CSSE2002/7023

Programming in the large

Week 7.2: Intro to Java IO

In this hour:

- Encoding
- Streams
- Extracting characters and primitive types
- Output
- Serialisation
- Assignments follow up

Simple example

`ScanInts.java`

Now let's look at what that all means.

Starting I/O

We need some concepts first:

1. bytes/chars and char encodings
2. encoding data
3. streams

We'll start by talking about input (but output is mostly symmetrical).

Bytes / chars / encodings

Characters are represented by numbers at all times apart from when they are actually displayed. The mapping between symbols and numbers is referred to as an "encoding".

For a long time, the dominant encoding was ASCII[1] which used 7 bits (later 8).

There (usually) only being one default encoding in use — where a character fit into a byte $\Rightarrow$ chars and bytes seen as being equivalent (eg C has no separate byte type, it just uses char).

Unicode was developed to *try to* address the fact that there are more symbols in the world than will fit in a byte.

[1] American standard code for information interchange

Unicode

Java uses unicode, so:

- Chars are not bytes
- There is no single automatic way to translate between bytes and chars[2].

[2] UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE, …

Unicode

Code	Glyph	Decimal	Description
U+0020		32	Space
U+0021	!	33	Exclamation mark
U+0022	"	34	Quotation mark
U+0023	#	35	Number sign,
U+0024	$	36	Dollar sign
U+0025	%	37	Percent sign
U+0026	&	38	Ampersand
U+0027	'	39	Apostrophe
U+0028	(	40	Left parenthesis
U+0029	)	41	Right parenthesis
U+002A	*	42	Asterisk
U+002B	+	43	Plus sign
U+002C	,	44	Comma
U+002D	-	45	Hyphen-minus
U+002E	.	46	Full stop
U+002F	/	47	Slash (Solidus)
U+0030	0	48	Digit Zero
U+0031	1	49	Digit One
U+0032	2	50	Digit Two
U+0033	3	51	Digit Three
U+0034	4	52	Digit Four

Code	Glyph	Decimal	Description
U+0035	5	53	Digit Five
U+0036	6	54	Digit Six
U+0037	7	55	Digit Seven
U+0038	8	56	Digit Eight
U+0039	9	57	Digit Nine
U+003A	:	58	Colon
U+003B	;	59	Semicolon
U+003C	<	60	Less-than sign
U+003D	=	61	Equal sign
U+003E	>	62	Greater-than sign
U+003F	?	63	Question mark
U+0040	@	64	At sign
U+0041	A	65	Latin Capital letter A
U+0042	B	66	Latin Capital letter B
U+0043	C	67	Latin Capital letter C
U+0044	D	68	Latin Capital letter D
U+0045	E	69	Latin Capital letter E
U+0046	F	70	Latin Capital letter F
U+0047	G	71	Latin Capital letter G
U+0048	H	72	Latin Capital letter H
U+0049	I	73	Latin Capital letter I

Encoding data

If data/objects are to be sent somewhere else (to a file, across a network,...) they need to be encoded.

1. As binary data
   - Values expressed as bytes -
     eg: 12348 (base 10) $\Rightarrow$ 00 00 30 3C (4 bytes)
   - Often more compact than using text
   - Sensitive to system differences[3]
   - Not generally human readible
2. As text
   - Values written out in characters - $12348 \Rightarrow 12348$ (5 bytes)
   - Human "readible"
   - Parsing is more complicated[4]

[3] Eg endian-ness

[4] Need delimiters, escape sequences?

streams — abstraction

Useful abstractions for input:

- Externally:
  - Origin? Keyboard, disk files, network, ...
  - Chunking? Does it arrive one byte at a time (keyboard) or many (files)
- Internally - Once we've set up an input source (eg file vs keyboard), we don't want to code differently when using it.

An input stream is an abstract source of input that can be read without concern for how it is supplied[5].

Picture a pipe with buckets of water being poured in one end - there is no division in the water at the other end.

[5] I've avoided using "continuous" because if there is no input available, you still need to wait.

java.io.InputStream

Warning: Java spreads its IO functionality over a lot more classes than most other languages.

InputStream and its subclasses represent a streams of *bytes* (not chars). The subclasses draw their bytes from different sources:

- *FileInputStream* – get bytes from a file
- ByteArrayInputStream – get bytes from an array
- …

The idea is to have methods expect superclass parameters and then the streams can be substituted as needed.

End of file

Note that *all* input streams (and things which build on them) need to consider "End of file". eg:

```
public int read() // returns -1 on end of file
```

BufferedInputStream

Getting information from files can be slow a byte or char at a time.

A `BufferedInputStream` is an input stream which wraps around another input stream.

`ReadAll.java`

BufferedInputStream

Getting information from files can be slow a byte or char at a time.

A `BufferedInputStream` is an input stream which wraps around another input stream.

`ReadAll.java`

On a test VM reading a 4Meg file gives 32 milliseconds with buffering and 3402 milliseconds to read without.

java.io.Reader — get me chars please

`Reader` is the base class for things which get chars out of streams.

`Reader` itself is abstract but some useful subclasses are:

- BufferedReader
- InputStreamReader
- FileReader

Streams need closure

Streams and Readers have a `close()` method. Systems may have limits on the number of files you can have open at a time. When you have finished with a one, close it.
Be careful of the following:

```
try {
    r.readLine();     // throws
    r.close();        // may be skipped
} catch (Exception e) {
}
```

Some basic demos:

`Read1.java` Read from standard in.

`Read2.java` Read from a file.

Note:

- once we have constructed our `Reader`s we don't need to worry about their source.
- `close()` streams when you are finished with them
- closing a reader will close the stream as well.

These are to demonstrate the principle, there are better ways.

FileReader — a shortcut

`Read3.java`

`new FileReader(fname)` is roughly equivalent to `new InputStreamReader(new FileInputStream(fname))`.

BufferedReader

`BufferedReader` wraps another `Reader`. As well as buffering it adds: `String readLine();`

Extracting ints

`ReadInts.java`

Output

Dealing with output is “similar”.

- `java.io.OutputStream`s are for sending bytes.
  - `FileOutputStream`
  - `BufferedOutputStream`
  - …
- `Writer`s are for sending chars.
  - PrintWriter
  - BufferedWriter
  - …

“Similar”

`System.out` is an `OutputStream` but a specific subclass — `PrintStream`. (`PrintStream` has `print()` methods).
A better option for character output is `PrintWriter`.
You can construct a PrintWriter from `System.out` using:

```
new PrintWriter(System.out)
```

`Output.java`

`flush()`

If an `OutputStream/Writer` has buffering, output might not be sent immediately. `flush()` will send any pending output. This is important for:

- "interactive" or other situations where you expect a response (they won't respond if you haven't actually sent anything yet).
- Debugging or logging situations where you need an up to date view of what is happening.

`close()`ing a stream will flush it as well.

err

`System.err`

is a `PrintStream` like `System.out`. Generally used for error messages or information which doesn't belong in normal output (even though they often end up in the same place).

Serialization and ObjectStreams

`Object???Streams` allow I/O with java objects.

`Cereal.java`

The object being serialized must implement `Serializable`.

Any objects referenced will be written as well.

Assignments, pracs and tutes

- Marking Assignment 1 currently.
- A final report (with marks) will be distributed by email.
- Assignment 2 will be released later this week.
- Assignment 2 will be due on September 21 (in 3 weeks time).
- Pracs and tutes will run as normal this week.