The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
1 February 2003

Notes – Exceptions

I. Function Requirements
   a. Partial functions don't accept all possible inputs – only a subset.
   b. Behavior is undefined if other values are sent.
      i. Could crash
      ii. Could enter an infinite loop.
      iii. Worst, perhaps, a result that *appears* valid could be returned.
   c. What to do if an input doesn't make sense?
      i. Return a special value (`null`, -1, etc)
         1. Not always possible. Maybe all possible values are legal.
         2. Caller needs to remember to check.
      ii. Modify an extra argument.
         1. Harder to call the function now.
         2. Caller still has to remember to check.
      iii. Throw an exception.

II. Exceptions
   a. The third option!
   b. The caller *cannot* forget to check!
   c. The function 'throws' an exception, which must be caught or thrown again by the caller.
   d. The object thrown is an instance of `Exception` or a subtype.
      i. Anyone can derive a new subtype
      ii. Can define extra fields and methods that give additional information about the error.
      iii. Some such methods are already defined.
      iv. `printStackTrace()`
         1. Shows the current state of the runtime stack.
         2. An easy way to get runtime information.
   e. If an exception is not caught, it propagates up until it finds a function that can catch it.

III. Exception Handling
   a. What should you do when you catch an exception?
   b. Four basic strategies.
   c. Fix the problem.
   d. Report an error and exit.
   e. Return from the function.
   f. Throw another exception.
      i. Maybe the same one.
      ii. Maybe a new one.
      iii. "Reflecting" the exception.

IV. Style
   a. If you might throw an exception, include a `throws` clause in the function header.
   b. MUST list all exceptions potentially thrown.
      i. Not listing one triggers a compiler error.
      ii. Could just say `throws Exception` but that doesn't help the programmer trying to use the function.
      iii. The exception to the rule is `RuntimeException` and its subclasses
         1. These are meant for "weird" errors for which you wouldn't likely be planning.
         2. Dereferencing null
         3. Dividing by zero
         4. Out of Memory
         5. About 50 classes, including `NullPointerException`
   c. The code becomes a little harder to understand both for the compiler and the programmer, since the flow of control breaks up a little more.

d. The code no longer executes in a user- or programmer-controlled sequence.
e. Sometimes you need to guarantee a set of statements will execute.
    i. Close the file you're using, release the lock on something, etc.
    ii. Use a `finally` block.
    iii. If an exception is thrown, `finally` executes and then it propagates.
    iv. `finally` is good even when no exceptions are involved because it runs even before returning from the function (if the return is done from inside a `try`.
    v. If you do catch the exception, `catch` will execute first, then run `finally`, then reflect the exception (if that's what your catch requested).
    vi. Essentially, one *cannot* leave the try block until the finally clause executes.
f. Use `@exception` in Javadoc.
    i. Should include one for everything that's in your `throws` clause.
    ii. Indicate the circumstances where it's thrown.
    iii. Indicate any likely runtime exceptions
    iv. If you listed a super-class, indicate which subclasses will actually be thrown.

V. Debugging Exceptions
a. BlueJ isn't very helpful at this.
b. If an un-handled runtime exception occurs, the debugger stops at the throw.
c. If it's handled somewhere, there's no easy/direct way to find the source.
d. If you created the subclass, put a breakpoint in its constructor.
e. jdb has a `catch ExceptionName` command to automatically stop at any applicable throw. Use `catch Exception` to stop for everything!