other hand, when classes depend on each others' internal details they are *tightly coupled* and this makes for difficult maintenance. Changing implementation details in one class may lead to having to make changes in many other classes.

`private` methods and `public` fields

In general, methods and constructors are public. They can however be private when they are intended for use only internally in their defining class, eg,

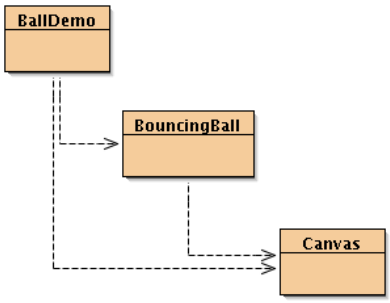
- to break up a large method into smaller submethods;
- to avoid code repetition when the same code would otherwise be used in several places.

Fields are almost always private so that they are hidden from the outside. If the field data must be accessed, then this is through accessor and mutator methods.

Learning about classes from their interfaces

The *bouncing balls* demo

We will use the *bouncingballs* project to illustrate using classes via their public interface. The project consists of three classes



The **`Canvas`** class provides a drawing window.

Exercises

57. As usual, you will need a **`Main`** class to run the code. Create a **`BallDemo`** object and execute the

drawDemo and **bounce** methods. Then read the **BallDemo** source code. Describe, in detail, how these methods work.

58. Read the documentation of the **Canvas** class. Then answer the following questions in writing, including fragments of Java code.
    a. How do you create a **Canvas**?
    b. How do you make it visible?
    c. How do you draw a line?
    d. How do you erase something?
    e. What is the difference between **draw** and **fill**?
    f. What does **wait** do?
59. Experiment with canvas operations by making changes to the **drawDemo** method of **BallDemo**. Draw some more lines, shapes, and text.
60. Draw a frame around the canvas by drawing a rectangle 20 pixels inside the window borders. Put this functionality into a method called **drawFrame** in the **BallDemo** class.
    a. You can do this by drawing four lines.
    b. You can also do this by using the draw method with a **Rectangle** object as an argument. The signature

       ```
       public void draw(Shape shape)
       ```

       says that it takes an object of type **Shape** as an argument. As we'll see later in the chapter *Improving Structure with Inheritance*, a **Rectangle** is a special type of **Shape** and a **Rectangle** object can in fact be used in place of a **Shape** object.
61. Improve your **drawFrame** method to adapt automatically to the current canvas's size (that is, do not hard-code the size of the canvas into the method). To do this, you need to find out how to make use of an object of class **Dimension**.
62. Change the method **bounce** to let the user choose how many balls should be bouncing. Use a collection to store the balls.
63. Which type of collection (**ArrayList**, **HashMap**, or **HashSet**) is most suitable for storing balls for the new **bounce** method? Discuss in writing, and justify your choice.
64. Change the **bounce** method to place the balls randomly anywhere in the top half of the screen.
65. Write a new method named **boxBounce**. This method draws a rectangle (the 'box') on the screen, and one or more balls inside the box. For the balls, do not use **bouncingBall**, but create a new class **BoxBall** that moves around inside the box, bouncing off the walls of the box so that it always stays inside. The initial **position** and **speed** of the ball should be random. The **boxBounce** method should have a parameter that specifies how many balls are in the box.
66. Give the balls in **boxBounce** random colours.

Class variables and constants

Exercise

67. In class **BouncingBall**, you will find a definition of **gravity**. Increase or decrease the **gravity** value, compile, and run the bouncing ball demo again. Do you observe any change?

The **BouncingBall** class contains the line

```
    private static final int GRAVITY = 3;
```

with two keywords, **`static`** and **`final`**.

The `static` keyword

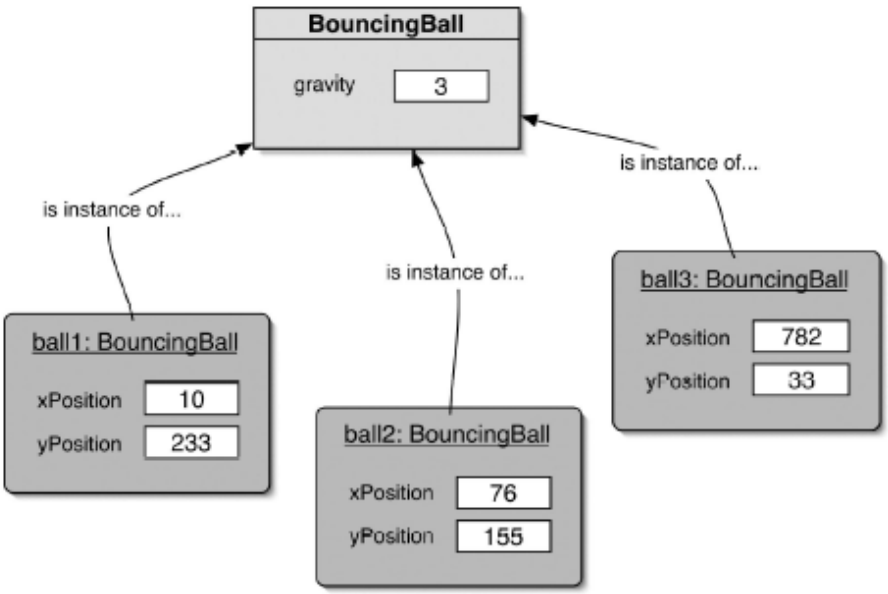
In Java, the key word **`static`** introduces *class variables*. Consider

```
class BouncingBall {
    // effect of gravity
    private static final int GRAVITY = 3;

    private int xPosition;
    private int yPosition;

    Other fields and method omitted.
}
```

Now imagine we create three **`BouncingBall`** instances. The resulting situation is as follows



Each of the three **`BouncingBall`** objects has its own independent copies of **`xPosition`** and **`yPosition`**. Changes the value of one of these instance variables for one object has no effect on its values for the other objects. **`GRAVITY`** however is a *class variable* and is stored in the class **`BouncingBall`** and not in any object. The same variable is seen by all three objects, and changing its value will change its value as seen by all objects. Source code accesses class variables just like other variables.

As we'll see later, *class methods* also exist.

Constants

The keyword **`final`** indicates that the field is a *constant*. Constants are like variables, except that once their value has been initialized it cannot be changed later. The following code would produce an error

```
private final int SIZE = 10;
// stuff omitted
SIZE = 25;  // incorrect - value cannot change
```

To indicate that they're special, constants are written in capital letters. Constants generally belong to classes.

Exercises

68. Write constant declarations for the following:
    - A public variable that is used to measure tolerance, with the value of 0.001.
    - A public variable that is used to indicate a passing mark in a course, with the integer value of 10.
    - A public variable that is used to indicate that the help command is 'h'.
69. Take a look at the **`LogEntry`** class in the *weblogana lyzer* project. How have constants been used in that class? Does that look reasonable?
70. Suppose that a change to the *weblogana lyzer* project meant that it was no longer necessary to store year values in the **`dataValues`** array in the **`LogEntryclass`**. How much of this class would need to be altered if the month value is now stored at index 0, the day value at index 1, and so on? Do you see how the use of named constants simplifies this sort of process?

Summary

Concept summary

Java library

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

library documentation

The Java standard library documentation shows details about all classes in the library. Using this documentation is essential in order to make good use of the library classes.

interface

The interface of class describes what a class does and how it can be used, without showing the implementation.

implementation

The complete source code that defines a class is called the implementation of that class

immutable

An object is said to be immutable if its contents or state cannot be changed once it has been created. Strings are an example of immutable objects.

documentation

The documentation of a class should be detailed enough for other programmers to use the class without the need to read the implementation.

access modifiers

Access modifiers define the visibility of a field, constructor, or method. Public elements are accessible from inside the same class and from other classes; private elements are accessible only form within the same class.

information hiding

Information hiding is a principle which states that internal details of a class's implementation should be hidden from other classes. It ensures better modularization of an application.

class variables, static variables

Classes can have fields. These are known as class variables or static variables. Exactly one copy exists of a class variable at all times, independent of the number of created instances.

Exercises

71. There is a rumor circulating on the Internet that George Lucas (the creator of the Star Wars movies) uses a formula to create the names for the characters in his stories (Jar Jar Binks, ObiWan Kenobi, etc.). The formula—allegedly—is this:

    Your Star Wars first name:

    1. Take the first three letters of your last name.
    2. Add to that the first two letters of your first name.

    Your Star Wars last name:

    1. Take the first two letters of your mother's maiden name.
    2. Add to this the first three letters of the name of the town or city where you were born.

    And now your task: Create a new project with a package named **`starwars`**. In it create a class named **`NameGenerator`**. This class should have a method named **`generateStarWarsName`** that generates a Star Wars name, following the method described above. You will need to find out about a method of the **`String`** class that generates a substring.

72. The following code fragment attempts to print out a string in uppercase letters:

    ```
    public void printUpper(String s) {
        s.toUpperCase();
        System.out.println(s);
    }
    ```

    This code, however, does not work. Find out why, and explain. How should it be written properly?

73. Assume that we want to swap the values of two integer variables, a and b. To do this, we write a method

```
public void swap(int i1, int i2) {
    int tmp = i1;
    i1 = i2;
    i2 = tmp;
}
```

Then we call this method with our a and b variables:

```
swap(a, b);
```

Are **a** and **b** swapped after this call? If you test it, you will notice that they are not! Why does this not work? Explain in detail.