Combining multiple Scanners

(This extends the previous lecture that introduced Java's `Scanner` and highlighted how to use it with the standard input stream.)

Connecting a `Scanner` to other sources

As explained earlier, `Scanner` provides additional structure to input data: it interprets the input bytes and converts them to Java's native data types.

Also as explained earlier, it is possible to "connect" as `Scanner` to any existing `InputStream`, in particular Java's standard input stream named `System.in`.

In addition to this, `Scanner` can be connected to a character string, to provide additional structure to the characters. This is done by making the `Scanner` as follows:

```
Scanner <somename> = new Scanner( <somestring> );
```

For example:

```
String myString = "12 34 567";
Scanner myIn = new Scanner(myString);
```

When doing this, the scanner uses the characters from the string as input bytes for interpretation. In this specific example, using `nextInt()` on `myIn` will return twelve, thirty-four and five hundred and sixty-seven, in this order.

Input delimiters

During the data interpretation, a question arises: where to stop interpreting data? For example, how does `Scanner` know that the string of digits "`1234`" represents a single number, and not two numbers "`1`" then "`234`" or "`123`" then "`4`"?

By default, "Scanner" interprets data until the next white space character. In other words, when a program uses one of `Scanner`'s services, eg. `readInt()`, the `Scanner` reads multiple bytes from its underlying input stream until it encounters some white space, then interprets theses bytes to the desired type, then returns the converted value to the program.

For example, if a `Scanner` encounters the string "`12 34 56`" in the input stream, a program's first call to `nextInt()` will return twelve, the second call will return thirty-four, and the third call will return fifty-six.

We say that "the standard delimiter for `Scanner` is white space". The *delimiter* is the character (or group of characters) that `Scanner` detects as a separation between subsequent values returned to the program.

Changing the input delimiter

If the program must read data from a source using a custom format, often the format specifies a different delimiter than white space. For example, the standard spreadsheet format "CSV" uses a comma (",") as delimiter between subsequent values.

If we attempt to connect a `Scanner` to a CSV source and use `nextInt()` directly, chances are Java will complain as an error, that it does not know how to interpret a comma in the input. To interpret the comma as a delimiter, we must use another `Scanner` service, `useDelimiter()`:

Definition:

```
useDelimiter(String newDelimiter)
```

Semantics:

> Changes this `Scanner`'s delimiter to the specified string. The change takes effect directly for the next data item read from the input.

For example, the following code reads floating-point approximate numbers between 0.0 and 1.0, separated by commas, from the standard input stream, and prints each number in turn as a percentage between 0% and 100%:

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

myIn.useDelimiter(","); // <- this!

while (myIn.hasNext())
{
    double value = myIn.nextDouble();

    System.out.printf("%d%%\n", (int)(value * 100.));
}
```

A special delimiter: the "new line" character

There is a special character in the input that is *always interpreted as a delimiter* by `Scanner`: the new-line character, with ASCII code 10. So even if a `Scanner` has been configured with `useDelimiter()` to use a comma as delimiter, two values on separate lines will be interpreted separately without errors.

The addition of this special role in turn enables an important extra service of `Scanner`:

Definition:

```
String nextLine()
```

Semantics:

> Reads bytes until and including the first subsequent newline character, and returns the resulting bytes without the final newline character. Reports an error if end-of-stream was reached. Pauses the program if no bytes are yet available for reading and end-of-stream was not reached yet; the execution is resumed only when data becomes available or end-of-stream is reached.

This service is useful when an input format has a *variable number of columns* in each line. If a program is only interested in, say, the third column, it can read the first 3 columns of each line using `nextInt()` (or any other service of `Scanner`) then skip all remaining input on the same line until the beginning of the next line. For example:

```
Scanner myIn = new Scanner(System.in);
myIn.useDelimiter(",");

while (myIn.hasNext())
{
   myIn.next(); // read 1st column
   myIn.next(); // read 2nd column;
   String v = myIn.next(); // read 3rd column
   System.out.println(v);

   myIn.nextLine(); // skip until beginning of next line
}
```

This code reads comma-separated tabular data from the standard input and prints the contents of the 3rd column on each line; if there are more columns than 3, they are automatically skipped.

Combining multiple Scanners

A common application is the use of programs to process multi-dimensional data sets. An input data file containing a multi-dimensional data set usually has multiple levels of delimiters.

For example, a 3-D data set that contains a time sequence of 2-dimensional points could be specified to use the following format:

- the coordinates for one point are separated by commas; and
- different points are separated by semicolons.

For example, an input stream in this format could be "`4,2;5,2;1,1;0,-1`", to represent 4 points with coordinates (4,2), (5,2), (1,1) and (0,-1).

How to go about writing a program that reads multi-dimensional data? In this general case, you should think about *combining "Scanner" multiple times*. There is a general method for this:

- make a 1st `Scanner` for the outer dimension, configured with the outer dimension's delimiter, and read each element in the outer dimension in turn as a `String` using `next()`;
- for every `String` read in this way, make a 2nd `Scanner` for the 2nd outermost dimension, configured with that dimension's delimiter, and read each element in turn as a `String` using `next()`;
- and so on, for all outer dimensions;
- for every `String` read in this way, make a last `Scanner` for the inner dimension, then read the individual values using `nextInt()`, `nextDouble()` or the otherwise appropriate interpretation service from `Scanner`.

For example, using the 3-D format described above, there is only 1 outer dimension, so we start as follows:

```
Scanner in1 = new Scanner(System.in);

in1.useDelimiter(";"); // outer delimiter

while (in1.hasNext())
{
   String outerElement = in1.next();

   ...
}
```

Then we fill the missing part with the logic for the inner dimension:

```
Scanner in1 = new Scanner(System.in);

in1.useDelimiter(";");

while (in1.hasNext())
{
   String outerElement = in1.next();

   Scanner in2 = new Scanner(outerElement);

   in2.useDelimiter(","); // inner delimiter

   while (in2.hasNext())
   {
      int value = in2.nextInt();
      System.out.printf("Found a value: %d", value);
   }
}
```

Of course, as soon as there are two or more layers nested in this way, you should also introduce new functions that isolate each piece of logic. For example, the code above can be rewritten as follows:

```
// readPoints(): read 2D points separated by semicolons
void readPoints()
{
   Scanner in1 = new Scanner(System.in);

   in1.useDelimiter(";");

   while (in1.hasNext())
   {
      String outerElement = in1.next();

      readCoordinates(outerElement);
   }
}

// readCoordinates(): read scalars separated by commas
void readCoordinates(String element)
{
   Scanner in2 = new Scanner(element);

   in2.useDelimiter(",");

   while (in2.hasNext())
   {
      int value = in2.nextInt();
```

```
            System.out.printf("Found a value: %d", value);
        }
    }
```

Important concepts

- connecting `Scanner` to a `String`;
- *delimiters* and `useDelimiter();`
- new line characters and `nextLine();`
- combining multiple scanners for multi-dimensional input data.

Further reading

- Programming in Java, section 11.1.5 (pp. 541-542)
- Absolute Java, sections 2.2 (pp. 79-88)

Copyright and licensing