CSSE2002/7023

Programming in the Large

Week 7.1: Intro to Java I/O

In this Session

- Encoding
- Streams
- Extracting characters and primitive types
- Output
- Serialisation

Simple Example

`ScanInts.java`

- `https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Scanner.html`
  - quite sophisticated – we've only used its simplist features
- Now let's look at what that all means

Introduction to I/O

We need some base concepts first:

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

We'll start by talking about input

- output is mostly symmetrical

Bytes / Chars / Encodings

Characters are represented by numbers

- apart from when they are actually displayed

"Encoding" is the mapping between symbols and numbers

ASCII[1] was the dominant encoding for a long time

- originally used 7 bits (later 8)
- many older languages, like C, took advantage of this
  - treating chars and bytes interchangeably

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

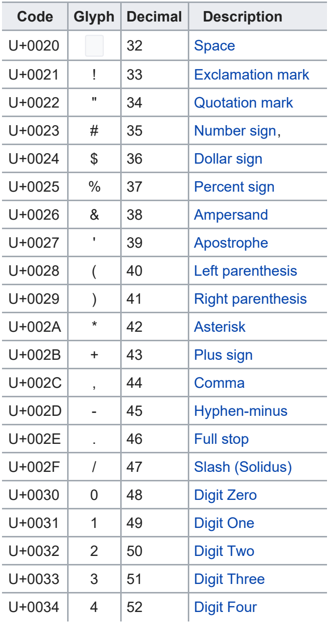
Unicode

Java uses Unicode

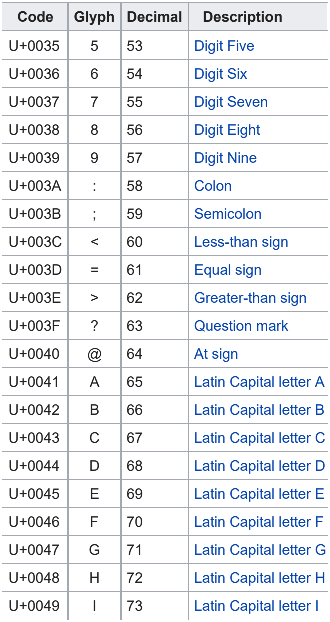
- Chars are not bytes
- No single automatic way to translate between bytes and chars[2].

[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. Binary
   - Values expressed as bytes –
     e.g. 12348 (base 10) ⇒ `00 00 30 3C` (4 bytes)
   - Often more compact than using text
   - Sensitive to system differences[3]
   - Not generally human readible
2. Text
   - Values written out in characters – 12348 ⇒ 12348 (5 bytes)
     - each character is ultimately encoded in binary in whatever medium it is stored or transported
   - Human "readible"
   - Parsing is more complicated[4]

[3] e.g. 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 (e.g. 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 I/O functionality over a *lot* more classes than most other languages

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

- `FileInputStream` — get bytes from a file
- `ByteArrayInputStream` — get bytes from an array
- ...

Methods using streams should expect superclass parameters

- Specific stream can be substituted as needed

End of File

All input streams (and things which build on them) need to consider “End of file”

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

BufferedInputStream

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

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

See: `ReadAll.java`

BufferedInputStream

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

BufferedInputStream is an input stream which wraps around another input stream

See: ReadAll.java

- On a test VM reading a 4MB file takes
  - 82 milliseconds with buffering
  - 18, 193 milliseconds without

`java.io.Reader` — get me chars please

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

`Reader` is abstract, 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 (IOException e) {
}
```

Streams Need Closure — What About `finally`?

```
try {
    r.readLine();    // throws
} catch (IOException e) {
    // ...
} finally {
    r.close();       // guaranteed to execute
}
```

- `close()` may also throw an `IOException`
    - not caught

Streams Need Closure — `try`-with-resources

```java
try (BufferedReader r =
        new BufferedReader(new FileReader(path)))
    r.readLine();    // throws
} catch (IOException e) {
    // ...
}
```

- Automatically closes resource
- Now, if either `readLine()` or `close()` throws an exception it will be caught

Examples

Some basic demos

- `Read1.java` — Read from standard in
- `Read2.java` — Read from a file

Note

- Once we have constructed our `Readers` we don't need to worry about their source
- `close()` streams when you are finished with them
- Closing a reader, closes the stream as well

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

`FileReader` — a shortcut

`Read3.java`

- `new FileReader(fname)`

is roughly equivalent to

- `new InputStreamReader(new FileInputStream(fname))`

- Also simplifies exception handling from `Read2.java`

BufferedReader

BufferedReader wraps another Reader

- As well as buffering it adds: `String readLine()`

Extracting ints

ReadInts.java

- Scanner reads ints from InputStream
- BufferedReader.readLine() reads Strings from stream
  - need to parse Strings to extract ints

Output

Dealing with output is “similar”

- `java.io.OutputStreams` are for sending bytes
  - `FileOutputStream`
  - `BufferedOutputStream`
  - `...`
- `Writers` are for sending chars
  - `PrintWriter`
  - `BufferedWriter`
  - `...`

“Similar”

`System.out` is an `OutputStream`

- actually, the subclass `PrintStream`
  - provides `print()` methods

Better option for character output is `PrintWriter`

You can construct a PrintWriter from `System.out` using:

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

See: `Output.java`

`flush()`

If an `OutputStream`/`Writer` is buffered, 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
```

- `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
    - can redirect output to different sources

Serialization and Object Streams

`ObjectInputStream` and `ObjectOutputStream` allow I/O with Java objects

See: `Cereal.java`

- Object being serialized *must* implement `Serializable`
- Any objects referenced will be written as well