25.5 Declaring and Using Classes

[*Note:* This section may be read after studying Chapter 3, Introduction to Classes, Objects, Methods and Strings.]

In Section 25.3, we demonstrated basic JShell capabilities. In this section, we create a class and manipulate an object of that class. We'll use the version of class `Account` presented in Fig. 3.1.

25.5.1 Creating a Class in JShell

Start a new JShell session (or `/reset` the current one), then declare class `Account`—we ignored the comments from Fig. 3.1:

```
jshell> public class Account {
   ...>    private String name;
   ...>
   ...>    public void setName(String name) {
   ...>       this.name = name;
   ...>    }
   ...>
   ...>    public String getName() {
   ...>       return name;
```

```
      …>    }
      …> }
   |  created class Account

   jshell>
```

JShell recognizes when you enter the class’s closing brace—then displays

```
   | created class Account
```

and issues the next `jshell>` prompt. Note that the semicolons throughout class `Account`’s body are required.

To save time, rather than typing a class’s code as shown above, you can load an existing source code file into JShell, as shown in Section 25.5.6. Though you can specify access modifiers like `public` on your classes (and other types), JShell ignores all access modifiers on the top-level types except for `abstract` (discussed in Chapter 10).

Viewing Declared Classes

To view the names of the classes you’ve declared so far, enter the **`/types`** command:7

7. `/types` actually displays all types you declare, including classes, interfaces and `enum`s.

```
   jshell> /types
   |    class Account

jshell>
```

25.5.2 Explicitly Declaring Reference-Type Variables

The following creates the `Account` variable `account`:

```
jshell> Account account
account ==> null

jshell>
```

The default value of a reference-type variable is `null`.

25.5.3 Creating Objects

You can create new objects. The following creates an `Account` variable named `account` and initializes it with a new object:

```
jshell> account = new Account()
account ==> Account@56ef9176

jshell>
```

The strange notation

```
Account@56ef9176
```

is the default text representation of the new `Account` object. If a class provides a custom text representation, you'll see that instead. We show how to provide a custom text representation for objects of a class in Section 7.6. We discuss the default text representation of objects in Section 9.6. The value after the `@` symbol is the object's *hashcode*. We discuss hashcodes in Section 16.10.

Declaring an Implicit `Account` Variable Initialized with an `Account` Object

If you create an object with only the expression `new Account()`, JShell assigns the object to an implicit variable of type `Account`, as in:

```
jshell> new Account()
$4 ==> Account@1ed4004b

jshell>
```

Note that this object's hashcode (`1ed4004b`) is different from the prior `Account` object’s hashcode (`56ef9176`)—these typically are different, but that’s not guaranteed.

Viewing Declared Variables

You can view all the variables you’ve declared so far with the JShell **`/vars`** command:

```
jshell> /vars
|    Account account = Account@56ef9176
|    Account $4 = Account@1ed4004b

jshell>
```

For each variable, JShell shows the type and variable name followed by an equal sign and the variable’s text representation.

25.5.4 Manipulating Objects

Once you have an object, you can call its methods. In fact, you already did this with the `System.out` object by calling its `println`, `print` and `printf` methods in earlier snippets.

The following sets the `account` object’s name:

```
jshell> account.setName("Amanda")
jshell>
```

The method `setName` has the return type `void`, so it does not return a value and JShell does not show any additional output.

The following gets the `account` object’s name:

```
jshell> account.getName()
$6 ==> "Amanda"

jshell>
```

Method `getName` returns a `String`. When you invoke a method that returns a value, JShell stores the value in an implicitly declared variable. In this case, `$6`’s type is *inferred* to be `String`. Of course, you could have assigned the result of the preceding method call to an explicitly declared variable.

Using the Return Value of a Method in a Statement

If you invoke a method as part of a larger statement, the return value is used in that statement, rather than stored. For example, the following uses `println` to display the `account` object’s name:

```
jshell> System.out.println(account.getName())
Amanda

jshell>
```

25.5.5 Creating a Meaningful Variable Name for an Expression

You can give a meaningful variable name to a value that JShell previously assigned to an implicit variable. For example, with the following snippet recalled

```
jshell> account.getName()
```

type

$$Shift + Tab\ v$$

The $+$ notation means that you should you press *both* the *Shift* and *Tab* keys together, then release those keys and press *v*. JShell infers the expression’s type and begins a variable declaration for you—`account.getName()` returns a `String`, so JShell inserts `String` and an equal sign (`=`) before the expression, as in

```
jshell> account.getName()
jshell> String _= account.getName()
```

JShell also positions the cursor (indicated by the _ above) immediately before the = so you can simply type the variable name, as in

```
jshell> String name = account.getName()
name ==> "Amanda"

jshell>
```

When you press *Enter*, JShell evaluates the new snippet and stores the value in the specified variable.

25.5.6 Saving and Opening Code-Snippet Files

You can save all of a session’s valid code snippets to a file, which you can then load into a JShell session as needed.

Saving Snippets to a File

To save just the *valid* snippets, use the **`/save`** command, as in:

```
/save filename
```

By default, the file is created in the folder from which you launched JShell. To store the file in a different location, specify the complete path of the file.

Loading Snippets from a File

Once you save your snippets, they can be reloaded with the **`/open`** command:

```
/open filename
```

which executes each snippet in the file.

Using `/open` to Load Java Source-Code Files

You also can open existing Java source code files using `/open`. For example, let's assume you'd like to experiment with class `Account` from Fig. 3.1 (as you did in Section 25.5.1). Rather than typing its code into JShell, you can save time by loading the class from the source file `Account.java`. In a command window, you'd change to the folder containing `Account.java`, execute JShell, then use the following command to load the class declaration into JShell:

```
/open Account.java
```

To load a file from another folder, you can specify the full pathname of the file to open. In Section 25.10, we’ll show how to use existing compiled classes in JShell.