C16: Exception Handling

how to recover gracefully from potentially error-producing situations

Motivation

- Exceptions are events that disrupt normal program flow; e.g. processing a file:
  - Ask the user for filename
  - Open file
  - Read contents
  - Do something with contents
  - Close file
- Lots of potential problems (file not found, not readable, …)
- Java supports strong distinction between normal and abnormal program flow

Force the programmer

- Must deal with exceptions: try/catch

```
try {
    File fd = new File(name);
    processFile(fd);
    fd.close();
} catch (FileNotFoundException e) {
    System.err.println("Cannot find file " + name);
} catch (FileOpenFailed e) {
    System.err.println("Cannot open file " + name);
}
```

- Much better than returning some magic value in case of error

Information transmitted to catch

- Similar to a method header catch defines a variable with type Throwable (or a subclass thereof)
- Contains useful information, e.g. stack trace, can be very useful for debugging:

```
try {
    File fd = new File(name);
    processFile(fd);
    fd.close();
} catch (FileNotFoundException e) {
    System.err.println("Cannot find file " + name);
    e.printStackTrace();
}
```

Catching multiple exceptions

- No one-to-one mapping between exceptions and try/catch blocks
- But exceptions must be handled eventually

```
try {
    File fd1 = new File(name);
    File fd2 = new File(name);
    processFile(fd1,fd2);
    fd1.close();
    fd2.close();
} catch (FileNotFoundException e) {
    System.err.println("Cannot find file " + e);
} catch (FileOpenFailed e) {
    System.err.println("Cannot open file " + e);
}
```

The finally clause

- Optional last clause in a try/catch block
- Will always be executed, no matter what exceptions have been thrown and handled:

```
try {
    throw new EOFException();
} catch (EOFException e) {
    e.printStackTrace();
    throw new ClassCastException();
} finally {
    System.out.println("Finally");
}
```

- Will even execute when "return" is used!

Termination or resumption

- Java (as most languages) has “terminating semantics” for exceptions:
  - You need to explicitly redo the prematurely terminated computation
- Some languages (CommonLisp, Dylan) allow to specify handlers/restart to deal with exceptional situations allowing computation to continue at the point where an exception was raised
  - E.g.: FileNotFound:
    - open a FileSelector dialog to have the user choose some file,
    - just start up an interactive debugger,
    - …

Exception hierarchy

- Throwable
  - Error
    - LinkageError
    - VirtualMachineError
    - …
  - Exception
    - IOException
    - RuntimeException
      - ClassCastException
      - …
    - …
- Every exception is an instance of a subclass of Throwable
- Programmers are not forced to deal with Error and RuntimeException cases (too common)

Throwing exceptions

- A method that throws (or passes on) an exception, must indicate so:

```
Object pop() throws Exception {
    … throw new Exception(“pop empty stack”);
    …}
```

- Better: define new Exception subclass:

```
class StackUnderFlowException extends Exception {
    public StackUnderFlowException () {super(); }}

… throw new StackUnderFlowException(); …
```

- If a method is overridden in a subclass, the “throws Exception” specification cannot be changed (similar to the way return types are handled)

Passing on exceptions

- Sometimes it is best to just pass on exceptions (because we don’t know locally what to do about it):

```
public void readLines( …) throws IOException
    { …  input.readLine(); … }
```

- readLine() can throw an IOException, we don’t deal with it (no try/catch block), so this method may also throw an IOException