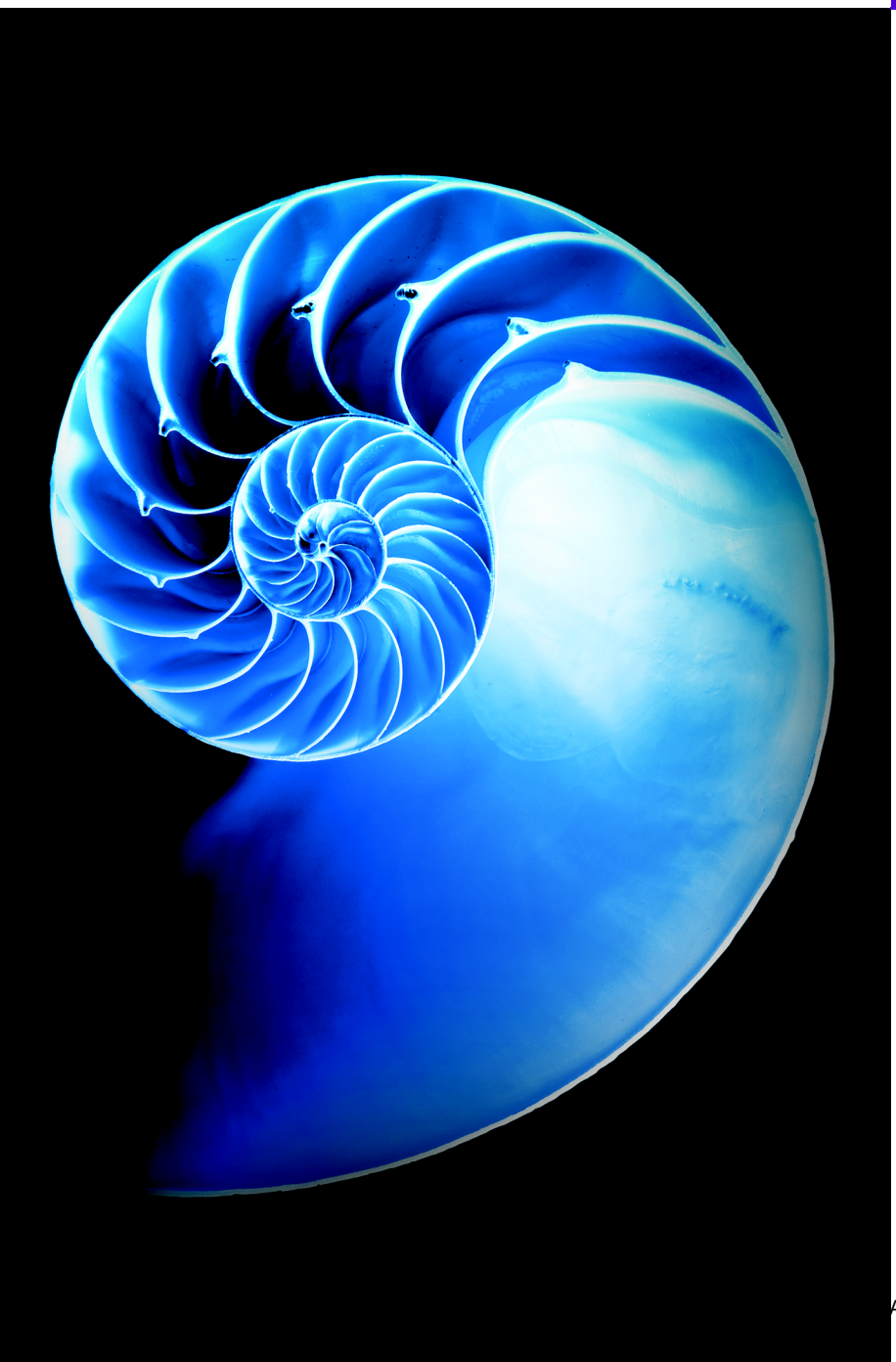
Introduction to JShell: Java 9's REPL for Interactive Java

25



Objectives

In this chapter you'll:

- See how using JShell can enhance the learning and software development processes by enabling you to explore, discover and experiment with Java language and API features.
- Start a JShell session.
- Execute code snippets.
- Declare variables explicitly.
- Evaluate expressions.
- Edit existing code snippets.
- Declare and use a class.
- Save snippets to a file.
- Open a file of JShell snippets and evaluate them.
- Auto-complete code and JShell commands.
- Display method parameters and overloads.
- Discover and explore with the Java API documentation in JShell.
- Declare and use methods.
- Forward reference a method that has not yet been declared.
- See how JShell wraps exceptions.
- Import custom packages for use in a JShell session.
- Control JShell's feedback level.

Outline

25.1 Introduction

As educators, it's a joy to write this chapter on what may be the most important pedagogic improvement in Java since its inception more than two decades ago. The Java community—by far the largest programming language community in the world—has grown to more than 10 million developers. But along the way, not much has been done to improve the learning process for novice programmers. That changes dramatically in Java 9 with the introduction of **JShell**—Java's **REPL (read-evaluate-print loop)**.[1]

1. We'd like to thank Robert Field at Oracle—the head of the JShell/REPL effort. We interacted with Mr. Field extensively as we developed Chapter 25. He answered our many questions. We reported JShell bugs and made suggestions for improvement.

Instructors have indicated a preference in introductory programming courses for languages with REPLs—and now Java has a rich REPL implementation. And with the new JShell APIs, third parties will build JShell and related interactive-development tools into the major IDEs like Eclipse, IntelliJ, NetBeans and others. Java 9 and JShell are evolving rapidly, so we've placed all our Java 9 content online—we'll keep it up-to-date as Java 9 evolves.

What is JShell?

What's the magic? It's simple. JShell provides a fast and friendly environment that enables you to quickly explore, discover and experiment with Java language features and its extensive libraries. REPLs like the one in JShell have been around for decades. In the 1960s, one of the earliest REPLs made convenient interactive development possible in the LISP programming language. Students of that era, like one of your authors, Harvey Deitel, found it fast and fun to use.

JShell replaces the tedious cycle of editing, compiling and executing with its read-evaluate-print loop. Rather than complete programs, you write **JShell commands** and Java code snippets. When you enter a snippet, JShell *immediately* **reads** it, **evaluates** it and **prints** the results that help you see the effects of your code. Then it **loops** to perform this process again for the next snippet. As you work through Chapter 25's scores of examples and exercises, you'll see how JShell and its instant feedback keep your attention, enhance your performance and speed the learning and software development processes.

Code Comes Alive

As you know, we emphasize the value of the live-code teaching approach in our books, focusing on *complete*, working programs. JShell brings this right down to the individual snippet level. Your code literally comes alive as you enter each line. Of course, you'll still make occasional errors as you enter your snippets. JShell reports compilation errors to you on a snippet-by-snippet basis. You can use this capability, for example, to test the items in our Common Programming Error tips and see the errors as they occur.

Kinds of Snippets

Snippets can be expressions, individual statements, multi-line statements and larger entities, like methods and classes. JShell supports all but a few Java features, but there are some differences designed to facilitate JShell's explore–discover–experiment capabilities. In JShell, methods do not need to be in classes, expressions and statements do not need to be in methods, and you do not need a `main` method (other differences are in Section 25.14). Eliminating this infrastructure saves you considerable time, especially compared to the lengthy repeated edit, compile and execute cycles of complete programs. And because JShell automatically displays the results of evaluating your expressions and statements, you do not need as many print statements as we use throughout this book's traditional Java code examples.

Discovery with Auto-Completion

We include a detailed treatment of **auto-completion**—a key discovery feature that speeds the coding process. After you type a portion of a name (class, method, variable, etc.) and press the *Tab* key, JShell completes the name for you or provides a list of all possible names that begin with what you've typed so far. You can then easily display method parameters and even the documentation that describes those methods.

Rapid Prototyping

Professional developers will commonly use JShell for rapid prototyping but not for full-out software development. Once you develop and test a small chunk of code, you can then paste it in to your larger project.

How This Chapter Is Organized

Chapter 25 is optional. For those who want to use JShell, the chapter has been designed as a series of units, paced to certain earlier chapters of the print book. Each unit begins with a statement like: "This section may be read after Chapter 2." So you'd begin by reading through Chapter 2, then read the corresponding section of this chapter—and similarly for subsequent chapters.

The Chapter 2 JShell Exercises

As you work your way through this chapter, execute each snippet and command in JShell to confirm that the features work as advertised. Sections 25.3–25.4 are designed to be read after Chapter 2. Once you read these sections, we recommend that you do Chapter 25's dozens of self-review exercises. JShell encourages you to "learn by doing," so the exercises have you write and test code snippets that exercise many of Chapter 2's Java features.

The self-review exercises are small and to the point, and the answers are provided to help you quickly get comfortable with JShell's capabilities. When you're done you'll have a great sense of what JShell is all about. Please tell us what you think of this new Java tool. Thanks!

Instead of rambling on about the advantages of JShell, we're going to let JShell itself convince you. If you have any questions as you work through the following examples and exercises, just write to us at `deitel@deitel.com` and we'll always respond promptly.

25.2 Installing JDK 9

Java 9 and its JShell are early access technologies that are still under development. This introduction to JShell is based on the JDK 9 Developer Preview (early access build 163). To use JShell, you must first install JDK 9, which is available in early access form at

```
https://jdk9.java.net/download/
```

The Before You Begin section that follows the Preface discusses the JDK version numbering schemes, then shows how to manage multiple JDK installations on your particular platform.

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>
```

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 application
6     public static void main(String[] args) {
7        System.out.println("Welcome to Java Programming!");
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 Programming!")
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.

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>
```

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.

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

```
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:

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> /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 Programming!")
   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 re-execute the last snippet you typed (whether it was valid or invalid) with `/!`:

```
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:

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> /list

   1 : System.out.println("Welcome to Java Programming!")
   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 Programming!")
   9 : System.out.println("Welcome to Java Programming!")

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:

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

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

jshell>
```

The list of snippets is now:

```
jshell> /list

   1 : System.out.println("Welcome to Java Programming!")
   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 Programming!")
   9 : System.out.println("Welcome to Java Programming!")
  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.

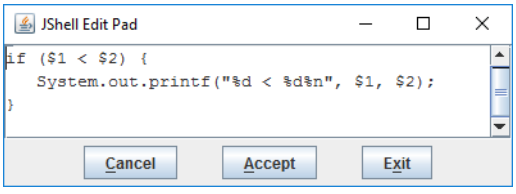


Fig. 25.2 | **JShell Edit Pad** showing the `if` statement from Section 25.3.9.

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.

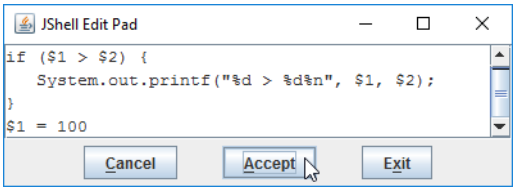


Fig. 25.3 | Entering a new statement following the `if` statement in **JShell Edit Pad**.

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).

25.4 Command-Line Input in JShell

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

In Chapter 2, we showed command-line input using a `Scanner` object:

```
Scanner input = new Scanner(System.in);

System.out.print("Enter first integer: ");
int number1 = input.nextInt();
```

We created a `Scanner`, prompted the user for input, then used `Scanner` method `nextInt` to read a value. Recall that the program then waited for you to type an integer and press *Enter* before proceeding to the next statement. The on-screen interaction appeared as:

```
Enter first integer: 45
```

This section shows what that interaction looks like in JShell.

Creating a Scanner

Start a new JShell session or `/reset` the current one, then create a `Scanner` object:

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
   \E][infinity string=\Q∞\E]

jshell>
```

You do not need to import `Scanner`. JShell automatically imports the `java.util` package and several others—we show the complete list in Section 25.10. When you create an object, JShell displays its text representation. The notation to the right of `input ==>` is the `Scanner`'s text representation (which you can simply ignore).

Prompting for Input and Reading a Value

Next, prompt the user for input:

```
jshell> System.out.print("Enter first integer: ")
Enter first integer:
jshell>
```

The statement's output is displayed immediately, followed by the next `jshell>` prompt. Now enter the input statement:

```
jshell> int number1 = input.nextInt()
_
```

At this point, JShell waits for your input. The input cursor is positioned below the `jshell>` prompt and snippet you just entered—indicated by the underscore (_) above—rather than next to the prompt `"Enter first integer:"` as it was in Chapter 2. Now type an integer and press *Enter* to assign it to `number1`—the last snippet's execution is now complete, so the next `jshell>` prompt appears.:

```
jshell> int number1 = input.nextInt()
45
number1 ==> 45

jshell>
```

Though you can use `Scanner` for command-line input in JShell, in most cases it's unnecessary. The goal of the preceding interactions was simply to store an integer value in the variable `number1`. You can accomplish that in JShell with the simple assignment

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

jshell>
```

For this reason, you'll typically use assignments, rather than command-line input in JShell. We introduced Scanner here, because sometimes you'll want to copy code you developed in JShell into a conventional Java program.

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]

```
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`.

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

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.

25.6 Discovery with JShell Auto-Completion

[*Note:* This section may be read after studying Chapter 3, Introduction to Classes, Objects, Methods and Strings, and completing Section 25.5.]

JShell can help you write code. When you partially type the name of an existing class, variable or method then press the *Tab* key, JShell does one of the following:

- If no other name matches what you've typed so far, JShell enters the rest of the name for you.
- If there are multiple names that begin with the same letters, JShell displays a list of those names to help you decide what to type next—then you can type the next letter(s) and press *Tab* again to complete the name.
- If no names match what you typed so far, JShell does nothing and your operating system's alert sound plays as feedback.

Auto-completion is normally an IDE feature, but with JShell it's IDE independent.

Let's first list the snippets we've entered since the last `/reset` (from Section 25.5):

```
jshell> /list

   1 : public class Account {
          private String name;

          public void setName(String name) {
             this.name = name;
          }

          public String getName() {
             return name;
          }
       }
   2 : Account account;
   3 : account = new Account()
   4 : new Account()
   5 : account.setName("Amanda")
   6 : account.getName()
   7 : System.out.println(account.getName())
   8 : String name = account.getName();

jshell>
```

25.6.1 Auto-Completing Identifiers

The only variable declared so far that begins with lowercase `"a"` is `account`, which was declared in snippet 2. Auto-completion is case sensitive, so `"a"` does not match the class name `Account`. If you type `"a"` at the `jshell>` prompt:

```
jshell> a
```

then press *Tab*, JShell auto-completes the name:

```
jshell> account
```

If you then enter a dot:

```
jshell> account.
```

then press *Tab*, JShell does not know what method you want to call, so it displays a list of everything—in this case, all the methods—that can appear to the right of the dot:

```
jshell> account.
equals(       getClass()    getName()     hashCode()    notify()
notifyAll()   setName(      toString()    wait(

jshell> account.
```

and follows the list with a new `jshell>` prompt that includes what you've typed so far. The list includes the methods we declared in class `Account` (snippet `1`) *and* several methods that all Java classes have (as we discuss in Chapter 9). In the list of method names

- those followed by `"()"` are methods that do not require arguments and
- those followed only by `"("` are methods that either require at least one argument or that are so-called *overloaded methods*—multiple methods with the same name, but different parameter lists (discussed in Section 6.11).

Let's assume you want to use `Account`'s `setName` method to change the name stored in the `account` object to `"John"`. There's only one method that begins with `"s"`, so you can type `s` then *Tab* to auto-complete `setName`:

```
jshell> account.setName(
```

JShell automatically inserts the method call's opening left parenthesis. Now you can complete the snippet as in:

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

jshell>
```

25.6.2 Auto-Completing JShell Commands

Auto-completion also works for JShell commands. If you type `/` then press *Tab*, JShell displays the list of JShell commands:

```
jshell> /
/!          /?          /drop       /edit       /env        /exit
/help       /history    /imports    /list       /methods    /open
/reload     /reset      /save       /set        /types      /vars

<press tab again to see synopsis>

jshell> /
```

If you then type `h` and press *Tab*, JShell displays only the commands that start with `/h`:

```
jshell> /h
/help       /history

<press tab again to see synopsis>

jshell> /h
```

Finally, if you type `"i"` and press *Tab*, JShell auto-completes `/history`. Similarly, if you type `/l` then press *Tab*, JShell auto-completes the command as `/list`, because only that command starts with `/l`.

25.7 Exploring a Class's Members and Viewing Documentation

[*Note:* This section may be read after studying Chapter 6, Methods: A Deeper Look, and the preceding portions of Chapter 25.]

The preceding section introduced basic auto-completion capabilities. When using JShell for experimentation and discovery, you'll often want to learn more about a class before using it. In this section, we'll show you how to:

- view the parameters required by a method so that you can call it correctly
- view the documentation for a method
- view the documentation for a field of a class
- view the documentation for a class, and
- view the list of overloads for a given method.

To demonstrate these features, let's explore class `Math`. Start a new JShell session or `/reset` the current one.

25.7.1 Listing Class `Math`'s `static` Members

As we discussed in Chapter 6, class `Math` contains only `static` members—`static` methods for various mathematical calculations and the `static` constants `PI` and `E`. To view a complete list, type `"Math."` then press *Tab*:

```
jshell> Math.
E                IEEEremainder(   PI               abs(
acos(            addExact(        asin(            atan(
atan2(           cbrt(            ceil(            class
copySign(        cos(             cosh(            decrementExact(
exp(             expm1(           floor(           floorDiv(
floorMod(        fma(             getExponent(     hypot(
incrementExact(  log(             log10(           log1p(
max(             min(             multiplyExact(   multiplyFull(
multiplyHigh(    negateExact(     nextAfter(       nextDown(
nextUp(          pow(             random()         rint(
round(           scalb(           signum(          sin(
sinh(            sqrt(            subtractExact(   tan(
tanh(            toDegrees(       toIntExact(      toRadians(
ulp(

jshell> Math.
```

As you know, JShell auto-completion displays a list of everything that can appear to the right of the dot (`.`). Here we typed a class name and a dot (`.`), so JShell shows only the class's `static` members. The names that are not followed by any parentheses (`E` and `PI`) are the class's `static` variables. All the other names are the class's `static` methods:

- Any method names followed by `()`—only `random` in this case—do not require any arguments.
- Any method names followed by only an opening left parenthesis, `(`, require at least one argument or are overloaded.

You can easily view the value of the constants `PI` and `E`:

```
jshell> Math.PI
$1 ==> 3.141592653589793

jshell> Math.E
$2 ==> 2.718281828459045

jshell>
```

25.7.2 Viewing a Method's Parameters

Let's assume you wish to test `Math`'s `pow` method (introduced in Section 5.4.2), but you do not know the parameters it requires. You can type

```
Math.p
```

then press *Tab* to auto-complete the name `pow`:

```
jshell> Math.pow(
```

Since there are no other methods that begin with "pow", JShell also inserts the left parenthesis to indicate the beginning of a method call. Next, you can type *Tab* to view the method's parameters:

```
jshell> Math.pow(
double Math.pow(double a, double b)

<press tab again to see documentation>

jshell> Math.pow(
```

JShell displays the method's return type, name and complete parameter list followed by the next `jshell>` prompt containing what you've typed so far. As you can see, the method requires two `double` parameters.

25.7.3 Viewing a Method's Documentation

JShell integrates the Java API documentation so you can view documentation conveniently in JShell, rather than requiring you to use a separate web browser. Suppose you'd like to learn more about `pow` before completing your code snippet. You can press *Tab* again to view the method's Java documentation (known as its javadoc)—we cut out some of the documentation text and replaced it with a vertical ellipsis (…) to save space (try the steps in your own JShell session to see the complete text):

```
jshell> Math.pow(
double Math.pow(double a, double b)
Returns the value of the first argument raised to the power of the
second argument.Special cases:
  * If the second argument is positive or negative zero, then the
    result is 1.0.
...
<press tab again to see next page>
```

For long documentation, JShell displays part of it, then shows the message

```
<press tab again to see next page>
```

You can press *Tab* to view the next page of documentation. The next `jshell>` prompt shows the portion of the snippet you've typed so far:

```
jshell> Math.pow(
```

25.7.4 Viewing a `public` Field's Documentation

You can use the *Tab* feature to learn more about a class's `public` fields. For example, if you enter `Math.PI` followed by *Tab*, JShell displays

```
jshell> Math.PI
PI

Signatures:
Math.PI:double

<press tab again to see documentation>
```

which shows `Math.PI`'s type and indicates that you can use *Tab* again to view the documentation. Doing so displays:

```
jshell> Math.PI
Math.PI:double
The double value that is closer than any other to pi, the ratio of
the circumference of a circle to its diameter.

jshell> Math.PI
```

and the next `jshell>` prompt shows the portion of the snippet you've typed so far.

25.7.5 Viewing a Class's Documentation

You also can type a class name then *Tab* to view the class's fully qualified name. For example, typing `Math` then *Tab* shows:

```
jshell> Math
Math          MathContext

Signatures:
java.lang.Math

<press tab again to see documentation>

jshell> Math
```

indicating that class `Math` is in the package `java.lang`. Typing *Tab* again shows the beginning of the class's documentation:

```
jshell> Math
java.lang.Math
The class Math contains methods for performing basic numeric opera-
tions such as the elementary exponential, logarithm, square root,
and trigonometric functions. Unlike some of the numeric methods of
...

<press tab again to see next page>
```

In this case, there is more documentation to view, so you can press *Tab* to view it. Whether or not you view the remaining documentation, the `jshell>` prompt shows the portion of the snippet you've typed so far:

```
jshell> Math
```

25.7.6 Viewing Method Overloads

Many classes have *overloaded* methods. When you press *Tab* to view an overloaded method's parameters, JShell displays the complete list of overloads, showing the parameters for every overload. For example, method `Math.abs` has four overloads:

```
jshell> Math.abs(
$1   $2

Signatures:
int Math.abs(int a)
long Math.abs(long a)
float Math.abs(float a)
double Math.abs(double a)

<press tab again to see documentation>

jshell> Math.abs(
```

When you press *Tab* again to view the documentation, JShell shows you the *first* overload's documentation:

```
jshell> Math.abs(
int Math.abs(int a)
Returns the absolute value of an int value.If the argument is not
negative, the argument is returned. If the argument is negative,
the negation of the argument is returned.
...

<press tab again to see next page>
```

You can then press *Tab* to view the documentation for the next overload in the list. Again, whether or not you view the remaining documentation, the `jshell>` prompt shows the portion of the snippet you've typed so far.

25.7.7 Exploring Members of a Specific Object

The exploration features shown in Sections 25.7.1–25.7.6 also apply to the members of a specific object. Let's create and explore a `String` object:

```
jshell> String dayName = "Monday"
dayName ==> "Monday"

jshell>
```

To view the methods you can call on the `dayName` object, type `"dayName."` and press *Tab*:

```
jshell> dayName.
charAt(                 chars()                 codePointAt(
codePointBefore(        codePointCount(         codePoints()
compareTo(              compareToIgnoreCase(    concat(
contains(               contentEquals(          endsWith(
equals(                 equalsIgnoreCase(       getBytes(
getChars(               getClass()              hashCode()
indexOf(                intern()                isEmpty()
lastIndexOf(            length()                matches(
notify()                notifyAll()             offsetByCodePoints(
regionMatches(          replace(                replaceAll(
replaceFirst(           split(                  startsWith(
subSequence(            substring(              toCharArray()
toLowerCase(            toString()              toUpperCase(
trim()                  wait(

jshell> dayName.
```

Exploring `toUpperCase`

Let's investigate the `toUpperCase` method. Continue by typing `"toU"` and pressing *Tab* to auto-complete its name:

```
jshell> dayName.toUpperCase(
toUpperCase(

jshell> dayName.toUpperCase(
```

Then, type *Tab* to view its parameters:

```
jshell> dayName.toUpperCase(
Signatures:
String String.toUpperCase(Locale locale)
String String.toUpperCase()

<press tab again to see documentation>

jshell> dayName.toUpperCase(
```

This method has two overloads. You can now use *Tab* to read about each overload, or simply choose the one you wish to use, by specifying the appropriate arguments (if any). In this case, we'll use the no-argument version to create a new `String` containing `MONDAY`, so we simply enter the closing right parenthesis of the method call and press *Enter*:

```
jshell> dayName.toUpperCase()
$2 ==> "MONDAY"

jshell>
```

Exploring `substring`

Let's assume you want to create the new `String` `"DAY"`—a subset of the implicit variable `$2`'s characters. For this purpose class `String` provides the overloaded method `substring`. First type `"$2.subs"` and press *Tab* to auto-complete its the method's name:

```
jshell> $2.substring(
substring(

jshell>
```

Next, use *Tab* to view the method's overloads:

```
jshell> $2.substring(
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)

<press tab again to see documentation>

jshell> $2.substring(
```

Next, use *Tab* again to view the first overload's documentation:

```
jshell> $2.substring(
String String.substring(int beginIndex)
Returns a string that is a substring of this string.The substring
begins with the character at the specified index and extends to the
end of this string.
...
<press tab again to see next page>
```

As you can see from the documentation, this overload of the method enables you to obtain a substring starting from a specific character index (that is, position) and continuing through the end of the `String`. The first character in the `String` is at index `0`. This is the version of the method we wish to use to obtain `"DAY"` from `"MONDAY"`, so we can return to our code snippet at the `jshell>` prompt:

```
jshell> $2.substring(
```

Finally, we can complete our call to substring and press *Enter* to view the results:

```
jshell> $2.substring(3)
$3 ==> "DAY"

jshell>
```

25.8 Declaring Methods

[*Note:* This section may be read after studying Chapter 6, Methods: A Deeper Look, and the preceding portions of Chapter 25.]

You can use JShell to prototype methods. For example, let's assume we'd like to write code that displays the cubes of the values from 1 through 10. For the purpose of this discussion, we're going to define two methods:

- Method `displayCubes` will iterate 10 times, calling method `cube` each time.
- Method `cube` will receive one int value and return the cube of that value.

25.8.1 Forward Referencing an Undeclared Method—Declaring Method `displayCubes`

Let's begin with method `displayCubes`. Start a new JShell session or `/reset` the current one, then enter the following method declaration:

```
void displayCubes() {
   for (int i = 1; i <= 10; i++) {
      System.out.println("Cube of " + i + " is " + cube(i));
   }
}
```

When you complete the method declaration, JShell displays:

```
|  created method displayCubes(), however, it cannot be invoked
until method cube(int) is declared

jshell>
```

Again, we *manually* added the indentation. Note that after you type the method body's opening left brace, JShell displays continuation prompts (`...>`) before each subsequent line until you complete the method declaration by entering its closing right brace. Also, although JShell says `"created method displayCubes()"`, it indicates that you cannot call this method until `"cube(int) is declared"`. This is *not* fatal in JShell—it recognizes that `displayCubes` depends on an undeclared method (`cube`)—this is known as **forward referencing** an undeclared method. Once you define `cube`, you can call `displayCubes`.

25.8.2 Declaring a Previously Undeclared Method

Next, let's declare method `cube`, but *purposely make a logic error* by returning the square rather than the cube of its argument:

```
jshell> int cube(int x) {
   ...>    return x * x;
   ...> }
|  created method cube(int)

jshell>
```

At this point, you can use JShell's **/methods** command to see the complete list of methods that are declared in the current JShell session:

```
jshell> /methods
|    void displayCubes()
|    int cube(int)

jshell>
```

Note that JShell displays each method's return type to the right of the parameter list.

25.8.3 Testing cube and Replacing Its Declaration

Now that method `cube` is declared, let's test it with the argument `2`:

```
jshell> cube(2)
$3 ==> 4

jshell>
```

The method correctly returns the `4` (that is, 2 * 2), based on how the method is implemented. However, our the method's purpose was to calculate the cube of the argument, so the result should have been `8` (2 * 2 * 2). You can edit `cube`'s snippet to correct the problem. Because `cube` was declared as a multiline snippet, the easiest way to edit the declaration is using **JShell Edit Pad**. You could use `/list` to determine `cube`'s snippet ID then use `/edit` followed by the ID to open the snippet. You also edit the method by specifying its name, as in:

```
jshell> /edit cube
```

In the **JShell Edit Pad** window, change `cube`'s body to:

```
   return x * x * x;
```

then press **Exit**. JShell displays:

```
jshell> /edit cube
|  modified method cube(int)

jshell>
```

25.8.4 Testing Updated Method cube and Method displayCubes

Now that method `cube` is properly declared, let's test it again with the arguments `2` and `10`:

```
jshell> cube(2)
$5 ==> 8

jshell> cube(10)
$6 ==> 1000

jshell>
```

The method properly returns the cubes of 2 (that is, 8) and 10 (that is, 1000), and stores the results in the implicit variables $5 and $6.

Now let's test displayCubes. If you type "di" and press *Tab*, JShell auto-completes the name, including the parentheses of the method call, because displayCubes receives no parameters. The following shows the results of the call:

```
jshell> displayCubes()
Cube of 1 is 1
Cube of 2 is 8
Cube of 3 is 27
Cube of 4 is 64
Cube of 5 is 125
Cube of 6 is 216
Cube of 7 is 343
Cube of 8 is 512
Cube of 9 is 729
Cube of 10 is 1000

jshell>
```

25.9 Exceptions

[*Note:* This section may be read after studying Chapter 7 and the preceding sections of Chapter 25.]

In Section 7.5, we introduced Java's exception-handling mechanism, showing how to catch an exception that occurred when we attempted to use an out-of-bounds array index. In JShell, catching exceptions is not required—it automatically catches each exception and displays information about it, then displays the next JShell prompt, so you can continue your session. This is particularly important for *checked exceptions* (Section 11.5) that are required to be caught in regular Java programs—as you know, catching an exception requires wrapping the code in a try…catch statement. By automatically, catching all exceptions, JShell makes it easier for you to *experiment* with methods that throw checked exceptions.

In the following new JShell session, we declare an array of int values, then demonstrate both valid and invalid array-access expressions:

```
jshell> int[] values = {10, 20, 30}
values ==> int[3] { 10, 20, 30 }

jshell> values[1]
$2 ==> 20

jshell> values[10]
|  java.lang.ArrayIndexOutOfBoundsException thrown: 10
|        at (#3:1)

jshell>
```

The snippet values[10] attempts to access an out-of-bounds element—recall that this results in an ArrayIndexOutOfBoundsException. Even though we did not wrap the code in a try…catch, JShell catches the exception and displays the its String representation. This

includes the exception's type and an error message (in this case, the invalid index 10), followed by a so-called stack trace indicating where the problem occurred. The notation

```
|        at (#3:1)
```

indicates that the exception occurred at line 1 of the code snippet with the ID 3. Section 6.6 discussed the method-call stack. A stack trace indicates the methods that were on the method-call stack at the time the exception occurred. A typical stack trace contains several `"at"` lines like the one shown here—one per stack frame. After displaying the stack trace, JShell shows the next `jshell>` prompt. Chapter 11 discusses stack traces in detail.

25.10 Importing Classes and Adding Packages to the CLASSPATH

[*Note:* This section may be read after studying Chapter 21, Custom Generic Data Structures and the preceding sections of Chapter 25.]

When working in JShell, you can import types from Java 9's packages. In fact, several packages are so commonly used by Java developers that JShell automatically imports them for you. (You can change this with JShell's `/set start` command—see Section 25.12.)

You can use JShell's **`/imports`** command to see the current session's list of `import` declarations. The following listing shows the packages that are auto-imported when you begin a new JShell session:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

The `java.lang` package's contents are always available in JShell, just as in any Java source-code file.

In addition to the Java API's packages, you can import your own or third-party packages to use their types in JShell. First, you use JShell's **`/env -class-path`** command to add the packages to JShell's `CLASSPATH`, which specifies where the additional packages are located. You can then use `import` declarations to experiment with the packages' contents in JShell.

Using Our Custom Generic `List` Class

In Chapter 21, we declared a custom generic `List` data structure and placed it in the package `com.deitel.datastructures`. Here, we'll add that package to JShell's `CLASSPATH`, `import` our `List` class, then use it in JShell. If you have a current JShell session open, use `/exit` to terminate it. Then, change directories to the `ch21` examples folder and start a new JShell session.

Adding the Location of a Package to the CLASSPATH

The `ch21` folder contains a folder named `com`, which is the first of a nested set of folders that represent the compiled classes in our package `com.deitel.datastructures`. The following uses adds this package to the `CLASSPATH`:

```
jshell> /env -class-path .
|  Setting new options and restoring state.

jshell>
```

The dot (`.`) indicates the current folder from which you launched JShell. You also can specify complete paths to other folders on your system or the paths of JAR (Java archive) files that contain packages of compiled classes.

Importing a Class from the Package

Now, you can import the `List` class for use in JShell. The following shows importing our `List` class and the complete list of imports in the current session:

```
jshell> import com.deitel.datastructures.List

jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
|    import com.deitel.datastructures.List

jshell>
```

Using the Imported Class

Finally, you can use class `List`. Below we create a `List<String>` and show that JShell's auto-complete capability can display the list of available methods. Then we insert two `String`s into the `List`, displaying its contents after each `insertAtFront` operation:

```
jshell> List<String> list = new List<>()
list ==> com.deitel.datastructures.List@31610302

jshell> list.
equals(           getClass()        hashCode()        insertAtBack(
insertAtFront(    isEmpty()         notify()          notifyAll()
print()           removeFromBack()  removeFromFront() toString()
wait(

jshell> list.insertAtFront("red")

jshell> list.print()
The list is: red
```

```
jshell> list.insertAtFront("blue")

jshell> list.print()
The list is: blue red

jshell>
```

A Note Regarding `import`*s*

As you saw at the beginning of this section, JShell imports the entire `java.util` package—which contains the `List` interface (Section 16.6)—in every JShell session. The Java compiler gives precedence to an explicit type `import` for our class `List` like

```
import com.deitel.datastructures.List;
```

vs. an `import`-on-demand like

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

Had we used the following `import`-on-demand:

```
import com.deitel.datastructures.*;
```

then we would have had to refer to our `List` class by its fully qualified name (that is, `com.deitel.datastructures.List`) to differentiate it from `java.util.List`.

25.11 Using an External Editor

Section 25.3.10 demonstrated **JShell Edit Pad** for editing code snippets. This tool provides only simple editing functionality. Many programmers prefer to use more powerful text editors. Using JShell's **`/set editor`** command, you can specify your preferred text editor. For example, we have a text editor named EditPlus, located on our Windows system at

```
C:\Program Files\EditPlus\editplus.exe
```

The JShell command

```
jshell> /set editor C:\Program Files\EditPlus\editplus.exe
|  Editor set to: C:\Program Files\EditPlus\editplus.exe

jshell>
```

sets `EditPlus` as the snippet editor for the current JShell session. The `/set editor` command's argument is *operating-system specific*. For example, on Ubuntu Linux, you can use the built-in `gedit` text editor with the command

```
/set editor gedit
```

and on macOS,[8] you can use the built-in `TextEdit` application with the command

```
/set editor -wait open -a TextEdit
```

Editing Snippets with a Custom Text Editor

When you're using a custom editor, each time you save snippet edits JShell immediately re-evaluates any snippets that have changed and shows their results (but not the snippets

8. On macOS, the `-wait` option is required so that JShell does not simply open the external editor, then return immediately to the next `jshell>` prompt.

themselves) in the JShell output. The following shows a new JShell session in which we set a custom editor, then performed JShell interactions—we explain momentarily the two lines of output that follow the `/edit` command:

```
jshell> /set editor C:\Program Files\EditPlus\editplus.exe
|  Editor set to: C:\Program Files\EditPlus\editplus.exe

jshell> int x = 10
x ==> 10

jshell> int y = 10
y ==> 20

jshell> /edit
y ==> 20
10 + 20 = 30
jshell> /list

   1 : int x = 10;
   3 : int y = 20;
   4 : System.out.print(x + " + " + y + " = " + (x + y))

jshell>
```

First we declared the int variables `x` and `y`, then we launched the external editor to edit our snippets. Initially, the editor shows the snippets that declare `x` and `y` (Fig. 25.4).

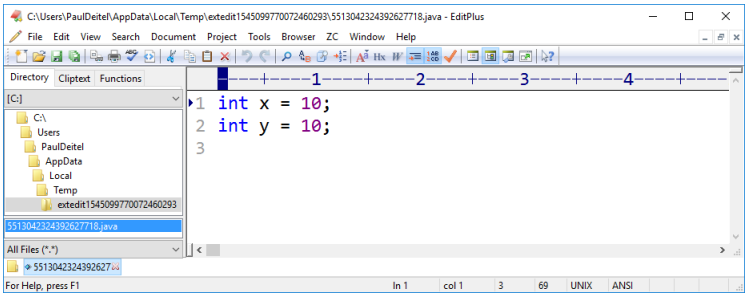


Fig. 25.4 | External editor showing code snippets to edit.

Next, we edited `y`'s declaration, giving it the new value `20`, then we added a new snippet to display both values and their sum (Fig. 25.5).

When we saved the edits in our text editor, JShell replaced `y`'s original declaration with the updated one and showed

```
y ==> 20
```

to indicate that `y`'s value changed. Then, JShell executed the new `System.out.print` snippet and showed its results

```
10 + 20 = 30
```

Finally, when we closed the external editor and pressed *Enter* in the command window, JShell displayed the next `jshell>` prompt.

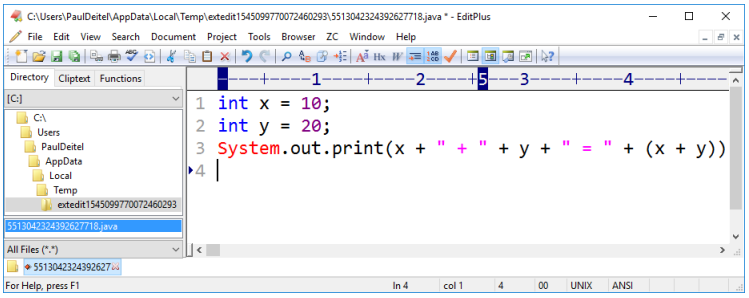


Fig. 25.5 | External editor showing code snippets to edit.

Retaining the Editor Setting

You can retain your editor setting for future JShell sessions as follows:

```
/set editor -retain commandToLaunchYourEditor
```

Restoring the JShell Edit Pad *As the Default Editor*

If you do not retain your custom editor, subsequent JShell sessions will use **JShell Edit Pad**. If you do retain the custom editor, you can restore **JShell Edit Pad** as the default with

```
/set editor -retain -default
```

25.12 Summary of JShell Commands

Figure 25.6 shows the basic JShell commands. Many of these commands have been presented throughout this chapter. Others are discussed in this section.

Command	Description
`/help` or `/?`	Displays JShell's list of commands.
`/help intro`	Displays a brief introduction to JShell.
`/help shortcuts`	Displays a description of several JShell shortcut keys.
`/list`	By default, lists the valid snippets you've entered in the current session. To list all snippets, use `/list -all`.
`/!`	Recalls and re-evaluates the last snippet.
/id	Recalls and re-evaluates the snippet with the specified *id*.
/-n	Recalls and re-evaluates a prior snippet—for *n*, `1` is the last snippet, `2` is the second-to-last, etc.
`/edit`	By default, opens a **JShell Edit Pad** window containing the valid snippets you've entered in the current session. See Section 25.11 to learn how to configure an external editor.
`/save`	Saves the current session's valid snippets to a specified file.

Fig. 25.6 | Jshell commands. (Part 1 of 2.)

Command	Description
`/open`	Opens a specified file of code snippets, loads the snippets into the current session and evaluates the loaded snippets.
`/vars`	Displays the current session's variables and their corresponding values.
`/methods`	Displays the signatures of the current session's declared methods.
`/types`	Displays types declared in the current session.
`/imports`	Displays the current session's `import` declarations.
`/exit`	Terminates the current JShell session.
`/reset`	Resets the current JShell session, deleting all code snippets.
`/reload`	Reloads a JShell session and executes the valid snippets (Section 25.12.3).
`/drop`	Deletes a specified snippet from the current session (Section 25.12.4).
`/env`	Makes changes to the JShell environment, such as adding packages or modules so you can use their types in JShell.
`/history`	Lists everything you've typed in the current JShell session, including all snippets (valid, invalid or overwritten) and JShell commands—the `/list` command shows only snippets, not JShell commands.
`/set`	Sets various JShell configuration options, such as the editor used in response to the `/edit` command, the text used for the JShell prompts, the imports to specify when a session starts, etc. (Sections 25.12.5–25.12.6).

Fig. 25.6 | Jshell commands. (Part 2 of 2.)

25.12.1 Getting Help in JShell

JShell's help documentation is incorporated directly via the **`/help`** or **`/?`** commands—`/?` is simply a shorthand for `/help`. For a quick introduction to JShell, type:

```
/help intro
```

To display JShell's list of commands, type

```
/help
```

For more information on a given command's options, type

```
/help command
```

For example

```
/help /list
```

displays the `/list` command's more detailed help documentation. Similarly

```
/help /set start
```

displays more detailed help documentation for the `/set` command's `start` option. For a list of the shortcut key combinations in JShell, type

```
/help shortcuts
```

25.12.2 `/edit` Command: Additional Features

We've discussed using `/edit` to load all valid snippets, a snippet with a specified ID or a method with a specified name into **JShell Edit Pad**. You can specify the identifier for any variable, method or type declaration that you'd like to edit. For example, if the current JShell session contains the declaration of a class named `Account`, the following loads that class into **JShell Edit Pad**:

```
/edit Account
```

25.12.3 `/reload` Command

At the time of this writing, you cannot use the */id* command to execute a range of previous snippets. However, JShell's **`/reload`** command can re-execute all valid snippets in the current session. Consider the session from Sections 25.3.9–25.3.10:

```
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>
```

The following reloads that session one snippet at a time:

```
jshell> /reload
|  Restarting and restoring state.
-: 45
-: 72
-: if ($1 < $2) {
      System.out.printf("%d < %d%n", $1, $2);
   }
45 < 72
-: if ($1 > $2) {
      System.out.printf("%d > %d%n", $1, $2);
   }
-: $1 = 100
-: if ($1 > $2) {
      System.out.printf("%d > %d%n", $1, $2);
   }
100 > 72

jshell>
```

Each reloaded snippet is preceded by `-:` and in the case of the `if` statements, the output (if any) is shown immediately following each `if` statement. If you prefer not to see the snippets as they reload, you can use the `/reload` command's `-quiet` option:

```
jshell> /reload -quiet
|  Restarting and restoring state.
45 < 72
100 > 72

jshell>
```

In this case, only the results of output statements are displayed. Then, you can view the snippets that were reloaded with the `/list` command.

25.12.4 /drop Command

You can eliminate a snippet from the current session with JShell's **`/drop`** command followed by a snippet ID or an identifier. The following new JShell session declares a variable `x` and a method `cube`, then drops `x` via its snippet ID and drops `cube` via its identifier:

```
jshell> int x = 10
x ==> 10

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> /list

   1 : int x = 10;
   2 : int cube(int y) {return y * y * y;}

jshell> /drop 1
|  dropped variable x

jshell> /drop cube
|  dropped method cube(int)

jshell> /list

jshell>
```

25.12.5 Feedback Modes

JShell has several feedback modes that determine what gets displayed after each interaction. To change the feedback mode, use JShell's **`/set feedback`** command:

```
/set feedback mode
```

where *mode* is `concise`, `normal` (the default), `silent` or `verbose`. All of the prior JShell interactions in this chapter used the `normal` mode.

Feedback Mode `verbose`

Below is a new JShell session in which we used `verbose` mode, which beginning programmers might prefer:

```
jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> int x = 10
x ==> 10
|  created variable x : int

jshell> int cube(int y) {return y * y * y;}
|  created method cube(int)

jshell> cube(x)
$3 ==> 1000
|  created scratch variable $3 : int

jshell> x = 5
x ==> 5
|  assigned to x : int

jshell> cube(x)
$5 ==> 125
|  created scratch variable $5 : int

jshell>
```

Notice the additional feedback indicating that

- variable `x` was created,
- variable `$3` was created on the first call to `cube`—JShell refers to the implicit variable as a *scratch variable*,
- an `int` was assigned to the variable `x`, and
- scratch variable `$5` was created on the second call to `cube`.

Feedback Mode `concise`

Next, we `/reset` the session then set the feedback mode to `concise` and repeat the preceding session:

```
jshell> /set feedback concise
jshell> int x = 10
jshell> int cube(int y) {return y * y * y;}
jshell> cube(x)
$3 ==> 1000
jshell> x = 5
jshell> cube(x)
$5 ==> 125
jshell>
```

As you can see, the only feedback displayed is the result of each call to `cube`. If an error occurs, its feedback also will be displayed.

Feedback Mode `silent`

Next, we `/reset` the session then set the feedback mode to `silent` and repeat the preceding session:

```
jshell> /set feedback silent
-> int x = 10
-> int cube(int y) {return y * y * y;}
-> cube(x)
-> x = 5
-> cube(x)
-> /set feedback normal
|  Feedback mode: normal

jshell>
```

In this case, the `jshell>` prompt becomes `->` and only error feedback will be displayed. You might use this mode if you've copied code from a Java source file and want to paste it into JShell, but do not want to see the feedback for each line.

25.12.6 Other JShell Features Configurable with `/set`

So far, we've demonstrated the `/set` command's capabilities for setting an external snippet editor and setting feedback modes. The `/set` command provides extensive capabilities for creating custom feedback modes via the commands:

- `/set mode`
- `/set prompt`
- `/set truncation`
- `/set format`

The `/set mode` command creates a user-defined custom feedback mode. Then you can use the other three commands to customize all aspects of JShell's feedback. The details of these commands are beyond the scope of this chapter. For more information, see JShell's help documentation for each of the preceding commands.

Customizing JShell Startup

Section 25.10 showed the set of common packages JShell `imports` at the start of each session. Using JShell's **`/set start`** command

```
/set start filename
```

you can provide a file of Java snippets and JShell commands that will be used in the current session when it restarts due to a `/reset` or `/reload` command. You can also remove all startup snippets with

```
/set start -none
```

or return to the default startup snippets with

```
/set start -default
```

In all three cases, the setting applies only to the current session unless you also include the `-retain` option. For example, the following command indicates that all subsequent JShell sessions should load the specified file of startup snippets and commands:

```
/set start -retain filename
```

You can restore the defaults for future sessions with

```
/set start -retain -default
```

25.13 Keyboard Shortcuts for Snippet Editing

In addition to the commands in Fig. 25.6, JShell supports many keyboard shortcuts for editing code, such as quickly jumping to the beginning or end of a line, or jumping between words in a line. JShell's command-line features are implemented by a library named JLine 2, which provides command-line editing and history capabilites. Figure 25.7 shows a sample of the shortcuts available.

Shortcut	Description
Ctrl + a	Move cursor to beginning of line.
Ctrl + e	Move cursor to end of line.
Alt + b	Move the cursor backwards by one word.
Alt + f	Move the cursor forwards by one word.
Ctrl + r	Perform a search for the last command or snippet containing the characters you type after typing *Ctrl + r*.
Ctrl + t	Reverse the two characters to the left of the cursor.
Ctrl + k	Cut everything from the cursor to the end of the line.
Ctrl + u	Cut everything from the beginning of the line up to, but not including the character at the cursor position.
Ctrl + w	Cut the word before the cursor.
Alt + d	Cut the word after the cursor.

Fig. 25.7 | Some keyboard shortcuts for editing the current snippet at the `jshell>` prompt.

25.14 How JShell Reinterprets Java for Interactive Use

In JShell:

- A `main` method is not required.
- Semicolons are not required on standalone statements.
- Variables do not need to be declared in classes or in methods.
- Methods do not need to be declared inside a class's body.
- Statements do not need to be written inside methods.
- Redeclaring a variable, method or type simply drops the prior declaration and replaces it with the new one, whereas the Java compiler normally would report an error.
- You do not need to catch exceptions, though you can if you need to test exception handling.
- JShell ignores top-level access modifiers (`public`, `private`, `protected`, `static`, `final`)—only `abstract` (Chapter 10) is allowed as a class modifier.
- The `synchronized` keyword (Chapter 23, Concurrency) is ignored.
- `package` statements and Java 9 `module` statements are not allowed.

25.15 IDE JShell Support

89 At the time of this writing, work is just beginning on JShell support in popular IDEs such as NetBeans, IntelliJ IDEA and Eclipse. NetBeans currently has an early access plug-in that enables you to work with JShell in both Java 8 and Java 9—even though JShell is a Java 9 feature. Some vendors will use JShell's APIs to provide developers with JShell environments that show both the code users type and the results of running that code side-by-side. Some features you might see in IDE JShell support include:

- Source-code syntax coloring for better code readability.
- Automatic source-code indentation and insertion of closing braces (`}`), parentheses (`)`) and brackets (`]`) to save programmers time.
- Debugger integration.
- Project integration, such as being able to automatically use classes in the same project from a JShell session.

25.16 Wrap-Up

9 In this chapter, you used JShell—Java 9's new interactive REPL for exploration, discovery and experimentation. We showed how to start a JShell session and work with various types of code snippets, including statements, variables, expressions, methods and classes—all without having to declare a class containing a `main` method to execute the code.

You saw that you can list the valid snippets in the current session, and recall and execute prior snippets and commands using the up and down arrow keys. You also saw that you can list the current session's variables, methods, types and `import`s. We showed how to clear the current JShell session to remove all existing snippets and how to save snippets to a file then reload them.

We demonstrated JShell's auto-completion capabilities for code and commands, and showed how you can explore a class's members and view documentation directly in JShell. We explored class `Math`, demonstrating how to list its `static` members, how to view a method's parameters and overloads, view a method's documentation and view a `public` field's documentation. We also explored the methods of a `String` object.

You declared methods and forward referenced an undeclared method that you declared later in the session, then saw that you could go back and execute the first method. We also showed that you can replace a method declaration with a new method—in fact, you can replace any declaration of a variable, method or type.

We showed that JShell catches all exceptions and simply displays a stack trace followed by the next `jshell>` prompt, so you can continue the session. You imported an existing compiled class from a package, then used that class in a JShell session.

Next, we summarized and demonstrated various other JShell commands. We showed how to configure a custom snippet editor, view JShell's help documentation, reload a session, drop snippets from a session, configure feedback modes and more. We listed some additional keyboard shortcuts for editing the current snippet at the `jshell>` prompt. Finally, we discussed how JShell reinterprets Java for interactive use and IDE support for JShell.

Self-Review Exercises

We encourage you to use JShell to do Exercises 25.1–25.43 after reading Sections 25.3–25.4. We've included the answers for all these exercises to help you get comfortable with JShell/REPL quickly.

25.1 Confirm that when you use `System.out.println` to display a `String` literal, such as `"Happy Birthday!"`, the quotes (`""`) are not displayed. End your statement with a semicolon.

25.2 Repeat Exercise 25.1, but remove the semicolon at the end of your statement to demonstrate that semicolons in this position are optional in JShell.

25.3 Confirm that JShell does not execute a `//` end-of-line comment.

25.4 Show that an executable statement enclosed in a multiline comment—delimited by `/*` and `*/`—does not execute.

25.5 Show what happens when the following code is entered in JShell:

```
/* incomplete multi-line comment
System.out.println(“Welcome to Java Programming!”)
/* complete multi-line
comment */
```

25.6 Show that indenting code with spaces does not affect statement execution.

25.7 Declare each of the following variables as type `int` in JShell to determine which are valid and which are invalid?

a) `first`
b) `first number`
c) `first1`
d) `1first`

25.8 Show that braces do not have to occur in matching pairs inside a string literal.

25.9 Show what happens when you type each of the following code snippets into JShell:

a) `System.out.println("seems OK")`
b) `System.out.println("missing something?)`
c) `System.out.println"missing something else?")`

25.10 Demonstrate that after a `System.out.print` the next print results appear on the same line right after the previous one's. [*Hint:* To demonstrate this, reset the current session, enter two `System.out.print` statements, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

25.11 Demonstrate that after a `System.out.println`, the next text that prints displays its text at the left of the next line. [*Hint:* To demonstrate this, reset the current session, enter a `System.out.println` statement followed by another print statement, then use the following two commands to save the snippets to a file, then reload and re-execute them:

```
/save mysnippets
/open mysnippets
```

The `/open` command loads the `mysnippets` file's contents then executes them.]

25.12 Demonstrate that you can reset a JShell session to remove all prior snippets and start from scratch without having to exit JShell and start a new session.

25.13 Using `System.out.println`, demonstrate that the escape sequence `\n` causes a newline to be issued to the output. Use the string

```
"Welcome\nto\nJShell!"
```

25.14 Demonstrate that the escape sequence \t causes a tab to be issued to the output. Note that your output will depend on how tabs are set on your system. Use the string

```
"before\tafter\nbefore\t\tafter"
```

25.15 Demonstrate what happens when you include a single backslash (\) in a string. Be sure that the character after the backslash does not create a valid escape sequence.

25.16 Display a string containing \\\\ (recall that \\ is an escape sequence for a backslash). How many backslashes are displayed?

25.17 Use the escape sequence `\"` to display a quoted string.

25.18 What happens when the following code executes in JShell:

```
System.out.println("Happy Birthday!\rSunny")
```

25.19 Consider the following statement

```
System.out.printf("%s%n%s%n", "Welcome to ", "Java Programming!")
```

Make the following intentional errors (separately) to see what happens.

a) Omit the parentheses around the argument list.
b) Omit the commas.
c) Omit one of the `%s%n` sequences.
d) Omit one of the strings (i.e., the second or the third argument).
e) Replace the first `%s` with `%d`.
f) Replace the string `"Welcome to "` with the integer `23`.

25.20 What happens when you enter the `/imports` command in a new JShell session?

25.21 Import class `Scanner` then create a `Scanner` object `input` for reading from `System.in`. What happens when you execute the statement:

```
int number = input.nextInt()
```

and the user enters the string `"hello"`?

25.22 In a new or `/reset` JShell session, repeat Exercise 25.21 without importing class `Scanner` to demonstrate that the package `java.util` is already imported in JShell.

25.23 Demonstrate what happens when you don't precede a `Scanner` input operation with a meaningful prompting message telling the user what to input. Enter the following statements:

```
Scanner input = new Scanner(System.in)
int value = input.nextInt()
```

25.24 Demonstrate that you can't place an `import` statement in a class.

25.25 Demonstrate that identifiers are case sensitive by declaring variables `id` and `ID` of types `String` and `int`, respectively. Also use the `/list` command to show the two snippets representing the separate variables.

25.26 Demonstrate that initialization statements like

```
String month = "April"
int age = 65
```

indeed initialize their variables with the indicated values.

25.27 Demonstrate what happens when you:

a) Add 1 to the largest possible `int` value 2,147,483,647.
b) Subtract 1 from the smallest possible integer –2,147,483,648.

25.28 Demonstrate that large integers like `1234567890` are equivalent to their counterparts with the underscore separators, namely `1_234_567_890`:

a) `1234567890 == 1_234_567_890`
b) Print each of these values and show that you get the same result.
c) Divide each of these values by 2 and show that you get the same result.

25.29 Placing spaces around operators in an arithmetic expression does not affect the value of that expression. In particular, the following expressions are equivalent:

```
17+23
17 + 23
```

Demonstrate this with an `if` statement using the condition

```
(17+23) == (17 + 23)
```

25.30 Demonstrate that the parentheses around the argument `number1 + number2` in the following statement are unnecessary:

```
System.out.printf("Sum is %d%n", (number1 + number2))
```

25.31 Declare the `int` variable `x` and initialize it to `14`, then demonstrate that the subsequent assignment `x = 27` is destructive.

25.32 Demonstrate that printing the value of the following variable is non-destructive:

```
int y = 29
```

25.33 Using the declarations:

```
int b = 7
int m = 9
```

a) Demonstrate that attempting to do algebraic multiplication by placing the variable names next to one another as in `bm` doesn't work in Java.
b) Demonstrate that the Java expression `b * m` indeed multiplies the two operands.

25.34 Use the following expressions to demonstrate that integer division yields an integer result:

a) `8 / 4`
b) `7 / 5`

25.35 Demonstrate what happens when you attempt each of the following integer divisions:

a) `0 / 1`
b) `1 / 0`
c) `0 / 0`

25.36 Demonstrate that the values of the following expressions:

a) `(3 + 4 + 5) / 5`
b) `3 + 4 + 5 / 5`

are different and thus the parentheses in the first expression are required if you want to divide the entire quantity 3 + 4 + 5 by 5.

25.37 Calculate the value of the following expression:

```
5 / 2 * 2 + 4 % 3 + 9 - 3
```

manually being careful to observe the rules of operator precedence. Confirm the result in JShell.

25.38 Test each of the two equality and four relational operators on the two values 7 and 7. For example, `7 == 7`, `7 < 7`, etc.

25.39 Repeat Exercise 25.38 using the values `7` and `9`.

25.40 Repeat Exercise 25.38 using the values `11` and `9`.

25.41 Demonstrate that accidentally placing a semicolon after the right parenthesis of the condition in an if statement can be a logic error.

```
if (3 == 5); {
   System.out.println("3 is equal to 5");
}
```

25.42 Given the following declarations:

```
int x = 1
int y = 2
int z = 3
int a
```

what are the values of a, x, y and z after the following statement executes?

```
a = x = y = z = 10
```

25.43 Manually determine the value of the following expression then use JShell to check your work:

```
(3 * 9 * (3 + (9 * 3 / (3))))
```

Answers to Self-Review Exercises

25.1

```
jshell> System.out.println("Happy Birthday!");
Happy Birthday!

jshell>
```

25.2

```
jshell> System.out.println("Happy Birthday!")
Happy Birthday!

jshell>
```

25.3

```
jshell> // comments are not executable

jshell>
```

25.4

```
jshell> /* opening line of multi-line comment
   ...>    System.out.println("Welcome to Java Programming!")
   ...>    closing line of multi-line comment */

jshell>
```

25.5 There is no compilation error, because the second /* is considered to be part of the first multi-line comment.

```
jshell> /* incomplete multi-line comment
   ...> System.out.println("Welcome to Java Programming!")
   ...> /* complete multi-line
   ...> comment */

jshell>
```

25.6

```
jshell> System.out.println("A")
A

jshell>    System.out.println("A") // indented 3 spaces
A

jshell>       System.out.println("A") // indented 6 spaces
A

jshell>
```

25.7 a) valid. b) invalid (space not allowed). c) valid. d) invalid (can't begin with a digit).

```
jshell> int first
first ==> 0

jshell> int first number
|  Error:
|  ';' expected
|  int first number
|            ^

jshell> int first1
first1 ==> 0

jshell> int 1first
|  Error:
|  '.class' expected
|  int 1first
|      ^
|  Error:
|  not a statement
|  int 1first
|        ^--^
|  Error:
|  unexpected type
|    required: value
|    found:    class
|  int 1first
|  ^--^
|  Error:
|  missing return statement
|  int 1first
|> ^---------^

jshell>
```

25.8

```
jshell> "Unmatched brace { in a string is OK"
$1 ==> "Unmatched brace { in a string is OK"

jshell>
```

25.9

```
jshell> System.out.println("seems OK")
seems OK

jshell> System.out.println("missing something?)
|  Error:
|  unclosed string literal
|  System.out.println("missing something?)
|                     ^

jshell> System.out.println"missing something else?")
|  Error:
|  ';' expected
|  System.out.println"missing something else?")
|                    ^
|  Error:
|  cannot find symbol
|    symbol:   variable println
|  System.out.println"missing something else?")
|  ^-----------------^

jshell>
```

25.10

```
jshell> System.out.print("Happy ")
Happy
jshell> System.out.print("Birthday")
Birthday
jshell> /save mysession

jshell> /open mysession
Happy Birthday
jshell>
```

25.11

```
jshell> System.out.println("Happy ")
Happy
jshell> System.out.println("Birthday")
Birthday
jshell> /save mysession

jshell> /open mysession
Happy
Birthday
jshell>
```

25.12

```
jshell> int x = 10
x ==> 10

jshell> int y = 20
y ==> 20
```

(continued...)

```
jshell> x + y
$3 ==> 30

jshell> /reset
|  Resetting state.

jshell> /list

jshell>
```

25.13

```
jshell> System.out.println("Welcome\nto\nJShell!")
Welcome
to
JShell!

jshell>
```

25.14

```
jshell> System.out.println("before\tafter\nbefore\t\tafter")
before  after
before          after

jshell>
```

25.15

```
jshell> System.out.println("Bad escap\e")
|  Error:
|  illegal escape character
|  System.out.println("Bad escap\e")
|                                ^

jshell>
```

25.16 Two.

```
jshell> System.out.println("Displaying backslashes \\\\")
Displaying backslashes \\

jshell>
```

25.17

```
jshell> System.out.println("\"This is a string in quotes\"")
"This is a string in quotes"

jshell>
```

25.18

```
jshell> System.out.println("Happy Birthday!\rSunny")
Sunny Birthday!

jshell>
```

25.19 a)

```
jshell> System.out.printf"%s%n%s%n", "Welcome to ", "Java
Programming!"
|  Error:
|  ';' expected
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|                   ^
|  Error:
|  cannot find symbol
|    symbol:   variable printf
|  System.out.printf"%s%n%s%n", "Welcome to ", "Java Programming!"
|  ^---------------^

jshell>
```

b)

```
jshell> System.out.printf("%s%n%s%n" "Welcome to " "Java
Programming!")
|  Error:
|  ')' expected
|  System.out.printf("%s%n%s%n" "Welcome to " "Java Programming!")
|                                ^

jshell>
```

c)

```
jshell> System.out.printf("%s%n", "Welcome to ", "Java Programming!")
Welcome to
$6 ==> java.io.PrintStream@6d4b1c02

jshell>
```

d)

```
jshell> System.out.printf("%s%n%s%n", "Welcome to ")
Welcome to
|  java.util.MissingFormatArgumentException thrown: Format
specifier '%s'
|        at Formatter.format (Formatter.java:2524)
|        at PrintStream.format (PrintStream.java:974)
|        at PrintStream.printf (PrintStream.java:870)
|        at (#7:1)

jshell>
```

e)

```
jshell> System.out.printf("%d%n%s%n", "Welcome to ", "Java
Programming!")
|  java.util.IllegalFormatConversionException thrown: d !=
java.lang.String
|        at Formatter$FormatSpecifier.failConversion
(Formatter.java:4275)
|        at Formatter$FormatSpecifier.printInteger
(Formatter.java:2790)
|        at Formatter$FormatSpecifier.print (Formatter.java:2744)
```

(continued...)

```
|        at Formatter.format (Formatter.java:2525)
|        at PrintStream.format (PrintStream.java:974)
|        at PrintStream.printf (PrintStream.java:870)
|        at (#8:1)

jshell>
```

f)

```
jshell> System.out.printf("%s%n%s%n", 23, "Java Programming!")
23
Java Programming!
$9 ==> java.io.PrintStream@6d4b1c02

jshell>
```

25.20

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*

jshell>
```

25.21

```
jshell> import java.util.Scanner

jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int number = input.nextInt()
hello
|  java.util.InputMismatchException thrown:
|        at Scanner.throwFor (Scanner.java:860)
|        at Scanner.next (Scanner.java:1497)
|        at Scanner.nextInt (Scanner.java:2161)
|        at Scanner.nextInt (Scanner.java:2115)
|        at (#2:1)

jshell>
```

25.22

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]
```

(continued...)

```
jshell> int number = input.nextInt()
hello
|  java.util.InputMismatchException thrown:
|        at Scanner.throwFor (Scanner.java:860)
|        at Scanner.next (Scanner.java:1497)
|        at Scanner.nextInt (Scanner.java:2161)
|        at Scanner.nextInt (Scanner.java:2115)
|        at (#2:1)

jshell>
```

25.23 JShell appears to hang while it waits for the user to type a value and press *Enter*.

```
jshell> Scanner input = new Scanner(System.in)
input ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ...
\E][infinity string=\Q∞\E]

jshell> int value = input.nextInt()
```

25.24

```
jshell> class Demonstration {
   ...>    import java.util.Scanner;
   ...> }
|  Error:
|  illegal start of type
|     import java.util.Scanner;
|     ^
|  Error:
|  <identifier> expected
|     import java.util.Scanner;
|                            ^

jshell> import java.util.Scanner

jshell> class Demonstration {
   ...> }
|  created class Demonstration

jshell>
```

25.25

```
jshell> /reset
|  Resetting state.

jshell> String id = "Natasha"
id ==> "Natasha"

jshell> int ID = 413
ID ==> 413

jshell> /list
```

(continued...)

```
   1 : String id = "Natasha";
   2 : int ID = 413;

jshell>
```

25.26

```
jshell> String month = "April"
month ==> "April"

jshell> System.out.println(month)
April

jshell> int age = 65
age ==> 65

jshell> System.out.println(age)
65

jshell>
```

25.27

```
jshell> 2147483647 + 1
$9 ==> -2147483648

jshell> -2147483648 - 1
$10 ==> 2147483647

jshell>
```

25.28

```
jshell> 1234567890 == 1_234_567_890
$4 ==> true

jshell> System.out.println(1234567890)
1234567890

jshell> System.out.println(1_234_567_890)
1234567890

jshell> 1234567890 / 2
$5 ==> 617283945

jshell> 1_234_567_890 / 2
$6 ==> 617283945

jshell>
```

25.29

```
jshell> (17+23) == (17 + 23)
$7 ==> true

jshell>
```

25.30

```
jshell> int number1 = 10
number1 ==> 10

jshell> int number2 = 20
number2 ==> 20

jshell> System.out.printf("Sum is %d%n", (number1 + number2))
Sum is 30
$10 ==> java.io.PrintStream@1794d431

jshell> System.out.printf("Sum is %d%n", number1 + number2)
Sum is 30
$11 ==> java.io.PrintStream@1794d431

jshell>
```

25.31

```
jshell> int x = 14
x ==> 14

jshell> x = 27
x ==> 27

jshell>
```

25.32

```
jshell> int y = 29
y ==> 29

jshell> System.out.println(y)
29

jshell> y
y ==> 29
```

25.33

```
jshell> int b = 7
b ==> 7

jshell> int m = 9
m ==> 9

jshell> bm
|  Error:
|  cannot find symbol
|    symbol:   variable bm
|  bm
|  ^^

jshell> b * m
$3 ==> 63

jshell>
```

25.34 a) 2. b) 1.

```
jshell> 8 / 4
$4 ==> 2

jshell> 7 / 5
$5 ==> 1

jshell>
```

25.35

```
jshell> 0 / 1
$6 ==> 0

jshell> 1 / 0
|  java.lang.ArithmeticException thrown: / by zero
|        at (#7:1)

jshell> 0 / 0
|  java.lang.ArithmeticException thrown: / by zero
|        at (#8:1)

jshell>
```

25.36

```
jshell> (3 + 4 + 5) / 5
$9 ==> 2

jshell> 3 + 4 + 5 / 5
$10 ==> 8

jshell>
```

25.37

```
jshell> 5 / 2 * 2 + 4 % 3 + 9 - 3
$11 ==> 11

jshell>
```

25.38

```
jshell> 7 == 7
$12 ==> true

jshell> 7 != 7
$13 ==> false

jshell> 7 < 7
$14 ==> false

jshell> 7 <= 7
$15 ==> true

jshell> 7 > 7
$16 ==> false
```

(continued...)

```
jshell> 7 >= 7
$17 ==> true

jshell>
```

25.39

```
jshell> 7 == 9
$18 ==> false

jshell> 7 != 9
$19 ==> true

jshell> 7 < 9
$20 ==> true

jshell> 7 <= 9
$21 ==> true

jshell> 7 > 9
$22 ==> false

jshell> 7 >= 9
$23 ==> false

jshell>
```

25.40

```
jshell> 11 == 9
$24 ==> false

jshell> 11 != 9
$25 ==> true

jshell> 11 < 9
$26 ==> false

jshell> 11 <= 9
$27 ==> false

jshell> 11 > 9
$28 ==> true

jshell> 11 >= 9
$29 ==> true

jshell>
```

25.41

```
jshell> if (3 == 5); {
   ...>    System.out.println("3 is equal to 5");
   ...> }
3 is equal to 5

jshell>
```

25.42

```
jshell> int x = 1
x ==> 1

jshell> int y = 2
y ==> 2

jshell> int z = 3
z ==> 3

jshell> int a
a ==> 0

jshell> a = x = y = z = 10
a ==> 10

jshell> x
x ==> 10

jshell> y
y ==> 10

jshell> z
z ==> 10

jshell>
```

25.43

```
jshell> (3 * 9 * (3 + (9 * 3 / (3))))
$42 ==> 324

jshell>
```