Introduction to Java's Scanner

Programs are not all about changing numbers into other numbers; a great part of automated process is *interacting* with the world around the computer. For example, a program may *input* data entered using a keyboard or mouse, and *output* diagrams visually to a screen.

The most common interface for input in imperative languages is the *sequential poll*: a program repeatedly performs a call over time to a system API to ask "is there some input available?", and the system responds either "no, no data available yet" or "yes, here is some data".

(There are other general interfaces to input data in programs than sequential polling. We will see another in a later course called "event loops"; others also exist.)

Sequential input stream

In languages with a sequential poll interface, an API exists to query the availability of input data "from the outside world". As data arrives, it is delivered to the program in the order of availability. We say that the program "reads" input data in sequential time order.

This ordering of input data over time is called *input streaming*, the input-side counterpart to output streaming seen in a previous lecture.

An *input stream* is an abstract concept, which models the arrival of fresh data into the program over time. An input stream is said to be *terminated* when there is a point time after which no data can be read any more. This point is called the *end-of-stream* (sometimes also called "end of file") event.

Data structuring in Java

At the level of hardware, input devices like a keyboard, mouse or network adapter delivers raw bits of data to the computer. Like for memory, it is the role of programming languages to *provide structure* to input data to ease the programmer's task.

In Java, two interfaces exist to give structure to bits of input data: `InputStream` to read input byte by byte, and `Scanner` to read input using more complex data types.

Raw bytes using Java's `InputStream`

Java's `InputStream` interface can be used in a program to read input byte per input byte. A byte is the numeric interpretation of 8 bits of raw data, interpreted as the binary representation of a value between 0 and 255.

The `InputStream` interface defines its main service `read` as follows:

Definition:

int read()

Semantics:

returns the next byte readable from the input stream as a value between 0 and 255. If the end of stream was reached, returns -1. This service will pause the program if it is called and no data is available yet; when this happens, execution only resumes when either at least one byte is available for reading or the end-of-stream has been reached.

Moreover, the *standard input stream* of a program, usually connected to the user's keyboard, is defined as an `InputStream` named `System.in`.

For example, the following code reads data from the standard input stream byte by byte, prints the numeric code of each byte on a separate line, and prints "goodbye" when the end-of-stream is reached:

```
int c;

do
{
   // Read one byte.
   c = System.in.read();

   if (c == -1) // end-of-file?
      System.out.println("goodbye");
   else
      System.out.println(c);

} while (c >= 0);
```

Structuring data using `Scanner`

Although bytes can represent all possible data input from the outside world, usually input data is structured in other ways: for example a number is usually encoded using its base-10 representation.

It is possible in Java to *add a layer of additional interpretation* to any `InputStream` using another interface called `Scanner`. To use this, the program must first "make" a `Scanner` using the following notation:

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

This notation will be explained in a subsequent lecture; for the time being, just use it as-is. You must replace "<somename>" by a name of your choosing and "<source>" by the name of an `InputStream` that can input bytes, for example `System.in`:

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

Once a `Scanner` is available, it offers to your program several services of interest, including:

Definition:

```
boolean hasNext()
```

Semantics:

> Returns "true" if data is available and the end-of-stream has not been reached yet. Returns "false" if the end-of-stream has been reached. Pauses the program if no data is available yet and the end-of-stream has not yet been reached, until either data becomes available or end-of-stream is encountered.

Definition:

```
int nextInt()
```

Semantics:

> Interprets the next bytes from the input as the representation of an integer in base 10 (for example "`123`" is interpreted to represents a hundred and twenty-three). Any leading minus sign ("-"), if present, is interpreted to denote a negative integer. Returns the integer value of the input representation. Reports an error if the end-of-stream is encountered. Pauses the program if no data is available yet and end-of-stream is not encountered; execution resumes only when some data is available or end-of-stream is encountered.

Definition:

```
double nextDouble()
```

Semantics:

> Same as `nextInt()`, except that the input bytes are interpreted as the scientific representation of a floating-point approximate number in base 10 (for example "314.15e-2" is interpreted to represent an approximation of 3.1415).

Definition:

```
String next()
```

Semantics:

> Return the next word as a string without interpretation. Reports an error if the end-of-stream is encountered. Pauses the program if no data is available yet and end-of-stream is not encountered; execution resumes only when some data is available or end-of-stream is encountered.

For example, the following program reads floating-point numbers from the standard input stream, until the end-of-stream is reached, and then prints their average if at least one value could be read:

```
Scanner myIn = Scanner(System.in);
int numberOfValues = 0;
double sum = 0;

while (myIn.hasNext())
```

```
{
    double value = myIn.nextDouble();
    sum = sum + value;
    numberOfValues = numberOfValues + 1;
}

if (numberOfValues > 0)
{
    double average = sum / (double)numberOfValues;
    System.out.printf("Average: %f\n", average);
}
```

For more comprehensive examples, refer to the files **Gemiddelde.java** and **NumberGame.java** in the accompanying source code repository.

Important concepts

- *sequential polling*
- *input stream*
- *standard input stream* in Java
- Difference between `InputStream` and `Scanner`
- `Scanner`'s `hasNext()`, `nextInt()` and `nextDouble()`

Further reading

- Programming in Java, sections 2.4.6 (pp. 46-47) and 11.1.5 (pp. 541-542)
- Absolute Java, section 2.2 (pp. 76-78)

Copyright and licensing