25.3 Introduction to JShell

[*Note:* This section may be read after studying Chapter 2, Introduction to Java Applications; Input/Output and Operators.]

In Chapter 2, to create a Java application, you:

1. created a class containing a `main` method.
2. declared in `main` the statements that will execute when you run the program.
3. compiled the program and fixed any compilation errors that occurred. This step had to be repeated until the program compiled without errors.
4. ran the program to see the results.

By automatically compiling and executing code as you complete each expression or statement, JShell eliminates the overhead of

- creating a class containing the code you wish to test,
- compiling the class and
- executing the class.

Instead, you can focus on interactively discovering and experimenting with Java's language and API features. If you enter code that does not compile, JShell immediately reports the errors. You can then use JShell's editing features to quickly fix and re-execute the code.

25.3.1 Starting a JShell Session

To start a JShell session in:

- Microsoft Windows, open a **Command Prompt** then type `jshell` and press *Enter*.
- macOS (formerly OS X), open a **Terminal** window then type the following command and press *Enter*.

```
$JAVA_HOME/bin/jshell
```

- Linux, open a shell window then type `jshell` and press *Enter*.

The preceding commands execute a new JShell session and display the following message and the `jshell>` **prompt**:

```
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro

jshell>
```

8

In the first line above, `"Version 9-ea"` indicates that you're using the `ea` (that is, early access) version of JDK 9. JShell precedes informational messages with vertical bars (`|`). You are now ready to enter Java code or JShell commands.

25.3.2 Executing Statements

[*Note:* As you work through this chapter, type the same code and JShell commands that we show at each `jshell>` prompt to ensure that what you see on your screen will match what we show in the sample outputs.]

JShell has two input types:

- Java code (which the JShell documentation refers to as **snippets**) and
- JShell commands.

In this section and Section 25.3.3, we begin with Java code snippets. Subsequent sections introduce JShell commands.

You can type any expression or statement at the `jshell>` prompt then press *Enter* to execute the code and see its results immediately. Consider the program of Fig. 2.1, which we show again in Fig. 25.1. To demonstrate how `System.out.println` works, this program required many lines of code and comments, which you had to write, compile and execute. Even without the comments, five code lines were still required (lines 4 and 6–9).

```
1   // Fig. 25.1: Welcome1.java
2   // Text-printing program.
3
4   public class Welcome1 {
5      // main method begins execution of Java applic
6      public static void main(String[] args) {
```

```
   7         System.out.println("Welcome to Java Program
              8      } // end method main
              9   } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 25.1

Text-printing program.

In JShell, you can execute the statement in line 7 without creating all the infrastructure of class `Welcome1` and its `main` method:

```
jshell> System.out.println("Welcome to Java Programmi
Welcome to Java Programming!

jshell>
```

In this case, JShell displays the snippet's command-line output below the initial `jshell>` prompt and the statement you entered. Per our convention, we show user inputs in bold.

Notice that we did not enter the preceding statement's semicolon (`;`). JShell adds *only* terminating semicolons.2 You need to add a semicolon if the end of the statement is not the end of the line—for example, if the statement is inside braces

({ and }). Also, if there is more than one statement on a line then you need a semicolon between statements, but not after the last statement.

2. Not requiring semicolons is one example of how JShell reinterprets standard Java for convenient interactive use. We discuss several of these throughout the chapter and summarize them in Section 25.14.

The blank line before the second `jshell>` prompt is the result of the newline displayed by method `println` and the newline that JShell always displays before each `jshell>` prompt. Using `print` rather than `println` eliminates the blank line:

```
jshell> System.out.print("Welcome to Java Programming
Welcome to Java Programming!

jshell>
```

JShell keeps track of everything you type, which can be useful for re-executing prior statements and modifying statements to update the tasks they perform.

25.3.3 Declaring Variables Explicitly

Almost anything you can declare in a typical Java source-code file also can be declared in JShell (Section 25.14 discusses some of the features you cannot use). For example, you can explicitly declare a variable as follows:

```
jshell> int number1
number1 ==> 0

jshell>
```

When you enter a variable declaration, JShell displays the variable's name (in this case, `number1`) followed by `==>` (which means, "has the value") and the variable's initial value (`0`). If you do not specify an initial value explicitly, the variable is initialized to its type's default value—in this case, `0` for an `int` variable.

A variable can be initialized in its declaration—let's redeclare `number1`:

```
jshell> int number1 = 30
number1 ==> 30

jshell>
```

JShell displays

```
number1 ==> 30
```

to indicate that `number1` now has the value `30`. When you declare a new variable with the *same name* as another variable in the current JShell session, JShell replaces the first declaration with the new one.3 Because `number1` was declared previously, we could have simply assigned `number1`

a value, as in

3. Redeclaring an existing variable is another example of how JShell reinterprets standard Java for interactive use. This behavior is different from how the Java compiler handles a new declaration of an existing variable—such a “double declaration” generates a compilation error.

```
jshell> number1 = 45
number1 ==> 45

jshell>
```

Compilation Errors in JShell

You must declare variables before using them in JShell. The following declaration of `int` variable `sum` attempts to use a variable named `number2` that we have not yet declared, so JShell reports a compilation error, indicating that the compiler was unable to find a variable named `number2`:

```
jshell> int sum = number1 + number2
|   Error:
|   cannot find symbol
|     symbol: variable number2
|   int sum = number1 + number2;
|                       ^-----^
jshell>
```

The error message uses the notation `^-----^` to highlight the error in the statement. No error is reported for the previously declared variable `number1`. Because this snippet has a compilation error, it’s invalid. However, JShell still maintains

the snippet as part of the JShell session's history, which includes valid snippets, invalid snippets and commands that you've typed. As you'll soon see, you can recall this invalid snippet and execute it again later. JShell's `/history` command displays the current session's history—that is, *everything* you've typed:

```
jshell> /history

System.out.println("Welcome to Java Programming!")
System.out.print("Welcome to Java Programming!")
int number1
int number1 = 45
number1 = 45
int sum = number1 + number2
/history

jshell>
```

Fixing the Error

JShell makes it easy to fix a prior error and re-execute a snippet. Let's fix the preceding error by first declaring `number2` with the value `72`:

```
jshell> int number2 = 72
number2 ==> 72

jshell>
```

Subsequent snippets can now use `number2`—in a moment, you'll re-execute the snippet that declared and initialized `sum` with `number1 + number2`.

Recalling and Re-executing a Previous Snippet

Now that both `number1` and `number2` are declared, we can declare the `int` variable `sum`. You can use the up and down arrow keys to navigate backward and forward through the snippets and JShell commands you've entered previously. Rather than retyping `sum`'s declaration, you can press the up arrow key three times to recall the declaration that failed previously. JShell recalls your prior inputs in reverse order—the last line of text you typed is recalled first. So, the first time you press the up arrow key, the following appears at the `jshell>` prompt:

```
jshell> int number2 = 72
```

The second time you press the up arrow key, the `/history` command appears:

```
jshell> /history
```

The third time you press the up arrow key, `sum`'s prior

declaration appears:

```
jshell> int sum = number1 + number2
```

Now you can press *Enter* to re-execute the snippet that declares and initializes `sum`:

```
jshell> int sum = number1 + number2
sum ==> 117

jshell>
```

JShell adds the values of `number1` (`45`) and `number2` (`72`), stores the result in the new `sum` variable, then shows `sum`'s value (`117`).

25.3.4 Listing and Executing Prior Snippets

You can view a list of all previous valid Java code snippets with JShell's `/list` command—JShell displays the snippets in the order you entered them:

```
jshell> /list
   1 : System.out.println("Welcome to Java Programmin
   2 : System.out.print("Welcome to Java Programming!
   4 : int number1 = 30;
   5 : number1 = 45
```

```
      6 : int number2 = 72;
      7 : int sum = number1 + number2;

   jshell>
```

Each valid snippet is identified by a sequential **snippet ID**. The snippet with ID 3 is *missing* above, because we replaced that original snippet

```
   int number1
```

with the one that has the ID 4 in the preceding `/list`. Note that `/list` may not display everything that `/history` does. As you recall, if you omit a terminating semicolon, JShell inserts it for you behind the scenes. When you say `/list`, *only* the declarations (snippets 4, 6 and 7) actually show the semicolons that JShell inserted.

Snippet 1 above is just an expression. If we type it with a terminating semicolon, it's an **expression statement**.

Executing Snippets By ID Number

You can execute any prior snippet by typing ***/id***, where *id* is the snippet's ID. For example, when you enter `/1`:

```
jshell> /1
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

JShell displays the first snippet we entered, executes it and shows the result.4 You can reexecute the last snippet you typed (whether it was valid or invalid) with `/!`:

4. At the time of this writing, you cannot use the */id* command to execute a *range* of previous snippets; however, the JShell command `/reload` can re-execute *all* existing snippets (Section 25.12.3).

```
jshell> /!
System.out.println("Welcome to Java Programming!")
Welcome to Java Programming!

jshell>
```

JShell assigns an ID to every valid snippet you execute, so even though

```
System.out.println("Welcome to Java Programming!")
```

already exists in this session as snippet `1`, JShell creates a new snippet with the next ID in sequence (in this case, `8` and `9` for the last two snippets). Executing the `/list` command shows that snippets `1`, `8` and `9` are identical:

```
jshell> /list

   1 : System.out.println("Welcome to Java Programmin
   2 : System.out.print("Welcome to Java Programming!
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programmin
   9 : System.out.println("Welcome to Java Programmin

jshell>
```

25.3.5 Evaluating Expressions and Declaring Variables Implicitly

When you enter an expression in JShell, it evaluates the expression, implicitly creates a variable and assigns the expression's value to the variable. **Implicit variables** are named $#, where # is the new snippet's ID.5 For example:

5. Implicitly declared variables are another example of how JShell reinterprets standard Java for interactive use. In regular Java programs you must explicitly declare *every* variable.

```
jshell> 11 + 5
$10 ==> 16

jshell>
```

evaluates the expression `11 + 5` and assigns the resulting value (`16`) to the implicitly declared variable `$10`, because there were nine prior valid snippets (even though one was deleted because we redeclared the variable `number1`). JShell *infers* that the type of `$10` is `int`, because the expression `11 + 5` adds two `int` values, producing an `int`. Expressions may also include one or more method calls. The list of snippets is now:

```
jshell> /list

   1 : System.out.println("Welcome to Java Programmin
   2 : System.out.print("Welcome to Java Programming!
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programmin
   9 : System.out.println("Welcome to Java Programmin
  10 : 11 + 5

jshell>
```

Note that the implicitly declared variable `$10` appears in the list simply as `10` without the `$`.

25.3.6 Using Implicitly Declared Variables

Like any other declared variable, you can use an implicitly

declared variable in an expression. For example, the following assigns to the *existing* variable `sum` the result of adding `number1` (`45`) and `$10` (`16`):

```
jshell> sum = number1 + $10
sum ==> 61

jshell>
```

The list of snippets is now:

```
jshell> /list

   1 : System.out.println("Welcome to Java Programmin
   2 : System.out.print("Welcome to Java Programming!
   4 : int number1 = 30;
   5 : number1 = 45
   6 : int number2 = 72;
   7 : int sum = number1 + number2;
   8 : System.out.println("Welcome to Java Programmin
   9 : System.out.println("Welcome to Java Programmin
  10 : 11 + 5
  11 : sum = number1 + $10

jshell>
```

25.3.7 Viewing a Variable's Value

You can view a variable's value at any time simply by typing

its name and pressing *Enter*:

```
jshell> sum
sum ==> 61

jshell>
```

JShell treats the variable name as an expression and simply evaluates its value.

25.3.8 Resetting a JShell Session

You can remove all prior code from a JShell session by entering the **`/reset`** command:

```
jshell> /reset
|   Resetting state.

jshell> /list

jshell>
```

The subsequent `/list` command shows that all prior snippets were removed. Confirmation messages displayed by JShell, such as

```
|   Resetting state.
```

are helpful when you're first becoming familiar with JShell. In Section 25.12.5, we'll show how you can change the JShell *feedback mode*, making it more or less verbose.

25.3.9 Writing Multiline Statements

Next, we write an `if` statement that determines whether `45` is less than `72`. First, let's store 45 and 72 in implicitly declared variables, as in:

```
jshell> 45
$1 ==> 45

jshell> 72
$2 ==> 72

jshell>
```

Next, begin typing the `if` statement:

```
jshell> if ($1 < $2) {
   …>
```

JShell knows that the `if` statement is incomplete, because we typed the opening left brace, but did not provide a body or a closing right brace. So, JShell displays the **continuation**

prompt …> at which you can enter more of the control statement. The following completes and evaluates the `if` statement:

```
jshell> if ($1 < $2) {
   …> System.out.printf("%d < %d%n", $1, $2);
   …> }
45 < 72

jshell>
```

In this case, a second continuation prompt appeared because the `if` statement was still missing its terminating right brace (`}`). Note that the statement-terminating semicolon (;) at the end of the `System.out.printf` statement in the `if`'s body is required. We manually indented the `if`'s body statement—JShell does *not* add spacing or braces for you as IDEs generally do. Also, JShell assigns each multiline code snippet—such as an `if` statement—only one snippet ID. The list of snippets is now:

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
          System.out.printf("%d < %d%n", $1, $2);
       }

jshell>
```

25.3.10 Editing Code Snippets

Sometimes you might want to create a new snippet, based on an existing snippet in the current JShell session. For example, suppose you want to create an `if` statement that determines whether `$1` is *greater than* `$2`. The statement that performs this task

```
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
```

is nearly identical to the `if` statement in Section 25.3.9, so it would be easier to edit the existing statement rather than typing the new one from scratch. When you edit a snippet, JShell saves the edited version as a new snippet with the next snippet ID in sequence.

Editing a Single-Line Snippet

To edit a single-line snippet, locate it with the up-arrow key, make your changes within the snippet then press *Enter* to evaluate it. See Section 25.13 for some keyboard shortcuts that can help you edit single-line snippets.

Editing a Multiline Snippet

For a larger snippet that's spread over several lines—such as a `if` statement that contains one or more statements—you can edit the entire snippet by using JShell's **`/edit`** command to open the snippet in the **JShell Edit Pad** (Fig. 25.2). The command

```
/edit
```

opens **JShell Edit Pad** and displays *all* valid code snippets you've entered so far. To edit a specific snippet, include the snippet's ID, as in

```
/edit id
```

So, the command:

```
/edit 3
```

displays the `if` statement from Section 25.3.9 in **JShell Edit Pad** (Fig. 25.2)—no snippet IDs are shown in this window. **JShell Edit Pad**'s window is *modal*—that is, while it's open, you cannot enter code snippets or commands at the JShell prompt.

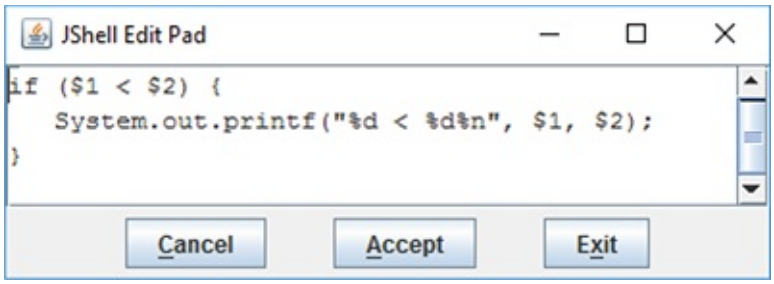

Fig. 25.2

JShell Edit Pad showing the `if` statement from Section 25.3.9.

JShell Edit Pad supports only basic editing capabilities. You can:

- click to insert the cursor at a specific position to begin typing,
- move the cursor via the arrow keys on your keyboard,
- drag the mouse to select text,
- use the *Delete* (*Backspace*) key to delete text,
- cut, copy and paste text using your operating system's keyboard shortcuts, and
- enter text, including new snippets separate from the one(s) you're editing.

In the first and second lines of the `if` statement, select each less than operator (`<`) and change it to a greater than operator (`>`), then click **Accept** to create a new `if` statement containing the edited code. When you click **Accept**, JShell also immediately evaluates the new `if` statement and displays its results (if any)—because `$1` (`45`) is *not* greater than `$2` (`72`) the `System.out.printf` statement does not execute,6 so

no additional output is shown in JShell.

6. We could have made this an `if`…`else` statement to show output when the condition is *false*, but this section is meant to be used with Chapter 2 where we introduce only the single-selection `if` statement.

If you want to return immediately to the JShell prompt, rather than clicking **Accept**, you could click **Exit** to execute the edited snippet and close **JShell Edit Pad**. Clicking **Cancel** closes **JShell Edit Pad** and discards any changes you made since the last time you clicked **Accept**, or since **JShell Edit Pad** was launched if have not yet clicked **Accept**.

When you change or create multiple snippets then click **Accept** or **Exit**, JShell compares the **JShell Edit Pad** contents with the previously saved snippets. It then executes every modified or new snippet.

Adding a New Snippet Via JShell Edit Pad

To show that **JShell Edit Pad** does, in fact, execute snippets immediately when you click **Accept**, let's change `$1`'s value to `100` by entering the following statement following the `if` statement after the other code in **JShell Edit Pad**:

```
$1 = 100
```

and clicking **Accept** (Fig. 25.3). Each time you modify a

variable's value, JShell immediately displays the variable's name and new value:

```
jshell> /edit 3
$1 ==> 100
```

Click **Exit** to close **JShell Edit Pad** and return to the `jshell>` prompt.

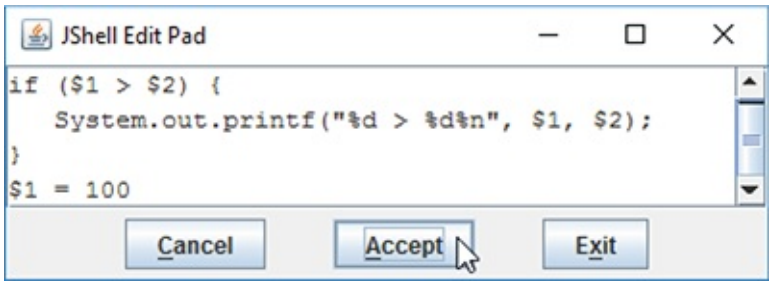

Fig. 25.3

Entering a new statement following the `if` statement in **JShell Edit Pad**.

The following lists the current snippets—notice that each multiline `if` statement has only one ID:

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
          System.out.printf("%d < %d%n", $1, $2);
```

```
          }
      4 : if ($1 > $2) {
             System.out.printf("%d > %d%n", $1, $2);
          }
      5 : $1 = 100

   jshell>
```

Executing the New `if` Statement Again

The following re-executes the new `if` statement (ID **4**) with the updated `$1` value:

```
jshell> /4
if ($1 > $2) {
   System.out.printf("%d > %d%n", $1, $2);
}
100 > 72

jshell>
```

The condition `$1` > `$2` is now `true`, so the `if` statement's body executes. The list of snippets is now

```
jshell> /list

   1 : 45
   2 : 72
   3 : if ($1 < $2) {
```

```
            System.out.printf("%d < %d%n", $1, $2);
         }
      4 : if ($1 > $2) {
            System.out.printf("%d > %d%n", $1, $2);
         }
      5 : $1 = 100
      6 : if ($1 > $2) {
            System.out.printf("%d > %d%n", $1, $2);
         }

   jshell>
```

25.3.11 Exiting JShell

To terminate the current JShell session, use the **`/exit`** command or type the keyboard shortcut *Ctrl + d* (or *control + d*). This returns you to the command-line prompt in your **Command Prompt** (in Windows), **Terminal** (in macOS) or shell (in Linux—sometimes called **Terminal**, depending on your Linux distribution).