CS 61B: Lecture 15
Wednesday, February 26, 2014

Today's reading: Sierra & Bates, pp. 315-338.

EXCEPTIONS
==========
When a run-time error occurs in Java, the JVM "throws an exception," prints an error message, and quits. Oddly, an exception is a Java object (named Exception), and you can prevent the error message from printing and the program from terminating by "catching" the Exception that Java threw.

Purpose #1: Coping with Errors

Exceptions are a way of coping with unexpected errors. By catching exceptions, you can recover. For instance, if you try to open a file that doesn't exist or that you aren't allowed to read, Java will throw an exception. You can catch the exception, handle it, and continue, instead of letting the program crash.

```
  try {
    f = new FileInputStream("~cs61b/pj2.solution");
    i = f.read();
  }
  catch (FileNotFoundException e1) {
    System.out.println(e1);                       // An exception handler.
  }
  catch (IOException e2) {
    f.close();                                    // Another exception handler.
  }
```

What does this code do?
(a) It executes the code inside the "try" braces.
(b) If the "try" code executes normally, we skip over the "catch" clauses.
(c) If the "try" code throws an exception, Java does not finish the "try" code. It jumps directly to the first "catch" clause that matches the exception, and executes that "catch" clause. By "matches", I mean that the actual exception object thrown is the same class as, or a subclass of, the static type listed in the "catch" clause.

    When the "catch" clause finishes executing, Java jumps to the next line of code immediately after all the "catch" clauses.

The code within a "catch" clause is called an _exception_handler_.

If the FileInputStream constructor fails to find the file, it will throw a FileNotFoundException. The line "i = f.read()" is not executed; execution jumps directly to the first exception handler.

FileNotFoundException is a subclass of IOException, so the exception matches both "catch" clauses. However, only one "catch" clause is executed--the first one that matches. The second "catch" clause would execute if the first were not present.

If the FileInputStream constructor runs without error, but the read() method throws an exception (for instance, because a disk track is faulty), it typically generates some sort of IOException that isn't a FileNotFoundException. This causes the second "catch" clause to execute and close the file. Exception handlers are often used to recover from errors and clean up loose ends like open files.

Note that you don't need a "catch" clause for every exception that can occur. You can catch some exceptions and let others propagate.

Purpose #2: Escaping a Sinking Ship

Believe it or not, you might want to throw your own exception. Exceptions are the easiest way to move program execution out of a method whose purpose has been defeated.

For example, suppose you're writing a parser that reads Java code and analyzes its syntactic structure. Parsers are quite complicated, and use many recursive calls and loops. Suppose that your parser is executing a method many methods deep within the program stack within many levels of loop nesting. Suddenly, your parser unexpectedly reaches the end of the file, because a student accidentally erased the last 50 lines of his program.

It's quite painful to write code that elegantly retraces its way back up through the method calls and loops when a surprise happens deep within a parser. A better solution? Throw an exception! You can even roll your own.

```
  public class ParserException extends Exception { }
```

This class doesn't have any methods except the default constructor. There's no need; the only purpose of a ParserException is to be distinguishable from other types of exceptions. Now we can write some parser methods.

```
  public ParseTree parseExpression() throws ParserException {
    [loops]
        if (somethingWrong) {
          throw new ParserException();
        }
        [more code]
      }
    }
    return pt;
  }
```

The "throw" statement throws a ParserException, thereby immediately getting us out of the routine. How is this different from a "return" statement? First, we don't have to return anything. Second, an exception can propagate several stack frames down the stack, not just one, as we'll see shortly.

The method signature has the modifier "throws ParserException". This is necessary; Java won't let you compile the method without it. "throws" clauses help you and the compiler keep track of which exceptions can propagate where.

```
  public ParseTree parse() throws ParserException, DumbCodeException {
    [loops and code]
        p = parseExpression();
        [more code]
      }
    }
  }

  public void compile() {
    ParseTree p;
    try {
      p = parse();
      p.toByteCode();
    }
    catch (ParserException e1) { }
    catch (DumbCodeException e2) { }
  }
```

The parse() method above shows how to define a method that can throw two (or more) exceptions. Since every exception is a subclass of Exception, we could have replaced the two exceptions with "Exception", but then the caller would have to catch all types of Exceptions. We don't want (in this case) to catch NullPointerExceptions or otherwise hide our bugs from ourselves.

When parseExpression() throws an exception, it propagates right through the calling method parse() and down to compile(), where it is caught. compile() doesn't need a "throws ParserException" clause because it catches any ParserException that can occur. In this code, the "catch" clauses don't do anything except stop the exceptions.

If an exception propagates all the way out of main() without being caught, the JVM prints an error message and halts. You've seen this happen many times.

Checked and Unchecked Throwables

The top-level class of things you can "throw" and "catch" is called Throwable. Here's part of the Throwable class hierarchy.

```
                                   Throwable
                                  /         \
                                 /           \
                                /             \
                               /               \
                              /                 \
                     Exception                   Error
                    /    \                      /     \
         IOException  RunTimeException AssertionError  VirtualMachineError
                       /            \                           \
        NullPointerException  ClassCastException              OutOfMemoryError
```

An Error generally represents a fatal error, like running out of memory or stack space. Failed "assert" statements also generate a subclass of Error called an AssertionError. Although you can throw or catch any kind of Throwable, catching an Error is rarely appropriate.

Most Exceptions, unlike Errors, signify problems you could conceivably recover from. The subclass RunTimeException is made up of exceptions that might be thrown by the Java Virtual Machine, such as NullPointerException, ArrayIndexOutOfBoundsException, and ClassCastException.

There are two types of Throwables. _Unchecked_ Throwables are those a method can throw without declaring them in a "throws" clause. All Errors and RunTimeExceptions (including all their subclasses) are unchecked, because almost every method can generate them inadvertently, and it would be silly if we had to declare them.

All Exceptions except RunTimeExceptions are _checked_, which means that if your method might throw one without catching it, it must declare that possibility in a "throws" clause. Examples of checked exceptions include IOException and almost any Throwable subclass you would make yourself.

When a method calls another method that can throw a checked exception, it has just two choices.

(1) It can catch the exception, or
(2) it must be declared so that it "throws" the same exception itself.

The easiest way to figure out which exceptions to declare is to declare none and let the compiler's error messages tell you. (This won't work on the exams, though.)