CS61B Summer 2006
Instructor: Erin Korber
Lecture 7, 6 June
Reading for tomorrow: Ch. 9

Exceptions

When a run-time error occurs in Java, the JVM "throws an exception," and the result is an error message. An exception is a Java object (of type Exception), and you can prevent the error message from printing and the program from terminating by "catching" the Exception that Java threw.

1 Surviving Errors

By catching exceptions, you can recover from an unexpected error. 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, print an error message, and continue, instead of letting the program crash.

```
try {
  f = new FileInputStream("~cs61b/pj2.solution");
  i = f.read();
}
catch (FileNotFoundException e1) {
  whine("Foiled!!");
}
catch (IOException e2) {
  f.close();
}
```

What does this code do?

1. It executes the code inside the "try" braces.
2. If the "try" code executes normally, we skip over the "catch" clauses.
3. 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 exception type listed in the "catch" statement.

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 open the file, it will throw a FileNotFoundException. The line "i = f.read()" is not executed; execution jumps directly to the whiny 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 or didn't match the exception type.

If the FileInputStream constructor runs without error, but the read() method throws an exception (for instance, because the disk 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.

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 levels 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 fly 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. Also, 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.

If an exception propagates all the way out of main() without being caught, the JVM prints an error message and halts.

3 Checked and Unchecked Exceptions

Some exceptions do not need to be declared in a "throws" clause. For instance, any class that uses a reference might throw a NullPointerException (as most of you have experienced), but there's no need to declare a "throws" clause for it. NullPointerException, ArrayIndexOutOfBoundsException, and ClassCastException are unchecked exceptions. All unchecked exceptions are subclasses of RuntimeException. Nearly any class can generate run-time exceptions, and it would be silly if we had to declare them.

Other exceptions, including IOException and most exceptions you will declare yourself, are checked. 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 can throw 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.

Both checked and unchecked exceptions can be caught.

4 Throwable

Exception is a subclass of Throwable. There's one other subclass of Throwable, called Error. An Error generally represents a fatal error, like running out of memory. An Exception generally represents a recoverable error.

You can throw or catch any kind of Throwable, but catching an Error is rarely appropriate. Errors, like RuntimeExceptions, are unchecked.