Programming Languages and Techniques (CIS120)

Lecture 32

April 14, 2014

java.io
Exceptions

java.io

Java I/O Design Strategy

1. Understand the concepts and how they relate:
   - What kind of stream data are you working with?
   - Is it byte-oriented or text-oriented?
     - InputStream vs. InputReader
   - What is the source of the data?
     - e.g. file, console, network, internal buffer or array
   - Does the data have any particular format?
     - e.g. comma-separated values, line-oriented, numeric
     - Consider using Scanner or another parser

2. Design the interface:
   - Browse through java.io libraries (to remind yourself what's there!)
   - Determine how to compose the functionality your need from the library
   - Some data formats require more complex *parsing* to convert the data stream into a useable structure in memory

I/O Streams

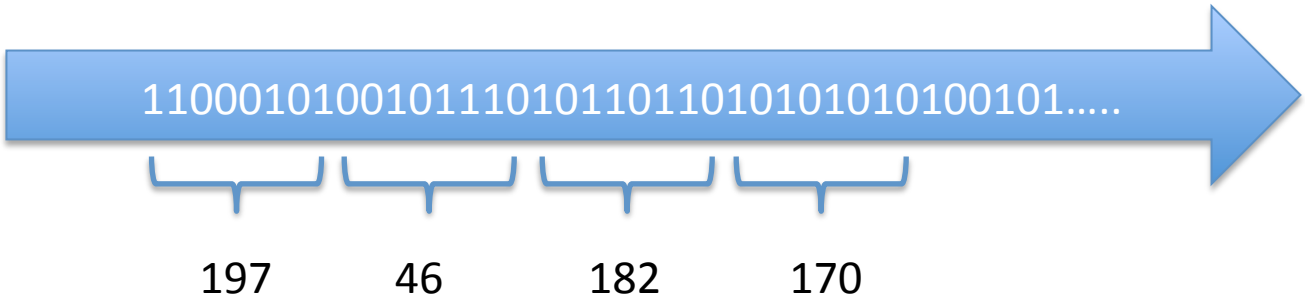
- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.

Different kinds of IO

- Character-oriented
  - For working with text (i.e. .txt files, webpages)
  - Reads/writes data in 16-bit chunks
  - Uses “character encoding” to interpret that data (multiple character sets possible, all agree for Latin characters)
  - Examples: subclasses of Reader and Writer
- Byte-oriented (aka “Binary” input)
  - Simplest form of input/output
  - Reads/writes data in 8-bit chunks
  - Interpretation of that data is up to your program
  - Gotcha: can also interpret 8-bit chunks as characters
- Special purpose file formats
  - Built on top of byte- or character- based formats
  - Examples: CSV, XML, JPG, MP3
  - *Parsing* is the process of converting the data stream into a useable structure in memory

Byte-oriented Streams

- At the lowest level, a stream is a sequence of binary numbers



- The simplest IO classes break up the sequence into 8-bit chunks, called *bytes*. Each byte corresponds to an integer in the range 0 – 255.

`InputStream` and `OutputStream`

- Abstract classes* that provide basic operations for the Stream class hierarchy:

```
abstract int read ();          // Reads the next byte of data
abstract void write (int b); // Writes the byte b to the output
```

- These operations read and write `int` values that represent *bytes*
  - range `0-255` represents a byte value
  - `-1` represents "no more data" (when returned from read)
- java.io provides many subclasses for various sources/sinks of data:
  - files, audio devices, strings, byte arrays, serialized objects
- Subclasses also provides rich functionality:
  - encoding, buffering, formatting, filtering

*Abstract classes are classes that cannot be directly instantiated (via `new`). Instead, they provide partial, concrete implementations of some operations. In this way, abstract classes are a bit like interfaces (they provide a partial specification) but also a bit like classes (they provide some implementation).

Demo

Binary input demo: Image.java

Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

Binary IO example

```
public Image() throws IOException {
   InputStream fin = new FileInputStream(“mandrill.pgm”);

   data = new int[width][height];
   for (int i=0; i < width; i++) {
      for (int j=0; j < height; j++) {
          int ch = fin.read();
          if (ch == -1) {
              fin.close();
              throw new IOException("File ended too early");
          }
          data[j][i] = ch;
      }
   }
   fin.close();
}
```

BufferedInputStream

- Reading one byte at a time can be slow!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.

  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program
  disk -> operating system -> JVM -> program

- A `BufferedInputstream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

  disk -> operating system ->>>> JVM -> program
  JVM -> program
  JVM -> program
  JVM -> program

Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

Buffering example

```
public Image() throws IOException {
    FileInputStream fin1 = new FileInputStream("mandrill.pgm");
    InputStream fin = new BufferedInputStream(fin1);

    data = new int[width][height];
    for (int i=0; i < width; i++) {
        for (int j=0; j < height; j++) {
            int ch = fin.read();
            if (ch == -1) {
                throw new IOException("File ended too early");
            }
            data[j][i] = ch;
        }
    }
    fin.close();
}
```

`PrintStream` Methods

`PrintStream` adds buffering and binary-conversion methods to `OutputStream`

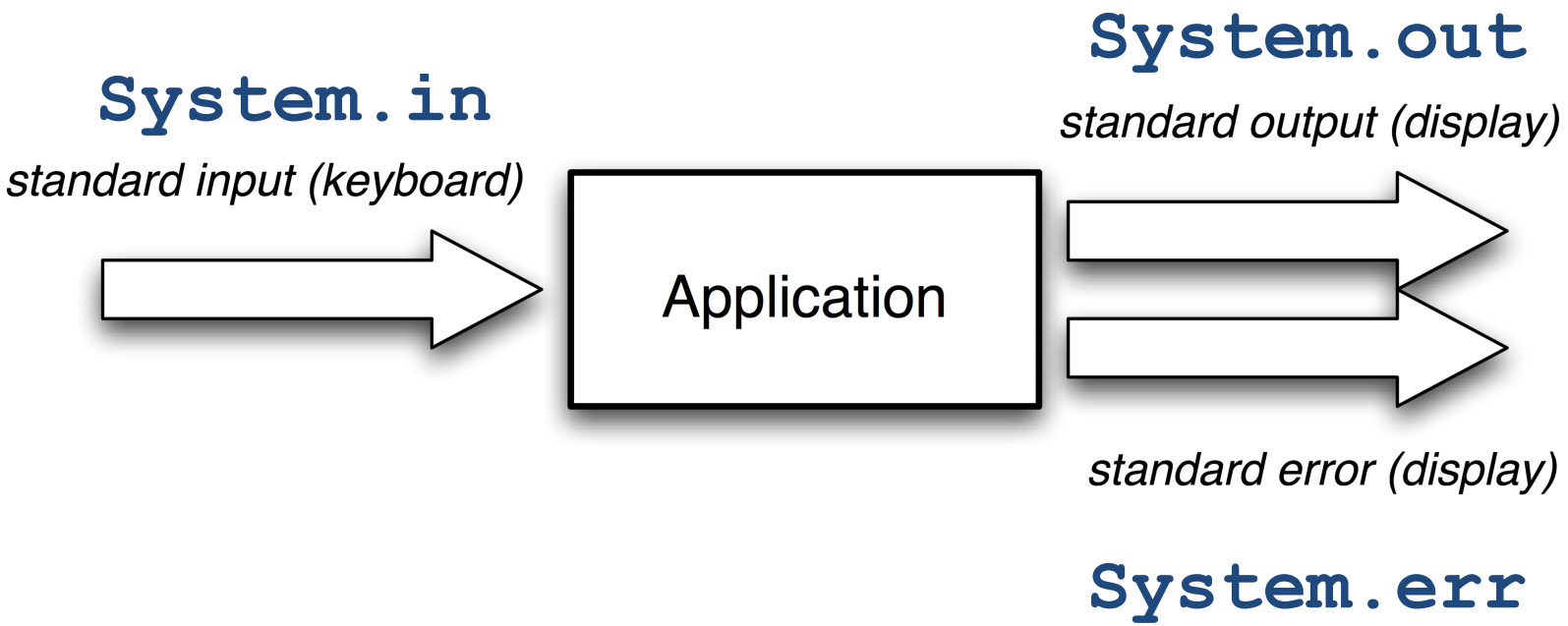
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

void print(String s);    // write s without terminating the line
                         //  (output may not appear until the stream is flushed)
void flush();            // actually output characters waiting to be sent
```

- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method's call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to "glue together" the right stream processing routines

The Standard Java Streams

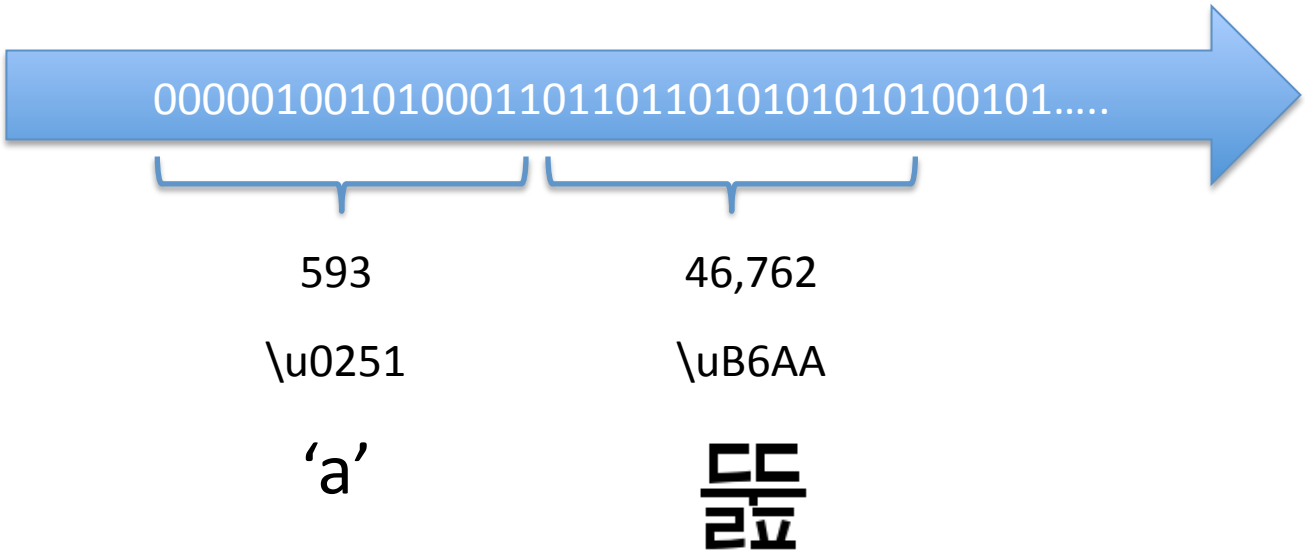
java.lang.System provides an `InputStream` and two standard `PrintStream` objects for doing console I/O.



Note that `System.in`, for example, is a *static member* of the class System – this means that the field "`in`" is associated with the *class*, not an *instance* of the class. Recall that static members in Java act like global variables. Methods can also be static – the most common being "`main`", but see also the `Math` class.

Character based IO

A character stream is a sequence of 16-bit binary numbers



The character-based IO classes break up the sequence into 16-bit chunks, of type `char`. Each character corresponds to a letter (specified by a *character encoding*).

Reader and Writer

- Similar to the `InputStream` and `OutputStream` classes, including:

```
abstract int read ();          // Reads the next character
abstract void write (int b); // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - read returns an integer in the range 0 to 65535 (i.e. 16 bits)
  - value `-1` represents "no more data" (when returned from read)
  - requires an "encoding" (e.g. UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, the library provides many subclasses of Reader and Writer Subclasses also provides rich functionality.
  - use these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - Wrap in an InputStreamReader / PrintWriter if you need unicode console I/O

Exceptions

Piazza question

```
public FileCorrector(String file) throws IOException {
   BufferedReader br =
       new BufferedReader(new FileReader(file));
   String line;
   while ((line = br.readLine()) != null) {
       // Secret stuff in fancy while loop
   }
   br.close();
}
```

Will br be closed at the end of the constructor?

1. yes
2. maybe
3. no

Finally

- A “finally” clause of a try/catch/finally statement *always* gets run, regardless of whether there is no exception, a propagated exception, or a caught exception — or even if the method returns from inside the try.
- “Finally” is often used for releasing resources that might have been held/created by the “try” block:

```
public void doSomeIO (String file) {
  FileReader r = null;
  try {
    r = new FileReader(file);
    … // do some IO
  } catch (FileNotFoundException e) {
    … // handle the absent file
  } catch (IOException e) {
    … // handle other IO problems
  } finally {
    if (r != null) {    // don’t forget null check!
      try { r.close(); } catch (IOException e) {…}
    }
  }
}
```

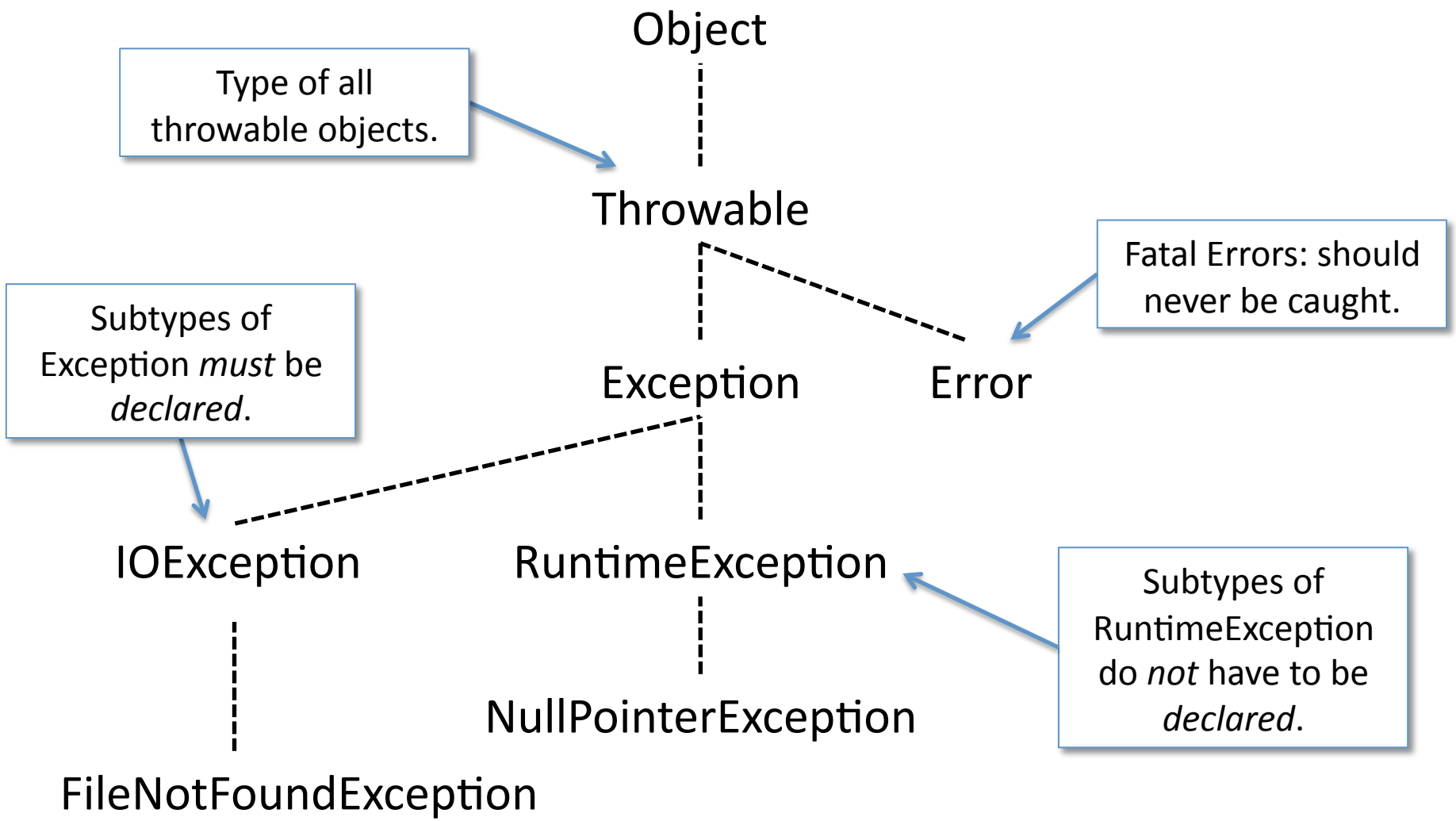
When are throws clauses necessary?

```
public method(String file) throws IOException {
   // do some stuff with IO
}
```

What classes of exceptions must be declared in throws clauses?

1. Only IOExceptions
2. All exceptions defined in libraries
3. All exceptions, except those that are subclasses of RuntimeException
4. All exceptions (*if you want full style points)*

Exception Class Hierarchy



Checked (Declared) Exceptions

- Exceptions that are subtypes of `Exception` but not `RuntimeException` are called *checked* or *declared*.
- A method that might throw a checked exception must declare it using a "throws" clause in the method type.

```
public void maybeDoIt (String file) throws AnException {
    if (…) throw new AnException();  // directly throw
    …
```

- Even if it doesn't throw the exception directly

```
public void doSomeIO (String file) throws IOException {
    Reader r = new FileReader(file);  // might throw
    …
```

Unchecked (Undeclared) Exceptions

- Subclasses of RuntimeException *do not* need to be declared via "throws"
  - even if the method does not explicitly handle them.
- Many "pervasive" types of errors cause RuntimeExceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

```
public void mightFail (String file) {
    if (file.equals("dictionary.txt") {
        // file could be null!
    …
```

- The original intent was that such exceptions represent disastrous conditions from which it was impossible to sensibly recover…

Declared vs. Undeclared?

- Tradeoffs in the software design process:
- Declared = better documentation
  - forces callers to acknowledge that the exception exists
- Undeclared = fewer static guarantees
  - but, much easier to refactor code
- In practice: "undeclared" exceptions are prevalent

- A reasonable compromise:
  - Use declared exceptions for libraries, where the documentation and usage enforcement are critical
  - Use undeclared exceptions in client code to facilitate more flexible development