7.13 Refactoring for language independence

One feature of the *zuul* game that we have not commented on yet is that the user interface is closely tied to commands written in English. This assumption is embedded in both the `CommandWords` class where the list of valid commands is stored, and the `Game` class where the `processCommand` method explicitly compares each command word against a set of English words. If we wish to change the interface to allow users to use a different language then we would have to find all the places in the source code where command words are used and change them. This is a further example of a form of implicit coupling, which we discussed in Section 7.9.

If we want to have language independence in the program then ideally we should have just one place in the source code where the actual text of command words is stored and have everywhere else refer to commands in a language-independent way. A programming language feature that makes this possible is *enumerated types* or *enums*. We will explore this feature of Java via the *zuul-with-enums* projects.

7.13.1 Enumerated types

Code 7.9 shows a Java enumerated type definition called `CommandWord`.

Code 7.9
An enumerated type for command words

```
/**
 * Representations for all the valid command words for the game.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2008.03.30
 */
public enum CommandWord
{
    // A value for each command word, plus one for unrecognised
    // commands.
    GO, QUIT, HELP, UNKNOWN;
}
```

In its simplest form an enumerated type definition consists of an outer wrapper that uses the word `enum` rather than `class`, and a body that is simply a list of variable names denoting the set of values that belong to this type. By convention, these variable names are fully capitalized. We never create objects of an enumerated type. In effect, each name within the type definition represents a unique instance of the type that has already been created for us to use. We refer to these instances as `CommandWord.GO`, `CommandWord.QUIT`, etc. Although the syntax for using them is similar, it is important to avoid thinking of these values as being like the numeric class constants we discussed in Section 5.13. Despite the simplicity of their definition, enumerated type values are proper objects and are not the same as integers.

How can we use the CommandWord type to make a step toward decoupling the game logic of zuul from a particular natural language? One of the first improvements we can make is to the following series of tests in the processCommand method of Game:

```
if(commandWord.equals("help")) {
    printHelp();
}
else if(commandWord.equals("go")) {
    goRoom(command);
}
else if(commandWord.equals("quit")) {
    wantToQuit = quit(command);
}
```

If commandWord is made to be of type CommandWord rather than String then this can be rewritten as:

```
if(commandWord == CommandWord.HELP) {
    printHelp();
}
else if(commandWord == CommandWord.GO) {
    goRoom(command);
}
else if(commandWord == CommandWord.QUIT) {
    wantToQuit = quit(command);
}
```

Now we just have to arrange for the user's typed commands to be mapped to the corresponding CommandWord values. Open the *zuul-with-enums-v1* project to see how we have done this. The most significant change can be found in the CommandWords class. Instead of using an array of strings to define the valid commands we now use a map between strings and CommandWord objects:

```
public CommandWords()
{
    validCommands = new HashMap<String, CommandWord>();
    validCommands.put("go", CommandWord.GO);
    validCommands.put("help", CommandWord.HELP);
    validCommands.put("quit", CommandWord.QUIT);
}
```

The command typed by a user can now easily be converted to its corresponding enumerated type value.