Exceptions in Java

- **Three kinds:**
  - Checked exceptions
    - Example: Missing file => java.io.FileNotFoundException
    - Must catch or specify
    - All exceptions are checked, except for Error, RuntimeException, and their subclasses
  - Errors (a kind of unchecked exception)
    - External to the application; usually cannot anticipate or recover from.
    - Example: Hardware I/O error => java.io.IOError
    - Not necessary to catch or specify
  - Runtime exception (a kind of unchecked exception)
    - Internal to the application; usually cannot anticipate or recover from.
    - Example: NullPointerException, division by zero, etc.
    - Not necessary to catch or specify
- **Questions:**
  - Why does Java force you to catch or specify checked exceptions?
    - Part of API
    - Caller must know what could happen so it can handle it
  - Why does java not force you to catch or specify runtime exceptions?
    - Caller cannot be expected to recover from bugs in callee.

Try-catch-finally

- **try { code; }**
  - Block of code that might throw an exception
  - If an exceptions is thrown, the rest of the block is skipped, and we move on to the catch handlers
- **catch (Type variable) { code; }**
  - Follows a try block
  - The appropriate handler (based on the type of the exception) is run if an exception is encountered in the try block
- **Finally**
  - The finally block always executes after the try block—regardless of whether unexpected exceptions occur
    - Runs after any handlers in the current scope
  - Cleanup code
  - How are the following different?
    - Example 1:

```
// void foo() throws MyException, java.io.FileNotFoundException;

try {
      foo();
} catch (MyException e) {
      System.out.println(e);
} finally {
```

```
        printf("Banana\n");
    }
```

- Example 2:

```
    // void foo() throws MyException, java.io.FileNotFoundException;

    try {
        foo();
    } catch (MyException e) {
        System.out.println(e);
    }
    printf("Banana\n");
```

- If foo throws a FileNotFoundException, example 1 will print Banana, but example 2 will not.

Run-time structure of the program

- **Abstract example**
  - Three functions: low, medium, and high
    - High calls medium, medium calls low
  - Actication records on the stack: high, medium, low
  - Low fails to do what it is supposed to, and throws an exception.
  - The stack unwinds
    - Local variables, etc., for low disappear
  - Check to see if medium will catch the exception.
  - If not, the stack unwinds more
    - Local variables, etc. for high disappear
  - Check to see if high will catch the exception
  - The catch block in high is the one that must decide how to recover.
    - But medium and low are gone from the stack.
    - Hard for high to actually recover from the error
      - Fix things so that medium works when called again?
      - Seem to need a lot of knowledge of exactly how medium works
        - Breaks modularity
- **Concrete example:**
  - Log parsing library, to be used by muliple applications
    - LogEntry parseLogEntry(String logLine)
    - Collection parseLogFile(String fileName)
  - parseLogEntry detects malformed log entries
    - But what to do?
    - Skip the entry? Give up on the rest of the file? Replace with some default entry? Count the number of malformed entried?
    - Wouldn't it be nice if we could go back and ask the application what to do, and then do it?

Common Lisp: Separate signaling, handling, and restarting

- **LogEntry parseLogEntry(String logLine)**
  - Throws a MalformedLogEntryError if logLine is malformed

- Otherwise returns a LogEntry
- **Collection parseLogFile(String Filename)**
  - Opens the file
  - Calls parseLogEntry on each line
  - Collects the results
  - But how to deal with malformed lines?
    - Attempt:

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
      for entry = (handler-case (parse-log-entry text)
                    (malformed-log-entry-error () nil))
      when entry collect it)))
```

    - Does too much!
    - parseLogFile decides to skip malformed entries.
    - That decision should be left to the caller of parseLogFile!
- **Restarts**
  - Low-level function defines named restarts
  - Exception handler in high-level function can decide what to do, then invoke one of those restarts to actually do it.
    - In log parsing library:

```
(defun parse-log-file (file)
  (with-open-file (in file :direction :input)
    (loop for text = (read-line in nil nil) while text
      for entry = (restart-case (parse-log-entry text)
                    (skip-log-entry () nil))
      when entry collect it)))
```

    - In application:

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (invoke-restart 'skip-log-entry))))
    (dolist (log (find-all-logs))
      (analyze-log log))))
```

  - Can have multiple restarts
    - In log parsing library:

```
(defun parse-log-entry (text)
  (if (well-formed-log-entry-p text)
    (make-instance 'log-entry ...)
    (restart-case (error 'malformed-log-entry-error :text text)
      (use-value (value) value)
      (reparse-entry (fixed-text) (parse-log-entry fixed-text)))))
```

    - In application

```
(defun log-analyzer ()
  (handler-bind ((malformed-log-entry-error
                  #'(lambda (c)
                      (use-value
```

```
                    (make-instance 'malformed-log-entry :text (text c))))))
(dolist (log (find-all-logs))
  (analyze-log log))))
```