Horstmann quality tips: exceptions

- "Throw early, catch late"
  - As soon as you don't know what to do – throw
  - Wait to catch until you're sure how to handle it
- "Do not squelch exceptions"
  - e.g., `catch (Exception e) { }` // "So there!"
    - Incompetent exception handlers create havoc later
- "Do throw *specific* exceptions"
  - Better: `throw new MyMeaningfulException();`
    - Easy handling: `catch(MyMeaningfulException e) {…}`
  - Worse: `throw new RuntimeException();`
    - Now meaningful handler hard to write – must identify problem

2 basic ways to store data

- Text format – a sequence of characters
  - e.g., 12345 is `'1' '2' '3' '4' '5'` (actually the binary equivalent of the Unicode values that represent these characters)
  - Purpose: for easy reading/editing by humans
    - Must translate to/from data – e.g., Integer.parseInt("12345")
- Binary format – a sequence of logical sets of bytes
  - e.g., 12345 stored as 4 bytes: `0 0 48 57` (48*256 + 57; actually the binary equivalent of these decimal values)
  - Purpose: fast reading/writing by computer
    - No need to translate – already data how the computer wants
- To Java they are 2 types of `Streams`

Character streams

- `Readers` – e.g., `FileReader`, `BufferedReader`
  - `FileReader.read()` returns just 1 character at a time
  - BufferedReader useful for its `readLine()` method
    - Routinely used to *pipe* FileReader through BufferedReader
  - Since Java 5, we can use a Scanner:

```
FileReader f = new FileReader("input.txt");
Scanner input = new Scanner(f); // ☺ (demos)
```

- `Writers` – e.g., `FileWriter`, `PrintWriter`
  - PrintWriter has the familiar print and println methods

```
FileWriter f = new FileWriter("message");
PrintWriter pr = new PrintWriter(f);
pr.println("Have a nice day.");
```

Byte streams

- `InputStream` and `OutputStream` – abstract superclasses (like `Reader` and `Writer`)
  - Basic methods – read() 1 byte; write(1 byte)
  - FileInputStream and FileOutputStream are subclasses
  - DataInputStream and DataOutputStream are too
- Note `System.in`, `.out`, and `.err` anomalies:
  - All are byte streams (from before character streams part of API) – but logically are character streams
    - System.in is an InputStream – but can pipe through a Reader
    - System.out and .err are PrintStreams – a deprecated subclass of FilterOutputStream – has same methods as PrintWriter
  - But careful: no "is a" relationship to Readers/Writers

Random access files

- Not "sequential access" (which has inflexible file pointer)
- Use random access for large, often-accessed files

```
RandomAccessFile f = new RandomAccessFile
("mydata", "rw"); // opens for read/write, not just "r"
f.seek(numBytes); // moves file pointer numBytes from start
```

  - Use DataInputStream and DataOuputStream methods:

```
f.writeInt(anInt); // writes and moves pointer 4 bytes
f.writeDouble(aDouble); // uses next 8 bytes
myNum = f.readInt(); // reads 4 bytes as int, moves pointer
```

- Must keep constant record size to be effective
  - See BankData example, pp. 829-832 (in chapter 19)

Object streams

- Most convenient way to store objects
  - Though usually not the most efficient way
- If `MyClass implements Serializable`, then

```
MyClass myObject = ...; // can read/write whole objects
ObjectOutputStream out = new ObjectOutputStream
(new FileOutputStream("myobjects")); // opens file
out.writeObject(myObject); // writes the whole object!
```

- Reverse everything to read it in another program

```
ObjectInputStream in = new ObjectInputStream
    ( new FileInputStream("myobjects")); // opens file
myObject = in.readObject(); // now use the object as is
```

- See SerialDemo example, p. 834-835

Some other streams

- `StringWriter`, `StringReader`
  - Handy string buffers; no IOExceptions are thrown
- `PipedOutputStream`, `PipedInputStream`
  - Handy way to write/read info between threads
- `java.net.URLConnection` – a handy way to read a stream over a network
  - First create a java.net.URL object:
    ```
    URL u = new URL("http://www...");
    ```
  - Then create the connection and get the InputStream:
    ```
    in = u.openConnection().getInputStream();
    ```

Introduction to Recursion

- Definition of a recursive method:
  A method that calls itself, directly or indirectly.
- Note: just intro – much more recursion in CS 20
  - For now just learn how it works – i.e., how to implement an algorithm we spell out for you
  - In the process, think about why it works
  - Begin to consider the range of applications
  - And know that you can *always* iterate instead
- The standard example: Fac.java

Recursive solution essentials

- Always need a base case
  - a.k.a. trivial case, or smallest case
  - A way to stop; otherwise infinite recursion
    - e.g., `if (n<=1)` in factorial method
- Recursive calls converge on base case
  - i.e., problems get smaller with each recursion
    - e.g., `factorial(n-1)`
- Solution must actually solve the problem!

Recursive Drawing Example

- Drawing tick marks on a ruler:
  - base case: draw nothing (tick too small)
  - general case: draw middle tick, then draw left and right "sub-rulers" (with smaller ticks)
  - Pseudocode:
    ```
    void ruler(int left, int right, int tickHeight) {
       if (not done yet) {
          int middle = (right - left) / 2;
          draw_tick(middle, tickHeight);
          ruler(left, middle, tickHeight / 2);
          ruler(middle, right, tickHeight / 2);
       }
    }
    ```

Recursive binary searching

- Start with a sorted array: `a[0..n-1]`
- Binary searching algorithm is naturally recursive:

```
int bsearch(Type key, Type a[], int left, int right) {
   /* first call is for left=0, and right=n-1 */
   int middle = (left + right) / 2;
   if (key == a[middle]) return middle; /* success */
   if (left > right) return -1; /* unsuccessful */
   if (key > a[middle]) /* search one half or the other */
      return bsearch(key, a, middle+1, right);
   else return bsearch(key, a, left, middle-1);
}
```

- Iterative version is a little trickier (but not too hard)