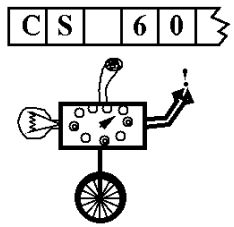


Finishing up Java

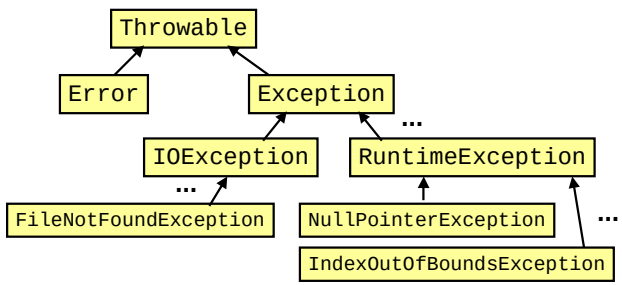
October 15, 2001

Generating Exceptions

- An **exception** is a non-standard exit from a piece of code
- Several ways code can cause an exception:
  - JVM failure
    - Out of memory, class not found, ...
  - Java language's run-time checks
    - Illegal casts, out-of-bounds array accesses, invoking methods on a `null` object reference, ...
  - Calling library routines with "wrong" arguments
    - `Integer.parseInt` will throw `NumberFormatException` if the given string doesn't contain a number
  - Invoking `throw`

Exception Objects

- Information about what caused the exception is packaged as an object of class `Throwable` (or a subclass)



Exception Lingo

- When an exception occurs it is said to be *thrown*
- If an exception is thrown inside a method, it can be *caught* (detected and handled); otherwise it is *passed* (sort of like a hot potato)
- Exceptions propagate up the call stack:
  - When an exception is thrown, the system checks to see if the exception should be caught.
  - First checks the current method
  - If not caught by the current method, looks at the method which called the current method
  - If not caught by the caller, looks at the caller's caller, etc.
  - If nobody catches it, the program gives up and aborts.

Typical Exception Handling

- Main keywords are:
  - `try`: execute some code, known as a *try-block,* in which an exception might be thrown
  - `catch`: handle the exception caused by the try-block if one is thrown
  - `finally`: (*optional*) code executed after a try-block whether or not an exception was thrown
- Also (with some important exceptions) every method which might throw exceptions must list the possible ones with `throws`

Example: Echoing S-expressions

```
class echoSexp {
  public static void main(String arg[]) {
    Tokenizer in = new Tokenizer(System.in); // S exp tokenizer
    Object ob;                               // Object to be read

    System.out.println("ready");
    while((ob=in.nextSexp()) != Tokenizer.eof ) {
      System.out.println(ob);              // echo the value
    }
    System.out.println();                 // new line
  }
}
```

class `InputStream`

Now want to read from a file

CAUTION: won't compile

```
import java.io.*;

class echoFile
  {
  public static void main(String arg[]) {
    InputStream inStream = System.in;
    if( arg.length > 0 ) {
      String filename = arg[0];
      inStream = new FileInputStream(filename);
      }

    Tokenizer in = new Tokenizer(inStream);
    ...
    // other code as before
}
```

The Problem

- The file specified by the user may not exist
  - Attempting to open a non-existent file will throw `IOException`
- The method `main` must catch this exception or the compiler rejects the code

Corrected Version

```
// as before
...
InputStream inStream = System.in;
if( arg.length > 0 ) {
  String filename = arg[0];
   try {
    inStream = new FileInputStream(filename);
   }
   catch( IOException e ) {
    System.err.println("*** unable to open file: "
                       + filename);
    System.exit(1);
    }
}
Tokenizer in = new Tokenizer(inStream);
...
// other code as before
```

General Form of Try-Catch-Finally

```
try {
  . . .
  }
catch( ExceptionClass1 e ) {
  . . .
  }
catch( ExceptionClass2 e ) {
  . . .
  }
. . .  // more catch blocks if desired
finally  {
  . . .
  }
```

optional, always executed if present

Java Subtleties

- The catch clause

```
catch( ExceptionClass1 e ) {
   ...
}
```

  catches those exceptions of class `ExceptionClass1` *or any subclass*
  - How could we write one `catch` for all exceptions?
- Methods which can throw exceptions must say so

```
static int fact(int n) throws FactInputException {
  ...
}
```

  - Exception: don't have to mention `Errors` or `RuntimeExceptions`

Matters of Style

- It is poor style to write code where "normal" behavior generates exceptions
  - Difficult to understand the code
  - Probably inefficient
- It is arguably poor style to make subclasses of `RuntimeException` instead of `Exception` without a good reason.

Some Code

```
class FactInputException extends Exception {
  public FactInputException(String s) { super(s); }
}
class Factorial {
  static int factorial(int n) throws FactInputException {
    if (n < 0) throw new FactInputException("Bad input");
    if (n == 0) return 1;
    return (n * factorial(n-1));
  }
  public static void main(String[] args)
                              throws FactInputException {
    int n = Integer.parseInt(args[0]);
    System.Out.println(factorial(n)+"");
  }
}
```

```
> java Factorial
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(fact3.java, Compiled Code)
> java Factorial 5
120
> java Factorial -2
Exception in thread "main" FactInputException: Bad input
        at Factorial.factorial(fact3.java, Compiled Code)
        at Factorial.main(fact3.java, Compiled Code)
```

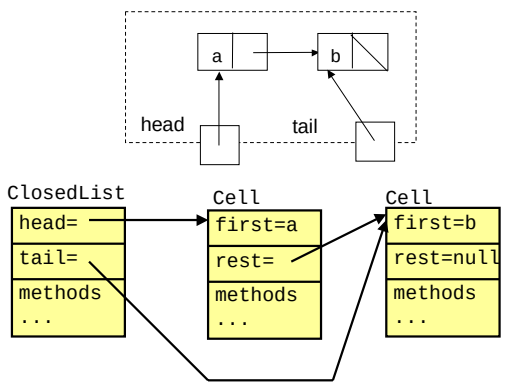
Really Bad Code

```
class FactAnswerException extends Exception {
  public FactAnswerException(String s) { super(s); }
}

class Factorial {
  static int factorial(int n) throws FactAnswerException {
    if (n < 0) return 0;
    if (n == 0) throw new FactAnswerException("1");
    try
      { return (factorial(n-1)); }
    catch (FactAnswerException s) {
      int ans = n * Integer.parseInt(s.getMessage());
      throw new FactAnswerException(ans+"");
    }
  }
  public static void main(String[] args)
            throws FactAnswerException {
    factorial(5);
  }
}
```

```
> java Factorial
Exception in thread "main" FactAnswerException: 120
        at Factorial.factorial(fact2.java, Compiled Code)
        at Factorial.main(fact2.java, Compiled Code)
```

Recall the Closed List Implementation



Copying Objects

- What does it mean to duplicate/copy/clone this `ClosedList` object?

Copying Objects

- Shallow Copying
  - Copying only the information in the object itself
  - Copies the references to other objects
- Deep Copying
  - Makes fresh copies of the object and everything it references, and everything they reference, etc.
- These are the endpoints of a *range* of possibilities
- Java has built-in support for shallow copying (`clone`)
  - Have to explicitly allow the object to be cloned; not completely straightforward.
  - You often don't want shallow copying anyway.