Programming Languages and Techniques (CIS120)

Lecture 30

November 13, 2015

Exceptions / IO

Chapter 27

Announcements

- HW7: PennPals Chat
  - Due: Tuesday, November 17th
  - Start *today* if you haven't already!

Poll

Have you started HW 07 PennPals?

1. No
2. I've thought about the ServerModel structure
3. I've got the basics working
4. I'm done

Exceptions

Dealing with the unexpected

Catching Exceptions

- There can be more than one “catch” clause associated with each “try”
  - Matched in order, according to the *dynamic* class of the exception thrown
  - The *first* clause that declares a supertype of the exception is triggered
  - Helps refine error handling

```
try {
        …    // do something with the IO library
} catch (FileNotFoundException e) {
        …    // handle an absent file
} catch (IOException e) {
        …    // handle other kinds of IO errors.
}
```

Finally

```
try {
    ...
} catch (Exn1 e1) {
   ...
} catch (Exn2 e2) {
   ...
} 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.
  - even if the method returns from inside the try.

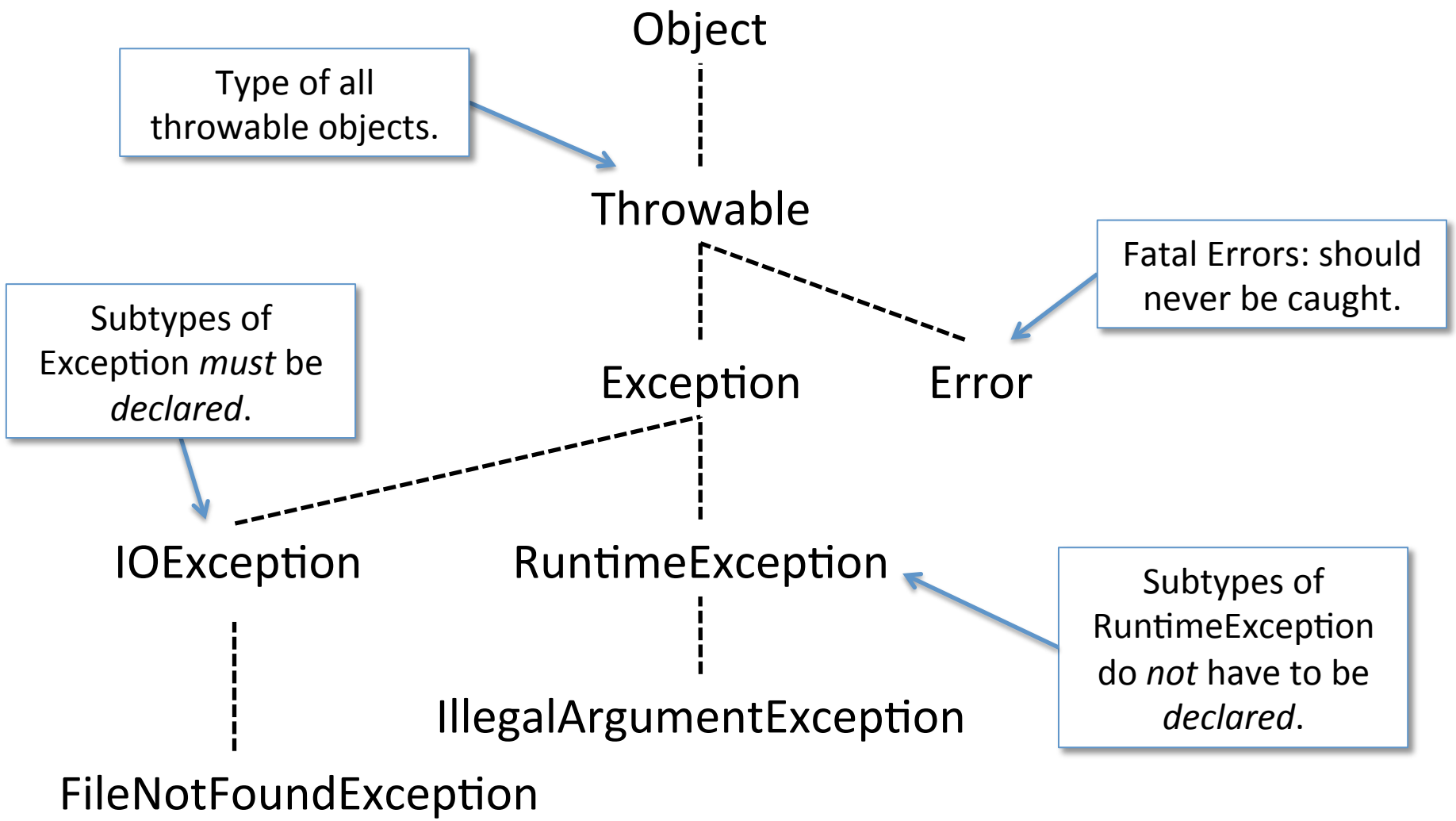
Using Finally

- `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) {…}
    }
  }
}
```

Informative Exception Handling

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.
- The method might raise a checked exception either by:
  - directly throwing such an exception

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

  - or by calling another method that might itself throw a checked exception

```
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…

Checked vs. Unchecked Exceptions

Which methods need a "throws" clause?

Note: IllegalArgumentException is a subtype of RuntimeException.

IOException is not.

Answer:
n, q and s should say throws IOException

```
public class ExceptionQuiz {
    public void m(Object x) {
      if (x == null)
          throw new IllegalArgumentException();
    }
    public void n(Object y) {
      if (y == null) throw new IOException();
    }
    public void p() {
      m(null);
    }
    public void q() {
      n(null);
    }
    public void r() {
      try { n(null); } catch (IOException e) {}
    }
    public void s() {
      n(new Object());
    }
}
```

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: test-driven development encourages "fail early/fail often" model of code design and lots of code refactoring, so "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

Good Style for Exceptions

- In Java, exceptions should be used to capture *exceptional* circumstances
  - Try/catch/throw incur performance costs and complicate reasoning about the program, don't use them when better solutions exist

- Re-use existing exception types when they are meaningful to the situation
  - e.g. use NoSuchElementException when implementing a container

- Define your own subclasses of Exception if doing so can convey useful information to possible callers that can handle the exception.

Good Style for Exceptions

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - e.g. when implementing `WordScanner` (in upcoming lectures), we catch `IOException` and throw `NoSuchElementException` in the `next` method.

- Catch exceptions as near to the source of failure as makes sense
  - i.e. where you have the information to deal with the exception

- Catch exceptions with as much precision as you can

BAD:
```
try {…} catch (Exception e) {…}
```
BETTER:
```
try {…} catch (IOException e) {…}
```

java.io

Poll

How many of these these classes have you used before CIS 120 (all part of the Java standard library)?

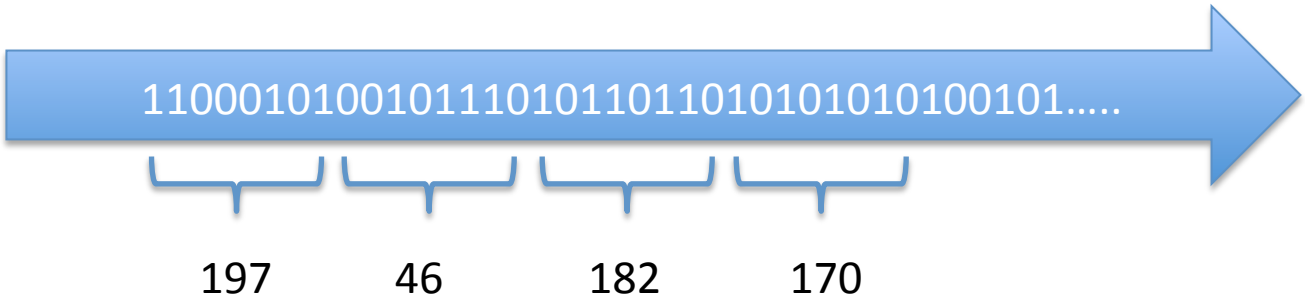
- Scanner
- Reader
- InputStream (e.g. System.in)
- FileReader
- BufferedReader
- Something else from java.io?

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.

Low-level 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:

```
int read ();          // Reads the next byte of data
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

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();
```

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

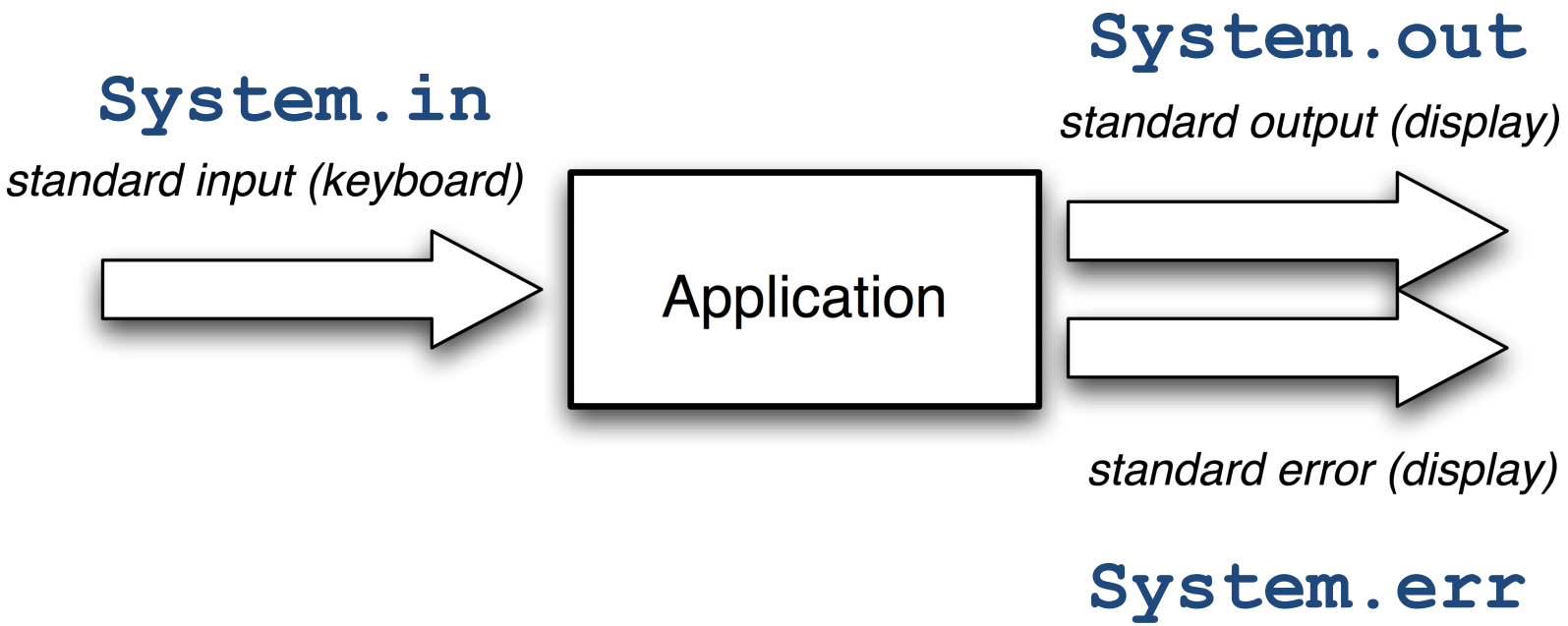
Buffering Example

```
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();
```

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.

PrintStream Methods

PrintStream adds buffering and binary-conversion methods to OutputStream

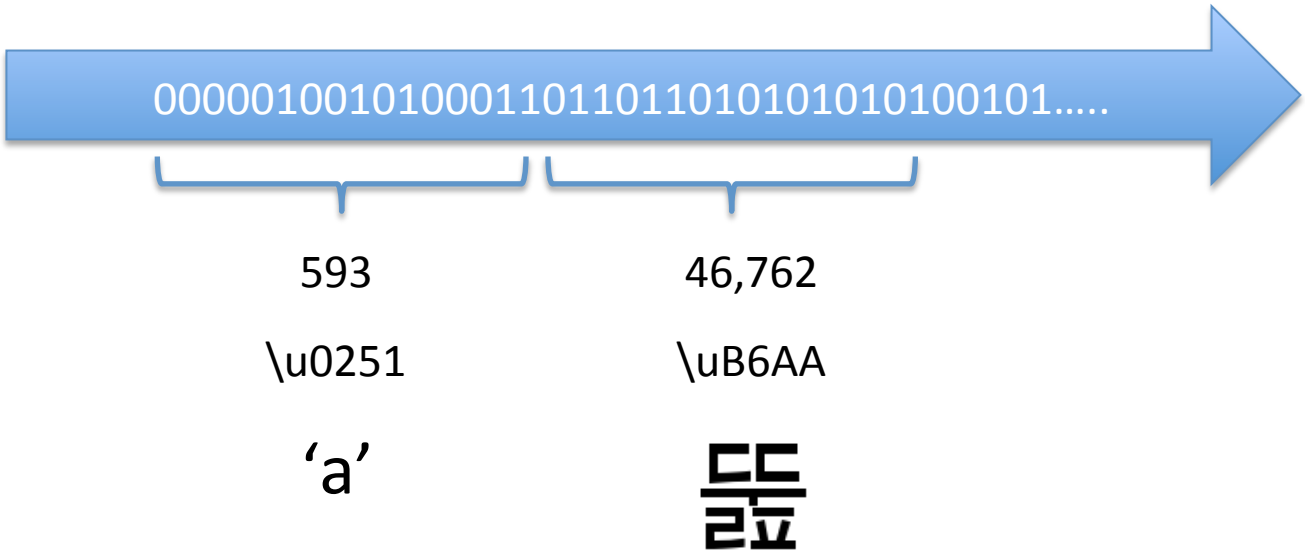
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

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

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:

```
int read ();          // Reads the next character
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
  - So wrap in an InputStreamReader / PrintWriter if you need unicode console I/O