Streams

I/O

From the program's perspective, all that matters about output is that it goes out, and all that matters about input is that it comes in.

So we should be able to work with input and output in a uniform matter regardless of whether it involves screen/keyboard, files, the network, …

Streams

Java's abstraction is the *stream*.

- *byte streams* read and write data as raw bytes
  - for machine-formatted data – the meaning comes from how the bytes are interpreted after being read
  - not human-readable – bytes must be interpreted correctly (application-specific)
  - efficient, in space and time
  - writing and reading must match – need the same interpretation
- *character streams* read and write data as characters
  - human-readable – bytes are interpreted using the standard character encoding
  - (what you want for most applications)

Streams

Answer	character streams	byte streams	Distractors
human-readable	✓ 12	0	0
bytes are interpreted using a standard encoding	✓ 5	6	1
Reader for input, Writer for output	✓ 11	1	0
typically preferred for text-based applications	✓ 9	1	2

all streams are ultimately byte streams but there is a standard encoding for characters

Streams

Answer	character streams	byte streams	Distractors
requires matching read/write interpretation to avoid corruption or data loss	7	✓ 4	1
efficient in space and time	1	✓ 10	1
for machine-formatted data	0	✓ 12	0
InputStream for input, OutputStream for output	1	✓ 11	0
requires interpretation for the correct meaning - have to know how to interpret the data correctly	5	✓ 5	2
not human-readable	0	✓ 12	0
the meaning comes from how the bytes are interpreted after being read	4	✓ 6	2

all streams are ultimately byte streams and so even character streams require a matching read/write interpretation – but there is a standard interpretation for "character stream"

Streams

Answer	character streams	byte streams	Distractors
requires additional buffering to function properly	5	2	✓ 5
used primarily for files	4	4	✓ 4
automatically converts numbers to text form	8	1	✓ 3
used primarily for network communication	1	8	✓ 3

buffering can improve performance for sources like files and URLs, but it isn't required or tied to byte vs character streams

`BufferedReader`/`Writer` are specific versions of buffered streams for character streams but the concept of buffering isn't limited to character streams

text files are common, but there are plenty of other kinds of files – including images

numbers are stored in text form with character streams, but the stream doesn't do the conversion – the number is written/read as characters and the program converts from/to numbers separately from the stream

underneath it is always a byte stream, but those can be wrapped in character streams so from the program's perspective network communication isn't limited to one or the other

Stream Classes

Stream classes come in two varieties –

- "hose" classes
  - think of a basic stream as a hose dispensing bytes or characters
  - attached to a source (for input) or a destination (for output)
- *wrapper* or "nozzle" classes
  - think of a wrapper class as a nozzle on the hose that converts the bytes/characters into something else as they leave the hose
  - constructors take an existing stream object as a parameter

Other classes –

- other classes may provide "nozzle" functionality without being part of the hierarchy of stream classes
  - e.g. `Scanner`

Working With Streams

Steps –

- acquire a hose connected to the desired input source or output destination
- (optionally) apply one or more nozzles to get useful functionality for reading/writing
- use the stream (read/write)
  - when reading from finite sources (such as a file), often need to be able to detect end-of-file (eof) / end-of-stream
- (in most cases) disconnect the hose (close the stream) when done
  - generally only disconnect the hose if you created the hose
    - don't close `System.in` or `System.out`, or a stream passed as a parameter

Working With Streams

- what hose you pick can depend on both the source/destination and byte vs character streams
  - know how your information is represented
- what nozzle(s) you pick depends on the underlying stream (byte vs character) and how you want to access it
  - e.g. for reading a text source, you can
    - use `Reader`'s `read` to get one character at a time
    - use `BufferedReader`'s `readLine` to get one line at a time
    - use `Scanner` to parse text
  - for writing text, typically use `PrintWriter`
  - can apply multiple nozzles
    - e.g. `BufferedReader` wraps a `Reader`, but `System.in` is an `InputStream` – first need to wrap `System.in` with an `InputStreamReader`

```
BufferedReader reader =
    new BufferedReader(new InputStreamReader(System.in));
```


A Brief Overview of Some Stream Classes

Question	for converting byte-based input into character-based input	for reading binary data from a file as raw bytes	for reading lines of text, as well as improving performance	for reading text data from a file	for writing formatted text output	for writing primitive types in a machine-formatted form	for converting character-based input into byte-based input	for keyboard input only, not file input
InputStreamReader	✓ 7	2	0	1	0	0	0	1
FileInputStream	0	✓ 10	0	0	0	1	0	0
BufferedReader	0	0	✓ 9	1	0	0	0	2
FileReader	0	0	0	✓ 11	0	0	0	0
PrintWriter	0	0	0	1	✓ 9	0	0	1
DataOutputStream	0	0	0	0	2	✓ 6	1	0

`...Reader` means things are ultimately read as characters

`BufferedReader` is suitable and commonly used for reading from files, but it is not limited to files

`...Writer` means writing (output), not input

`...OutputStream` means writing (output), not input

you can write text with a `DataOutputStream`, but if that is all you want to do, `PrintWriter` is a better choice

Detecting End-of-Stream (and Other Reading-Related Issues)

When reading from a finite source (such as a file), it is common to want to read until the end of the source.

In general, there are two strategies when you want to do something that may or may not succeed –

- look before you leap
  - check for the conditions that mean success and only take the action if those conditions are satisfied
- leap before you look
  - take the action, and handle any problems that occur

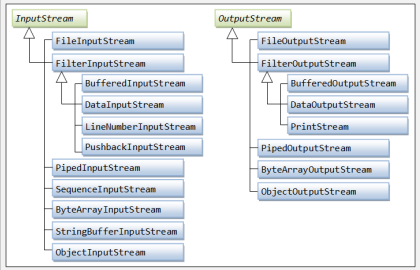
But looking first may not always be possible.

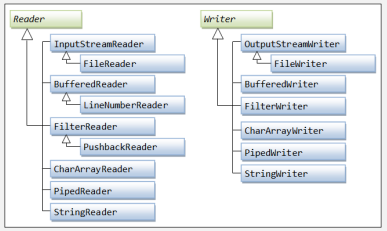
- e.g. can't know what's next without reading it
- e.g. can't know if there's any more to read without reading it

So...for reading input, leap before you look – and look at the full description of `read` methods in the API to find out how to determine if end-of-stream was reached.

Stream Type Hierarchy

- byte streams
  - by convention, `InputStream` and `OutputStream` denote byte streams
- character streams
  - by convention, `Reader` and `Writer` denote character streams





Writing Reusable Code

- declare variables and parameters using the most general type possible
  - prefer nozzle types to hose types

Acquiring Hoses

Existing hose objects already connected –

- standard input – `System.in`
  - type `InputStream`
- standard output – `System.out`
  - type `PrintStream`
- standard error – `System.err`
  - type `PrintStream`

Are these byte streams or character streams?
– byte streams

Acquiring Hoses

Create a new hose connected to a file –

- `FileInputStream`, `FileOutputStream` – for machine-readable (byte-oriented) files

`FileInputStream`(`File file`) Creates a `FileInputStream` by opening a connection to an actual file, the file named by the `File` object `file` in the file system.
`FileInputStream`(`String name`) Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name `name` in the file system.

- `FileReader`, `FileWriter` – for human-readable (character-oriented) files

`FileReader`(`File file`) Creates a new `FileReader`, given the `File` to read from.
`FileReader`(`String fileName`) Creates a new `FileReader`, given the name of the file to read from.

Acquiring Hoses

Create a new hose connected to a URL –

- create a URL and get the stream from it
  - byte stream

`URL(String spec)`	Creates a `URL` object from the `String` representation.
`URL(String protocol, String host, int port, String file)`	Creates a `URL` object from the specified `protocol`, `host`, `port` number, and `file`.

`InputStream` **`openStream`**()	Opens a connection to this `URL` and returns an `InputStream` for reading from that connection.

Acquiring Hoses

Create a new hose connected to a string –

- `StringReader`
  - character stream

`StringReader(String s)`
Creates a new string reader.

- `StringWriter`
  - character stream

`StringWriter()`
Create a new string writer using the default initial string-buffer size.

StringBuffer	**getBuffer**() Return the string buffer itself.
String	**toString**() Return the buffer's current value as a string.

Top-Level Stream Classes

- byte streams
  - `InputStream`

both also provide methods to work with an array of bytes

```
public abstract int read()
                throws IOException
```

Reads the next byte of data from the input stream. The value byte is returned as an `int` in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value `-1` is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

A subclass must provide an implementation of this method.

Returns:
the next byte of data, or `-1` if the end of the stream is reached.
Throws:
IOException - if an I/O error occurs.

  - `OutputStream`

```
public abstract void write(int b)
                 throws IOException
```

Writes the specified byte to this output stream. The general contract for `write` is that one byte is written to the output stream. The byte to be written is the eight low-order bits of the argument `b`. The 24 high-order bits of `b` are ignored.

Subclasses of `OutputStream` must provide an implementation for this method.

Parameters:
`b` - the byte.
Throws:
IOException - if an I/O error occurs. In particular, an `IOException` may be thrown if the output stream has been closed.

Top-Level Stream Classes

- character streams
  - `Reader`

both also provide methods to work with an array of chars

```
public int read()
        throws IOException
```

Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.

Subclasses that intend to support efficient single-character input should override this method.

Returns:
The character read, as an integer in the range 0 to 65535 (`0x00-0xffff`), or -1 if the end of the stream has been reached
Throws:
IOException - If an I/O error occurs

  - `Writer`

```
public void write(int c)
         throws IOException
```

Writes a single character. The character to be written is contained in the 16 low-order bits of the given integer value; the 16 high-order bits are ignored.

Subclasses that intend to support efficient single-character output should override this method.

Parameters:
`c` - int specifying a character to be written
Throws:
IOException - If an I/O error occurs

```
public void write(String str)
         throws IOException
```

Writes a string.

Parameters:
`str` - String to be written
Throws:
IOException - If an I/O error occurs

Wrapper/Nozzle Classes – Character Streams

`PrintWriter(OutputStream out)`	Creates a new PrintWriter, without automatic line flushing, from an existing OutputStream.
`PrintWriter(Writer out)`	Creates a new PrintWriter, without automatic line flushing.

void	**println**() Terminates the current line by writing the line separator string.
void	**println**(boolean x) Prints a boolean value and then terminates the line.
void	**println**(char x) Prints a character and then terminates the line.
void	**println**(char[] x) Prints an array of characters and then terminates the line.
void	**println**(double x) Prints a double-precision floating-point number and t
void	**println**(float x) Prints a floating-point number and then terminates th
void	**println**(int x) Prints an integer and then terminates the line.
void	**println**(long x) Prints a long integer and then terminates the line.
void	**println**(Object x) Prints an Object and then terminates the line.
void	**println**(String x) Prints a String and then terminates the line.

void	**print**(boolean b) Prints a boolean value.
void	**print**(char c) Prints a character.
void	**print**(char[] s) Prints an array of characters.
void	**print**(double d) Prints a double-precision floating-point number.
void	**print**(float f) Prints a floating-point number.
void	**print**(int i) Prints an integer.
void	**print**(long l) Prints a long integer.
void	**print**(Object obj) Prints an object.
void	**print**(String s) Prints a string.

- `PrintWriter`

note: unlike many streams, `PrintWriter` does not throw exceptions – full robustness requires calling `checkError()` to determine if an error occurred

Wrapper/Nozzle Classes – Character Streams

- `BufferedReader`

`BufferedReader(Reader in)`	Creates a buffering character-input stream that uses a default-sized input buffer.

`String`	`readLine()` Reads a line of text.

Utility Classes

- `Scanner` acts like a nozzle in terms of reading from a stream, but isn't part of the streams type hierarchy
  - you can't add another nozzle to a `Scanner`

`Scanner(File source)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(InputStream source)`	Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner(Readable source)` ◄ `Readable` covers the `Reader` classes	Constructs a new `Scanner` that produces values scanned from the specified source.
`Scanner(String source)`	Constructs a new `Scanner` that produces values scanned from the specified string.

  - it provides more sophisticated text parsing than the wrapper classes

`String`	`next()`	Finds and returns the next complete token from this scanner.
`String`	`next(String pattern)`	Returns the next token if it matches the pattern constructed from the specified string.
`String`	`next(Pattern pattern)`	Returns the next token if it matches the specified pattern.
`BigDecimal`	`nextBigDecimal()`	Scans the next token of the input as a `BigDecimal`.
`BigInteger`	`nextBigInteger()`	Scans the next token of the input as a `BigInteger`.
`BigInteger`	`nextBigInteger(int radix)`	Scans the next token of the input as a `BigInteger`.
`boolean`	`nextBoolean()`	Scans the next token of the input into a boolean value and returns that value.
`byte`	`nextByte()`	Scans the next token of the input as a `byte`.

Wrapper/Nozzle Classes – Byte Streams

- `InputStreamReader/OutputStreamWriter`
  - read/write characters to byte streams

`InputStreamReader(InputStream in)`	Creates an InputStreamReader that uses the default charset.
`OutputStreamWriter(OutputStream out)`	Creates an OutputStreamWriter that uses the default character encoding.

Wrapper/Nozzle Classes – Byte Streams

- `DataOutputStream`
  - write values of various types (uses binary representation)

`DataOutputStream(OutputStream out)`	Creates a new data output stream to write data to the specified underlying output stream.

`void`	`writeBoolean(boolean v)` Writes a `boolean` to the underlying output stream as a 1-byte value.
`void`	`writeByte(int v)` Writes out a `byte` to the underlying output stream as a 1-byte value.
`void`	`writeBytes(String s)` Writes out the string to the underlying output stream as a sequence of bytes.
`void`	`writeChar(int v)` Writes a `char` to the underlying output stream as a 2-byte value, high byte first.
`void`	`writeChars(String s)` Writes a string to the underlying output stream as a sequence of characters.
`void`	`writeDouble(double v)` Converts the double argument to a `long` using the `doubleToLongBits` method in class `Double`, and then writes that `long` value to the underlying output stream as an 8-byte quantity, high byte first.
`void`	`writeFloat(float v)` Converts the float argument to an `int` using the `floatToIntBits` method in class `Float`, and then writes that `int` value to the underlying output stream as a 4-byte quantity, high byte first.
`void`	`writeInt(int v)` Writes an `int` to the underlying output stream as four bytes, high byte first.
`void`	`writeLong(long v)` Writes a `long` to the underlying output stream as eight bytes, high byte first.
`void`	`writeShort(int v)` Writes a `short` to the underlying output stream as two bytes, high byte first.

Wrapper/Nozzle Classes – Byte Streams

- `DataInputStream`
  - read values of various types (uses binary representation)

`DataInputStream(InputStream in)`	Creates a DataInputStream that uses the specified underlying InputStream.

boolean	`readBoolean()` See the general contract of the `readBoolean` method of `DataInput`.
byte	`readByte()` See the general contract of the `readByte` method of `DataInput`.
char	`readChar()` See the general contract of the `readChar` method of `DataInput`.
double	`readDouble()` See the general contract of the `readDouble` method of `DataInput`.
float	`readFloat()` See the general contract of the `readFloat` method of `DataInput`.
int	`readInt()` See the general contract of the `readInt` method of `DataInput`.
long	`readLong()` See the general contract of the `readLong` method of `DataInput`.
short	`readShort()` See the general contract of the `readShort` method of `DataInput`.

Serialized Object I/O

`DataInputStream`/`DataOutputStream` provide methods for reading/writing primitive types.

`ObjectInputStream`/`ObjectOutputStream` provide `readObject()`, `writeObject(obj)` to read/write objects.

Notes.

- the object class must implement `Serializable`
- the binary format of objects is specific to Java – can't use `ObjectOutputStream` to write data to be read by something other than another Java program
- the binary format of objects is subject to change with different versions of Java – don't use it for long-term storage
- only one copy of an object is written even if there are multiple references to it
  - only serialize immutable objects (e.g. `String`), and/or
  - call `reset()` on the stream when needed