Java Platform, Standard Edition

Java Shell User's Guide

Release 22
F87201-01
March 2024

Java Platform, Standard Edition Java Shell User's Guide, Release 22

F87201-01

Contents

Preface

This guide provides information about using the Java Shell tool (JShell) to explore the Java language and prototype code.

Audience

This document is intended for developers interested in using a Read-Eval-Print Loop (REPL) tool to learn the Java language, explore new Java APIs and features, and prototype complex code.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See JDK 22 Documentation.

Conventions

The following conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.
`<keys>`	Name of a keyboard key or key combination in angle brackets (<>) indicates that the key or key combination is pressed as part of the user input.

1

Introduction to JShell

The Java Shell tool (JShell) is an interactive tool for learning the Java programming language and prototyping Java code. It was introduced in JDK 9. JShell is a Read-Evaluate-Print Loop tool (REPL), which evaluates declarations, statements, and expressions as they are entered and immediately shows the results. The tool is run from the command line.

Topics

For reference information for this tool, see The jshell Command in *Java Development Kit Tool Specifications*.

Why Use JShell?

Using JShell, you can enter program elements one at a time, immediately see the result, and make adjustments as needed.

Java program development typically involves the following process:

- Write a complete program.
- Compile it and fix any errors.
- Run the program.
- Figure out what is wrong with it.
- Edit it.
- Repeat the process.

JShell helps you try out code and easily explore options as you develop your program. You can test individual statements, try out different variations of a method, and experiment with unfamiliar APIs within a JShell session. JShell doesn't replace an IDE. As you develop your program, paste code into JShell to try it out, and then paste working code from JShell into your program editor or IDE.

Starting and Stopping JShell

JShell was introduced in JDK 9. To start JShell, enter the `jshell` command on the command line.

JDK 9 or higher must be installed on your system. If your path doesn't include the bin directory, for example *`java-home`*`/jdk-16.0.1/bin`, then start the tool from within that directory.

The following example shows the command and the response from JShell. Text that you enter is shown in bold:

```
% jshell
|  Welcome to JShell -- Version 17.0.1
|  For an introduction type: /help intro

jshell>
```

The examples in this tutorial use the verbose mode. Verbose mode is recommended as you work through this tutorial so that what you see matches the examples. When you are more familiar with the tool, you might prefer to run in normal or a more concise mode.

To start JShell in verbose mode, use the `-v` option:

```
% jshell -v
```

To exit JShell, enter `/exit`:

```
jshell> /exit
|  Goodbye
```

2
Snippets

JShell accepts Java statements; variable, method, and class definitions; imports; and expressions. These pieces of Java code are referred to as snippets.

Topics

Trying Out Snippets

Snippets of Java code are entered into JShell and immediately evaluated. Feedback about the results, actions performed, and any errors that occurred are shown. Use the examples in this section to give JShell a try.

Start JShell with the verbose option to get the maximum amount of feedback available:

```
% jshell -v
|  Welcome to JShell -- Version 17.0.1
|  For an introduction type: /help intro
```

Enter the following sample statement at the prompt, and review the output that is shown:

```
jshell> int x = 45
x ==> 45
|  created variable x : int
```

First, the result is shown. Read this as: the variable x has the value 45. Because you are in verbose mode, a description of what occurred is also shown. Informative messages start with a vertical bar. Notice that both the name and the type of the created variable are shown.

> **Note:**
>
> Terminating semicolons are automatically added to the end of a complete snippet if not entered.

When an expression is entered that doesn't have a named variable, a scratch variable is created so that its value can be referenced later. The following example shows scratch values

for an expression and for the results of a method. The example also shows the continuation prompt (`...>`) that is used when a snippet requires more than one line of input to complete:

```
jshell> 2 + 2
$3 ==> 4
|  created scratch variable $3 : int

jshell> String twice(String s) {
   ...>     return s + s;
   ...> }
|  created method twice(String)

jshell> twice("Ocean")
$5 ==> "OceanOcean"
|  created scratch variable $5 : String
```

Changing Definitions

As you experiment with code, you might find that the definition of a variable, method, or class isn't doing what you want it to do. The definition is easily changed by entering a new one, which overwrites the previous definition.

To change the definition of a variable, method, or class, simply enter a new definition. For example, the `twice` method that was defined in Trying Out Snippets gets a new definition in the following example:

```
jshell> String twice(String s) {
   ...>     return "Twice:" + s;
   ...> }
|  modified method twice(String)
|  update overwrote method twice(String)

jshell> twice("thing")
$7 ==> "Twice:thing"
|  created scratch variable $7 : String
```

Notice that instead of showing `created method` as before, the feedback shows `modified method`. This message means that the definition changed, but the method has the same signature, and therefore all existing usages continue to be valid.

You can also change definitions in incompatible ways. The following example shows `x` being changed from `int` to `String`:

```
jshell> int x = 45
x ==> 45
|  created variable x : int

jshell> String x
x ==> null
|  replaced variable x : String
|  update overwrote variable x : int
```

The type of the variable `x` changed, and the feedback now shows `replaced`.

Changing the Level of Feedback

JShell was started in the verbose feedback mode, which provides a lot of commentary. You can set the amount and format of output with the `/set feedback` command, for example `/set feedback concise`. If you primarily use JShell by pasting from other windows, then you might prefer a feedback mode with no prompt and only error feedback. If so, then enter the `/set feedback silent` command.

Forward References

JShell accepts method definitions that reference methods, variables, or classes that aren't yet defined. This is done to support exploratory programming and because some forms of programming require it.

As an example, if you want to define a method for the volume of a sphere, then you can enter the following formula as the method `volume`:

```
jshell> double volume(double radius) {
   ...>     return 4.0 / 3.0 * PI * cube(radius);
   ...> }
|  created method volume(double), however, it cannot be invoked until
variable PI, and method cube(double) are declared
```

JShell allows the definition but warns of what is yet to be defined. The definition can be referenced, but if execution is attempted, then it fails until all of the required elements are defined:

```
jshell> double PI = 3.1415926535
PI ==> 3.1415926535
|  created variable PI : double

jshell> volume(2)
|  attempted to call method volume(double) which cannot be invoked until
method cube(double) is declared

jshell> double cube(double x) { return x * x * x; }
|  created method cube(double)
|    update modified method volume(double)

jshell> volume(2)
$5 ==> 33.510321637333334
|  created scratch variable $8 : double
```

With all of the definitions in place, the `volume` method now works.

This method is now used to illustrate more about incompatible replacement. For example, to change the precision of `PI`, enter the new value as shown in the following example:

```
jshell> BigDecimal PI = new BigDecimal("3.14159265358979323846264338 3")
PI ==> 3.14159265358979323846264338 3
|  replaced variable PI : BigDecimal
```

```
|    update modified method volume(double) which cannot be invoked
until this error is corrected:
|      bad operand types for binary operator '*'
|        first type:  double
|        second type: java.math.BigDecimal
|          return 4.0 / 3.0 * PI * cube(radius);
|                 ^------------^
|    update overwrote variable PI : double
```

The new definition of `PI` is type-incompatible with the definition of `volume()`. Because you are in verbose mode, update information is shown for other definitions affected by the change, which in this case describes the incompatibility. Notice that verbose mode is the only predefined feedback mode that displays update information. In other feedback modes, no warning is displayed until the code is executed. The purpose of this is to prevent an overload of updates. In all predefined modes, executing the `volume()` method displays the issue:

```
jshell> volume(2)
|  attempted to call method volume(double) which cannot be invoked
until this error is corrected:
|      bad operand types for binary operator '*'
|        first type:  double
|        second type: java.math.BigDecimal
|          return 4.0 / 3.0 * PI * cube(radius);
|                 ^------------^
|
```

Exceptions

In an exception backtrace, feedback identifies the snippet and the location within the snippet where the exception occurred.

The location within the code entered into JShell is displayed as `#`*`ID`*`:`*`line-number`*, where snippet *ID* is the number displayed by the `/list` command, and *line-number* is the line number within the snippet. In the following example, the exception occurs in snippet 1, which is the `divide()` method, on the second line of the method:

```
jshell> int divide(int x, int y) {
   ...> return x / y;
   ...> }
|  created method divide(int,int)

jshell> divide(5, 0)
|  java.lang.ArithmeticException thrown: / by zero
|        at divide (#1:2)
|        at (#2:1)

jshell> /list

   1 : int divide(int x, int y) {
           return x / y;
       }
   2 : divide(5, 0)
```

Tab Completion for Snippets

When you enter snippets, use the Tab key to automatically complete the item. If the item can't be determined from what was entered, then possible options are provided.

For example, if you entered the `volume` method from Forward References, then you can enter the first few letters of the method name, and then press the Tab key to complete the entry:

```
jshell> vol<Tab>
```

The input changes to the following:

```
jshell> volume(
```

If the item can be completed in more than one way, the set of possibilities is displayed:

```
jshell> System.c<Tab>
class                clearProperty(       console()
currentTimeMillis()

jshell> System.c
```

Any common characters are added to what you entered, and the cursor is placed at the end of the input so that more can be entered.

When you are at a method call's open parenthesis, pressing Tab shows completion possibilities with the parameter types:

```
jshell> "hello".startsWith(<Tab>
Signatures:
boolean String.startsWith(String prefix, int toffset)
boolean String.startsWith(String prefix)

<press tab again to see documentation>

jshell> "hello".startsWith(
```

Pressing Tab again shows a plain-text version of the documentation for the first method.

```
jshell> "hello".startsWith(<Tab>
boolean String.startsWith(String prefix, int toffset)
Tests if the substring of this string beginning at the specified index
starts with the
specified prefix.

Parameters:
prefix - the prefix.
toffset - where to begin looking in this string.

Returns:
true if the character sequence represented by the argument is a prefix of
```

```
the substring of this
          object starting at index toffset ; false otherwise. The
result is false if toffset is
          negative or greater than the length of this String object;
otherwise the result is
          the same as the result of the expression
                    this.substring(toffset).startsWith(prefix)


<press tab to see next documentation>

jshell> "hello".startsWith(
```

Snippet Transformation

JShell makes it easy to import a needed class when it is first referenced and convert an expression to a variable declaration using keyboard shortcuts.

When you enter an identifier that isn't imported yet, press Shift+Tab i (that is, hold down shift while pressing tab, then release and press i) immediately after the identifier to see the options that enable you to add the import to your session:

```
jshell> new JFrame<Shift+Tab i>
0: Do nothing
1: import: javax.swing.JFrame
Choice: 1
Imported: javax.swing.JFrame

jshell> new JFrame
```

Enter the number of the option you want. More than one import option could be provided.

You can convert an expression to a variable declaration by pressing Shift+Tab v after you enter the expression. The expression becomes the initial value of the variable declaration, and the type of the expression becomes the type of the variable. After you press Shift+Tab v, the cursor, which is indicated by a vertical bar (|) in the example, is placed in the line where you need to enter the variable name:

```
jshell> new JFrame("Demo") <Shift+Tab v>
jshell> JFrame | = new JFrame("Demo")
```

> **Note:**
>
> The expression must be valid or the transformation request is ignored. In the example, the import of `JFrame` is required before the variable transformation.

To complete the previous example, enter the variable name `frame` at the cursor and press Enter:

```
jshell> JFrame frame = new JFrame("Demo")
frame ==> javax.swing.JFrame[frame0,0,0,0x0,invalid,hidden, ...
tPaneCheckingEnabled=true]
|  created variable frame : JFrame

jshell>
```

Sometimes the result type of the expression isn't imported yet. In that case, Shift+Tab v offers to both import and create the variable, as shown in the following example:

```
jshell> frame.getGraphics() <Shift+Tab v>
0: Do nothing
1: Create variable
2: import: java.awt.Graphics. Create variable
Choice: 2
Imported: java.awt.Graphics

jshell> Graphics | = frame.getGraphics()
```

To complete the previous example, enter the variable name `gc` at the cursor and press Enter:

```
jshell> Graphics gc = frame.getGraphics()
|  created variable gc :java.awt.Graphics

jshell>
```

3
Commands

JShell commands are entered in a JShell session, and used to control the environment and display information.

Topics

Introduction to Commands

JShell commands control the environment and display information within a session.

Commands are distinguished from snippets by a slash (/). For information about the current variables, methods, and types, use the `/vars`, `/methods`, and `/types` commands. For a list of entered snippets, use the `/list` command. The following example shows these commands based on the examples in Trying Out Snippets:

```
jshell> /vars
|    int x = 45
|    int $3 = 4
|    String $5 = "OceanOcean"

jshell> /methods
|    twice (String)String

jshell> /list

   1 : System.out.println("Hi");
   2 : int x = 45;
   3 : 2 + 2
   4 : String twice(String s) {
         return s + s;
       }
   5 : twice("Ocean")
```

Notice that the types and values of variables and the type signature of methods are displayed.

JShell has a default startup script that is silently and automatically executed before JShell starts, so that you can get to work quickly. Entries from the startup script aren't listed unless you request them with the `/list -start` or `/list -all` command:

```
jshell> /list -all

  s1 : import java.io.*;
```

```
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;
   1 : System.out.println("Hi");
   2 : int x = 45;
   3 : 2 + 2
   4 : String twice(String s) {
         return s + s;
       }
   5 : twice("Ocean")
```

The default startup script consists of several common imports. You can personalize your startup entries with the `/set start` command. For information about this command, enter `/help /set start`. The `/save -start` command saves the current startup script as a starting point for your own startup script.

Other important commands include `/exit` to leave JShell, `/save` to save your snippets, and `/open` to enter snippets from a file.

Enter `/help` for a list of the JShell commands.

Tab Completion for Commands

Similar to snippet completion, when you enter commands and command options, use the Tab key to automatically complete the command or option. If the completion can't be determined from what was entered, then possible choices are provided.

The following example shows the feedback when Tab is pressed after the leading slash (/) for commands:

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

<press tab again to see synopsis>

jshell> /
```

Unique completions are done in-place. For example, after you enter `/l` and press Tab, the line is replaced with `/list`:

```
jshell> /l<Tab>

jshell> /list
```

Tab completion also works for command options. The following example shows the use of the Tab key to display the options for the `/list` command:

```
jshell> /list -<Tab>
-all       -history   -start

<press tab again to see synopsis>

jshell> /list -
```

Notice the message about pressing Tab again to show the command synopsis, which is a short description of the command. Press Tab a third time to show the help documentation. The following example shows the results of pressing Tab a second and third time:

```
jshell> /list -<Tab>
list the source you have typed

<press tab again to see full documentation>

jshell> /list -<Tab>
Show the snippets, prefaced with their snippet IDs.

/list
        List the currently active snippets of code that you typed or read
with /open

/list -start
        List the evaluated startup snippets

/list -all
        List all snippets including failed, overwritten, dropped, and startup

/list <name>
        List snippets with the specified name (preference for active
snippets)

/list <id>
        List the snippet with the specified snippet ID.
        One or more IDs or ID ranges may used, see '/help id'

jshell> /list -
```

Completion of unique arguments is done in place. For example, after you enter `/list -a<Tab>`, the `-all` option is automatically shown:

```
jshell> /list -a<Tab>

jshell> /list -all
```

Snippet names can also be completed with Tab. For example, if you defined the `volume` method earlier in the JShell session, then pressing Tab after you start to enter the method name results in the full method name being displayed:

```
jshell> /ed v<Tab>

jshell> /ed volume
```

Using Tab in a file argument position of the command shows the available files:

```
jshell> /open <Tab>
myfile1      myfile2    definitions.jsh

<press tab again to see synopsis>

jshell> /open
```

Completion of unique file names is done in place:

```
jshell> /open d<Tab>

jshell> /open definitions.jsh
```

Command Abbreviations

Reduce the amount of typing you have to do by using abbreviations. Commands, `/set` subcommands, command arguments, and command options can all be abbreviated, as long as the abbreviation is unique.

The only command that begins with `/l` is `/list`, and the only `/list` option that begins with `-a` is `-all` . Therefore, you can use the following abbreviations to enter the `/list -all` command :

```
jshell> /l -a
```

Also, the only command that begins with `/se` is `/set`, the only `/set` subcommand that begins with `fe` is `feedback`, and the only feedback mode that begins with `v` is `verbose`, assuming no custom feedback modes that start with `v` exist. Therefore, you can use the following abbreviations to set the feedback mode to verbose:

```
jshell> /se fe v
```

Notice that `/s` isn't a sufficient abbreviation because `/save` and `/set` both begin with the same letter. When in doubt, you can use Tab completion to see the options.

4
Editing

JShell supports editing input at the `jshell` prompt and editing in an external editor of your choice.

Shell editing enables you to edit snippets and commands as you enter them, and to retrieve and change previously entered snippets and commands. An external editor provides an alternative way to edit and create snippets, which is easier when you work with multiline snippets.

Topics

Shell Editing

Editing input at the command prompt makes it easy to correct your input and to retrieve and modify previously entered commands and snippets.

Shell editing in JShell is built on JLine2, which is functionally similar to BSD `editline` and GNU `readline` in Emacs mode. See JLine2 user information and GNU Readline documentation.

Topics

Input Line Navigation

Shell editing is supported for editing the current line, or accessing the history through previous sessions of JShell.

For navigating the input line, the Ctrl key and Meta key are used in key combinations. If your keyboard doesn't have a Meta key, then the Alt key is often mapped to provide Meta key functionality.

For basic navigation within a line, use the right and left arrow keys or Ctrl+B for backwards and Ctrl+F for forward. For navigation between lines in the history, use the up and down arrow keys. Pressing the up arrow once replaces the current line with the previous command or snippet line. Pressing the up arrow again brings you to the line previous to that. The history contains both commands and snippet lines. If a snippet has multiple lines, then the up and down arrows navigate through each line of a snippet.

The following table identifies the keys used and the actions taken to navigate the input line.

Keys	Action
Return	Enters the current line
Left arrow	Moves backward one character
Right arrow	Moves forward one character
Up arrow	Moves up one line, backward through history
Down arrow	Moves down one line, forward through history
Ctrl+A	Moves to the beginning of the line
Ctrl+E	Moves to the end of the line
Meta+B	Moves backward one word
Meta+F	Moves forward one word

History Navigation

A history of snippets and commands is maintained across JShell sessions. This history provides you with access to items that you entered in the current and previous sessions.

To reenter or edit prior input, navigate the history using the up, down, left, and right arrows. Entered text is inserted at the cursor. The Delete key is used to delete text. Press the Enter key to reenter the history line, modified or not.

The up and down arrow keys move backward and forward through the history one line at a time, for example:

```
jshell> class C {
   ...>    int x;
   ...> }
|  created
class

jshell> /list

   1 : class C
         int x;
       }

jshell> <up arrow>
```

The up arrow key shows the following line:

```
jshell> /list
```

Pressing the up arrow again shows the last line of the class definition:

```
jshell> }
```

Pressing the down arrow returns to the `/list` command. Pressing Enter executes it:

```
jshell> /list

   1 : class C {
         int x;
       }
```

Ctrl+up arrow goes up by snippets. For single-line snippets, Ctrl+up arrow behaves the same as up arrow. For multiline snippets, such as class `C`, Ctrl+up arrow skips the additional lines and goes to the top of the snippet.

Input Line Modification

Input lines retrieved from the history can be modified as needed and reentered, which saves you from having to retype a line just to make small changes.

Add text at the current cursor position simply by entering it. See Input Line Navigation for the keys used move the cursor within a line.

The following table identifies the keys used and the actions taken to modify the input line.

Keys	Action
Delete	Deletes the character at or after the cursor, depending on the operating system.
Backspace	Deletes the character before the cursor.
Ctrl+K	Deletes the text from the cursor to the end of the line.
Meta+D	Deletes the text from the cursor to the end of the word.
Ctrl+W	Deletes the text from the cursor to the previous white space.
Ctrl+Y	Pastes the most recently deleted text into the line.
Meta+Y	After Ctrl+Y, Meta+Y cycles through previously deleted text.

Search and More

Searching the history is a feature of JShell that makes it easier to find the line you want without going through the history one line at a time.

To start your search, press Ctrl+R. At the prompt, enter the search string. The search proceeds backward from your most-recent entry and includes previous sessions of JShell. The following example shows the prompt that is presented after pressing Ctrl+R:

```
jshell> <Ctrl+R>
((reverse-i-search)`':
```

Based on the example in History Navigation, entering `class` changes the display to show the most-recent line with the text `class`:

```
(reverse-i-search)`class': class C {
```

The search is incremental, so this line is retrieved with just the first character *c*. You can continue to search earlier in the history by pressing Ctrl+R repeatedly. Ctrl+S moves the search forward towards the present.

You can define a keyboard macro by entering `Ctrl+x (`, then entering your text, and finally entering `Ctrl+x )`. To use your macro, enter `Ctrl+x e`.

The following table shows the key combinations for searching and creating macros.

Keys	Action
Ctrl+R	Searches backward through history
Ctrl+S	Searches forwards through history
Ctrl+X (	Starts a macro definition
Ctrl+X)	Finishes a macro definition
Ctrl+X e	Executes a macro

External Editor

An alternative to editing at the command prompt is to use an external editor. This editor can be used to edit and create snippets, and is especially helpful for multiline snippets. You can configure JShell to use the editor of your choice.

To edit all existing snippets at once in an editor, use `/edit` without an option. To edit a specific snippet in an editor, use the `/edit` command with the snippet name or ID. Use the `/list` command to get the snippet IDs. The following example opens an editor to edit the snippet named `volume`, which was defined in Forward References:

```
jshell> /edit volume
```

You can also enter new snippets in the editor. When you save in the editor, any snippet that is changed or new is entered into the JShell session. Feedback from the snippets is shown in the JShell window, however, no JShell prompt is shown. You can't enter commands or snippets in the JShell window until the editor is closed.

If you don't specify an editor, then the following environment variables are checked in order: `JSHELLEDITOR`, `VISUAL`, and `EDITOR`. If none of those are set, then a simple default editor is used. To set up JShell to open the editor of your choice, use the `/set editor` command. The argument to the `/set editor` command is the command needed to start the external editor that you want to use. The following example sets `kwrite` as the editor and opens the editor with all existing snippets:

```
jshell> /set editor kwrite
|  Editor set to: kwrite

jshell> /edit
```

Defining `x` in the external editor window and saving the change generates the following output in the JShell window:

```
|  created variable x of type int with initial value 6
```

Closing the external editor restores the JShell prompt.

```
jshell>
```

5
External Code

External classes are accessed from a JShell session through the class path. External modules are accessed through the module path, additional modules setting, and module exports setting.

Topics

Setting the Class Path

You can use external code that is accessible through the class path in your JShell session.

Set the class path on the command line as shown in the following example:

```
% jshell --class-path myOwnClassPath
```

Point your class path to directories or JAR files that have the packages that you want to access. The code must be compiled into class files. Code in the default package, which is also known as the unnamed package, can't be accessed from JShell. After you set the class path, these packages can be imported into your session:

```
jshell> import my.cool.code.*
```

You can also use the `/env` command to set the class path, as shown in the following example:

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

The `/env` command resets the execution state, reloading any current snippets with the new class path setting or other environment setting entered with the command.

Setting Module Options

Modules are supported in JShell. The module path can be set, additional modules to resolve specified, and module exports given.

Module options can be provided in options to the `/env` command or on the command line as shown in the following example:

```
% jshell --module-path myOwnModulePath  --add-modules my.module
```

To see current environment settings, use `/env` without options. The following example includes class path information that was set in Setting the Class Path:

```
jshell> /env
|     --add-modules my.module
|     --module-path myOwnModulePath
|     --class-path myOwnClassPath
```

For details about the options, enter the following command:

```
jshell> /help context
```

6
Feedback Modes

The feedback mode determines the prompts, feedback, and other interactions within JShell. Predefined modes with different levels of feedback are provided. Custom modes can be created as needed.

Topics

Setting the Feedback Mode

A feedback mode is a named user interaction configuration. A feedback mode defines the prompts and feedback that are used in your interaction with JShell. Predefined modes are provided for your convenience. You can create custom modes as needed.

The predefined modes can't be modified, but they can be used as the base of a custom mode. The predefined modes, in descending order of verbosity are `verbose`, `normal`, `concise`, and `silent`.

The following table shows the differences in the predefined modes.

Mode	Value Snippets	Declaration	Updates	Commands	Prompt
verbose	name ==> value (and description)	Yes	Yes	Yes	jshell>
normal	name ==> value	Yes	No	Yes	jshell>
concise	name ==> value (only expressions)	No	No	No	jshell>
silent	No	No	No	No	->

- The Mode column indicates the mode that is being described.
- The Value Snippets column indicates what is shown for snippets that have values, such as expressions, assignments, and variable declarations.
- The Declaration column indicates if feedback is provided for declarations or methods, classes, enum, interfaces, and annotation interfaces.
- The Updates column indicates if changes to other than the current snippet are shown.
- The Commands column indicates if commands give feedback indicating success.
- The Prompt column indicates the prompt that is used.

The default feedback mode is `normal`. Change the feedback mode by setting a command-line option or using the `/set feedback` command as shown in the following example:

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

jshell> 2 + 2
$1 ==> 4
|  created scratch variable $1 : int

jshell> /set feedback silent
-> 2 + 2
-> /set feedback normal
|  Feedback mode: normal

jshell> 2 + 2
$3 ==> 4

jshell> /set feedback concise
jshell> 2 + 2
$4 ==> 4
jshell>
```

Notice that when the setting is `normal` or `verbose`, the command feedback shows you the setting, but `concise` and `silent` modes don't. Also notice that the three different forms of feedback for the expression `2+2` include no feedback when the mode is set to `silent`.

To see the current and available feedback modes, use the `/set feedback` command without options. Notice that the current mode is shown as the command that set it:

```
jshell> /set feedback
|  /set feedback verbose
|
|  Available feedback modes:
|     concise
|     normal
|     silent
|     verbose
```

Defining a Feedback Mode

Custom feedback modes enable you to define the prompts that you want to see and the feedback that you want to receive for the different elements that you enter into JShell.

A feedback mode has the following settings:

- Prompts: Regular and continuation
- Truncation: Maximum length of values displayed
- Format: Format of the feedback provided

The predefined modes can't be changed, but you can easily create a copy of an existing mode, as shown in the following example:

```
jshell> /set mode mine normal -command
|  Created new feedback mode: mine
```

The new mode `mine` is a copy of the `normal` mode. The `-command` option indicates that you want command feedback. If you don't want commands to describe the action that occurred, then use `-quiet` instead of `-command`.

Set Prompts

As with all `/set` commands, use the `/set prompt` command without settings to show the current setting:

```
jshell> /set prompt normal
|  /set prompt normal "\njshell> " "   ...> "
```

In the feedback provided for the previous example, the first string is the regular prompt, and the second string is the continuation prompt that is used if the snippet extends to multiple lines. The following example shows how to switch to the new mode to try it out:

```
jshell> /set prompt mine "\nmy mode: "  ".......: "

jshell> /set feedback mine
|  Feedback mode: mine

my mode: class C {
.......:     int x;
.......: }
|  created class C

my mode:
```

The prompt strings can contain `%s`, which is substituted with the next snippet ID. However, if a command is entered or the snippet results in an error, then the value that users enter at the prompt might not be assigned that ID.

All settings have a duration of the current session; they are not reset by the `/reset` command. If you want the settings to be the default for future sessions, then use the `-retain` option to keep them. The following example shows how to keep your custom mode across sessions:

```
my mode: /set mode mine -retain

my mode: /set feedback mine -retain
|  Feedback mode: mine

my mode: /exit
|  Goodbye
% jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro
```

```
my mode:
```

Set Truncation

If the values are too long, then they are truncated when displayed. Use the `/set truncation` command to set the maximum length shown for a value. If no settings are entered with the command, then the current setting is displayed. The following example shows the settings that were inherited from the `normal` mode:

```
my mode: /set truncation mine
|  /set truncation mine 80
|  /set truncation mine 1000 expression,varvalue

my mode: String big = IntStream.range(0,1200).mapToObj(n -> "" +
(char) ('a' + n % 26)).collect(Collectors.joining())
big ==> "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuv ...
fghijklmnopqrstuvwxyzabcd"
```

The conditions under which a truncation setting is in effect is determined by the optional selectors that are entered after the truncation length. Two types of selectors (called selector kinds in the online help) are defined:

- A case selector indicates the type of snippet whose value is displayed.
- An action selector describes what happened to the snippet.

Enter `/help /set truncation` for details about selectors.

The setting shown in the previous example means that values are truncated to 80 characters unless the value is the value of an expression (the `expression` case selector) or the value of a variable, as explicitly requested by entering just the variable name (the `varvalue` case selector). The order is important; the last one entered is used. If the order were reversed, then all of the values would be truncated to 80 characters.

The following example sets the default truncation to `100`, and only shows long values if they are explicitly requested:

```
my mode: /set truncation mine 100

my mode: /set truncation mine 300 varvalue

my mode: big + big
$2 ==>
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghi ...
yzabcdefghijklmnopqrstuvwxyzabcd"

my mode: big
big ==>
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqr
stuvwxyzabcdefghijklmnopqrstu
vwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmn
opqrstuvwxyzabcdefghijkl ... jklmnopq
rstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghij
klmnopqrstuvwxyzabcd"
```

```
my mode: /set mode mine -retain
```

To keep the new settings, retain the changes as shown at the end of the example.

Set Formats

Snippet output is another setting that you can customize. In the format inherited from the `normal` mode, an import doesn't give any feedback, and the type of a value isn't shown:

```
my mode: import java.beans.*

my mode: Locale.CANADA.getUnicodeLocaleAttributes()
$5 ==> []
```

The format of the snippet output is set with the `/set format` command. Enter this with the mode name and no settings to see the current format settings:

```
my mode: /set format mine
```

Extensive help about this command is available with the `/help /set format` command. You might want to bring it up to use as a reference for the remainder of this section, which mentions the fields that are used when defining the formats.

The primary feedback displayed is determined by the `display` field. Other fields can be defined to assist in defining the `display` field. Predefined modes other than `silent` define several of these fields, as can be seen in the output from the `/help /set format` command. These fields are inherited in the example mode. Display definitions for import are shown in the following example:

```
my mode: /set format mine display "{pre}added import {name}{post}" import-
added

my mode: /set format mine display "{pre}re-added import {name}{post}" import-
modified,replaced
```

The `name` field is predefined as the name of the snippet. The following example shows that feedback is now provided for imports:

```
my mode: import java.beans.*
|  re-added import java.beans.*
```

The `pre` and `post` fields used in the display definition are the prefix and postfix characters for each line of feedback output. The following example changes the vertical bar prefix to the empty string:

```
my mode: /set format mine pre ""

my mode: void m() {}
created method m()

my mode: import java.beans.*
```

```
re-added import java.beans.*

my mode: /set truncation mine
/set truncation mine 100
/set truncation mine 300 varvalue
```

> **Note:**
>
> The change to the prefix character affects all feedback, including command feedback.

To show the type when displaying values, change the `result` field defined by the predefined modes:

```
my mode: /set format mine result "{type} {name} = {value}{post}"
added,modified,replaced-primary-ok

my mode: Locale.CANADA.getUnicodeLocaleAttributes()
Set<String> $11 = []

my mode: 2 + 2
int $12 = 4
```

This change makes `result` nonempty only when it is new or updated (`added`,`modified`,`replaced`), is on the entered snippet (`primary`), and doesn't have errors (`ok`).

To permanently delete a retained mode, use the `-retain` option with the `-delete` option:

```
my mode: /set feedback verbose -retain
|  Feedback mode: verbose

jshell> /set mode mine -delete -retain
```

7

Scripts

A JShell script is a sequence of snippets and JShell commands in a file, one snippet or command per line.

Scripts can be a local file, or one of the following predefined scripts:

Script Name	Script Contents
DEFAULT	Includes commonly needed import declarations. This script is used if no other startup script is provided.
JAVASE	Imports the core Java SE API defined by the `java.se` module, which causes a noticeable delay in starting JShell due to the number of packages.
PRINTING	Defines JShell methods that redirect to the `print`, `println`, and `printf` methods in `PrintStream`.
TOOLING	Defines JShell methods accessing JDK tools like `javac`, `javadoc`, and `javap` using their command-line interface.

Topics

Startup Scripts

Startup scripts contain snippets and commands that are loaded when a JShell session is started. The default startup script contains common import statements. You can create custom scripts as needed.

Startup scripts are loaded each time the `jshell` tool is reset. Reset occurs during the initial startup and with the `/reset`, `/reload`, and `/env` commands. If you do not set the script, then, the default startup script, `DEFAULT`, is used. This default script defines commonly needed import declarations.

> **Note:**
>
> The Java language defines that the `java.lang` package is automatically imported so this package doesn't need to be explicitly imported.

To set the startup script, use the `/set start` command:

```
jshell> /set start mystartup.jsh
```

```
jshell> /reset
|  Resetting state.
```

As with all `/set` commands, the duration of the setting is the current session unless the `-retain` option is used. Typically, the `-retain` option isn't used when you test a startup script setting. When the desired setting is found, use the`-retain` option to preserve it:

```
jshell> /set start -retain
```

The startup script is then loaded the next time you start the `jshell` tool.

Remember that the startup scripts are loaded into the current session only when the state is reset. The contents of the script is stored, not a reference to the script. The script is read only at the time the `/set start` command is run. However, predefined scripts are loaded by reference and can be updated with new releases of the JDK.

Startup scripts can also be specified with the `--startup` command-line flag:

```
% jshell --startup mystartup.jsh
```

For experimentation, it is useful to have print methods that don't need the `System.out.` prefix. Use the predefined `PRINTING` script to access the `print`, `println`, and `printf` methods. You can specify more than one startup script with `/set start`. The following example sets the startup to load both the default imports and printing definitions:

```
jshell> /set start -retain DEFAULT PRINTING

jshell> /exit
|  Goodbye

% jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell> println("Hello World!")
Hello World!
```

The `-retain` flag is used to set these predefined scripts as the startup scripts for future sessions of the `jshell` tool. Use `/set start` without arguments to see the details of what is defined by these startup scripts.

To set more than one startup script on the command line, use the `--startup` flag for each script:

```
% jshell --startup DEFAULT --startup PRINTING
```

Creating and Loading Scripts

Use scripts to set up your JShell session with import statements and code that you want available during the session.

Creating Scripts

A script can be created externally in an editor, or generated from items entered in JShell. Use one of the following commands to create a script from the entries in a JShell session:

```
jshell> /save mysnippets.jsh

jshell> /save -history myhistory.jsh

jshell> /save -start mystartup.jsh
```

The first command shown in the example saves the current active snippets to `mysnippets.jsh`. The second command shown saves the history of all of the snippets and commands, both valid and invalid, to `myhistory.jsh`. The last command shown saves the contents of the current startup script setting to `mystartup.jsh`. The file name provided can be any valid file path and name.

Loading Scripts

Scripts can be loaded from the command line when a JShell session is started:

```
% jshell mysnippets.jsh
```

Scripts can also be loaded within a JShell session by using the `/open` command:

```
jshell> /open PRINTING
```

Accessing Command Line Tools Through JShell

The predefinded script `TOOLING` provides direct access to the JDK's command line tools, such as `javac`, `javadoc`, and `javap`, within the `jshell` tool. Load the `TOOLING` script when the `jshell` tool starts by running the following command:

```
jshell TOOLING
```

Alternatively, load it within within a JShell session with the following command:

```
jshell> /open TOOLING
```

Once you have loaded the `TOOLING` script, you can run observable tool services that have implemented the `java.util.spi.ToolProvider` interface by passing a name and an

array of arguments to the `run(String, String...)` method. Call `tools()` to print a sorted list of names for all runnable tools. For example:

```
jshell> /open TOOLING

jshell> tools()
jar
javac
javadoc
javap
jdeps
jlink
jmod
jpackage

jshell> run("javac","--version")
javac 22
```

For well known JDK tools, the `TOOLING` script defines convenience methods such as `javac(String... args) { run("javac", args); }`. This shortens the last call of the previous example to the following:

```
jshell> javac("--version")
javac 22
```

In addition, the `TOOLING` script defines a `javap` method that takes a `class` literal. With this method, you can disassemble and print an overview of an existing or newly created type without leaving the JShell session. For example:

```
jshell> interface Empty {}
|  created interface Empty

jshell> javap(Empty.class)
Classfile /C:/tmp/TOOLING-13600306095244067647.class
  Last modified Oct 4, 2023; size 191 bytes
  SHA-256 checksum
1e53e9d7d4549a00361937701d3b0a613b520a68854310796db7879efc08d195
  Compiled from "$JShell$22.java"
public interface REPL.$JShell$22$Empty
  minor version: 0
  major version: 65
  flags: (0x0601) ACC_PUBLIC, ACC_INTERFACE, ACC_ABSTRACT
  this_class: #1                          // REPL/$JShell$22$Empty
  super_class: #3                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 0, attributes: 3
Constant pool:
   #1 = Class              #2             // REPL/$JShell$22$Empty
   #2 = Utf8               REPL/$JShell$22$Empty
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               SourceFile
   #6 = Utf8               $JShell$22.java
```

```
   #7 = Utf8               NestHost
   #8 = Class              #9             // REPL/$JShell$22
   #9 = Utf8               REPL/$JShell$22
  #10 = Utf8               InnerClasses
  #11 = Utf8               Empty
{
}
SourceFile: "$JShell$22.java"
NestHost: class REPL/$JShell$22
InnerClasses:
  public static #11= #1 of #8;            // Empty=class
REPL/$JShell$22$Empty of class REPL/$JShell$22
```

The following JShell script creates a module named `com.greetings` that prints `Greetings!`. It creates a modular JAR that contains the module `com.greetings`, then prints its module declaration. Afterwards, the script creates a runtime image with the `jlink` tool that contains the module `com.greetings`.

This example is based on the example described in Project Jigsaw: Module System Quick-Start Guide.

```
/open PRINTING

print(
    """
    ------------------------------------------------
    Project Jigsaw: Module System Quick-Start Guide
    ------------------------------------------------
    """)

/* Create a module named com.greetings that prints "Greetings!". */

Files.createDirectories(Path.of("src/com.greetings/com/greetings"))

/* Create a module declaration named module-info.java */

Files.writeString(Path.of("src/com.greetings/module-info.java"),
    """
    module com.greetings {}
    """)

/* Create the main class */

Files.writeString(Path.of("src/com.greetings/com/greetings/Main.java"),
    """
    package com.greetings;
    public class Main {
        public static void main(String[] args) {
            System.out.println("Greetings!");
        }
    }
    """)

/open TOOLING
```

```
/* Compile the source code to the directory to the directory mods/
com.greetings */

javac("-d", "mods", "--module", "com.greetings", "--module-source-
path", "src")

/* Create a modular JAR that contains the module greetings.com. */

jar("--create", "--file=mlib/com.greetings.jar", "--main-
class=com.greetings.Main", "-C", "mods/com.greetings", ".")

/* Print the module declaration of the modular JAR com.greetings */

jar("--describe-module", "--file=mlib/com.greetings.jar")

/* Create a runtime image in the folder greetingsruntime that contains
the module com.greetings. /*

jlink("--module-path", "mlib", "--add-modules", "com.greetings", "--
output", "greetingsruntime", "--launcher", "greet=com.greetings")

/* You can run com.greetings.Main with this runtime as follows:
 *
 * greetingsruntime/bin/java --module com.greetings
 */

/exit
```