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()
insertAtFront(   isEmpty()          notify()
print()          removeFromBack()   removeFromFront()
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 `imports`

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`.