USING EXCEPTIONS EFFECTIVELY

James W. Cooper

Recently I talked with Peter Haggar (1) about exceptions in Java and read some comments by Ken Arnold that got me thinking about how little we actually do with exceptions in Java programs. I have seen few examples of how to use exceptions effectively in actual programs and the advice in most programming texts and articles is rather brief.

To review, the Java runtime system throws exceptions for a number of error conditions, such as

- No such file
- Read past end of file
- Input errors
- Number format error
- Array index out of bounds
- Null pointer

There are, of course, dozens of other exceptions, especially in the Java Foundation Classes.

Most input and output class methods throw exceptions, and if the method declares that it throws an exception, the Java compiler requires you to catch that exception. The general form of these exceptions is

```
try{
      //statements...
}
catch (SomeException e)
{
      //handle exception
}
finally
{
      //code always executed
}
```

If one of the statements inside the try block fails and throws and exception, the code inside the catch block is executed. Then, whether there is a failure or not, code inside the *finally* block is executed.

You can also catch a number of different exceptions and deal with them in separate catch blocks

```
try {
 //open some sort of file
}
catch(FileNotFoundException fn)
      {     //handle file not found
      }
```

```
catch (EOFException e)
      {     //handle eof
      }
catch (IOException e)
      {     //handle general IO failure
      }
```

It is important, however, that you start with the most specific, derived exception and end with the most general exception.

Careless Exception Handling

Too often, we write exception handling code to be too simple -- if the compiler compels us to catch an exception, we do the minimum we can get away with:

```
try {
 f = new RandomAccessFile(fname, “r”);
}
catch (IOException e)
{System.out.println(“Can’t open file”);}
```

And usually, printing out a message is the least useful thing you can do, especially in a GUI, windowing system, where the printed message may not show up anywhere anyone will ever see it. Worse yet, the effect of some of these exceptions is to make the program unusable, so that it soon folds up its tents and steals away, without doing any useful work.

With these objections in mind, I decided to write a program that uses a couple of exceptions in the ways the designers of Java actually intended, and keeps running even though the input data contain some pretty sloppy errors.

An Input Data File

Let’s assume that we have a set of data representing results of a swim meet that we need to read into a program. We’ll further suppose that these data were typed into a file manually and therefore contain some typing errors. The data we start with are:

```
1 Amanda Mc Carthy            12  WCA         29.28
2 Jamie Falco                 12  HNHS        29.80
3 Meaghan O' Donnell          12  EDST        30.00
4 Greer Gibbs                 l2  CDEV        30.04
5 Rhiannon Jeffrey            11  WYW         3O.O4
6 Sophie Connolly             12  WAC
                                              30.05
7 Dana Helyer                 12  ARAC        30.18
8 Lindsay Marotto             12  OAK         30.23
9 Sarah Treichel              12  WYW         30.35
10 Ashley McEntee             12  RAC
```

These data are perfectly readable by a human being, but contain the kind of errors that drive a computer program bananas. For example,

1. While we assume one first name and one last name, two swimmers have spaces in the middle of their last names.

2. The fourth swimmer has a lower-case “L” instead of a one for the first digit of her age.
3. The fifth swimmer has letter O’s instead of zeroes in her time.
4. The sixth swimmer has an extra line feed between the club initials and her time.
5. The last swimmer’s time is missing
6. There is an extra blank line in the file after the last swimmer.

Most of these problems will generate exceptions if we just try to read in each line and assign the data to a particular kid. We could catch those exceptions and throw up our hands, or we could try to be a little more thoughtful in figuring out how to recover from these errors.

The Philosophy of Exceptions

Exceptions in Java are intended to handle unexpected occurrences. If you can and should be testing for a condition (such as end-of file) directly n your code, then the exception is the wrong way to handle the condition. Of course this is a very general sort of rule and can be interpreted in lots of ways. One thing you should realize is that exceptions are fairly expensive to throw and each thrown exception can reduce the performance of the program. This suggests that creating lots of new kinds of exceptions within your class hierarchy could be a bad idea. On the other hand, *using* exceptions that the system provides or requires is a pretty good coding practice, and of course, in many instances, enforced by the compiler.

A Kid Class

In order to handle this “dirty data,” we’ll create a Kid class that parses each line of the file and creates a Kid object which contains as many of the data values as it can figure out. The way to handle our data ingestion problem is to create a Kid object for each line in the file and check each line in a separate method. If it can’t find at least a first name, last name and club, it sets the *legal* state to *false* and skips over that line of data.

We are going to need to look ahead to the next token without committing ourselves to its use so we can see if the last name contains a space. So we derive the *saveTokenizer* class from *StringTokenizer*. This class contains a *pushToken* method that puts a token back if it isn’t the one we need.

Our basic Kid class constructor takes a line from the data file and parses it into the names, age, club and time values:

```
public Kid(String s)   {
  tok = new saveTokenizer(s);
  //must be at least one
  if(tok.hasMoreElements()){
     legal = true;      //Start as legal
    //discard line number
     String lnum = tok.nextToken();
     getFrname();       //read first name
     getLname();        //last name
     getAge();          //convert age
     getClub();         //read club
     getTime();         //convert time
   }
     else
        legal = false;  //otherwise not legal kid
  }
```

There is nothing special about reading the first and last name – either they are there or they aren’t:

```
private void getFrname()   {
  if(tok.hasMoreElements())
      fr_name = tok.nextToken();
  else
     legal = false;
}
//----------------------------
 private void getLname()   {
   if(tok.hasMoreElements())
      l_name = tok.nextToken();
  else
      legal = false;
}
```

Looking Ahead to the Ages

However, when we read the age token, it is possible that the first token we encounter is really part of a last name which contains an intervening space. If this token is not numeric, we add it to the last name and get the next one for the age:

```
private void getAge()   {
     String agstring= "";
     try{
        agstring = tok.nextToken();
        kid_age = new Integer(agstring).intValue();
     }
     catch(NumberFormatException e)      {
        kid_age = 0;    //no legal age
        //look ahead for next token as legal
        testInteger testAge = new testInteger(tok.nextToken());
        if (testAge.isLegal())           {
           kid_age = testAge.toInteger(); //age is in next token
           l_name += agstring;      //this is part of last name
           }
        else     {
           kid_age = 0;       //no age
           tok.pushToken();   //push next token back
        }
     }
     catch(NoSuchElementException n)
     {legal = false;}
    }
```

Here we see an excellent example of using the exception mechanism to catch a number conversion error and how we can recover from it in the *catch* portion of the code. If the token is *not* a number, the exception is triggered. If the following token *is* a number then the age is set to that number and the previous token appended to the last name. If the following token is *not* an integer then it is pushed back to be reread. This would occur if he actual age were mistyped and not convertible to an integer.

This routine actually illustrates two ways of dealing with integer conversion errors. We also create an instance of the *testInteger* class, which encapsulates the *Integer* class and allows you to ask if the token it has just engulfed is an integer or not. Note that since the *Integer* class is a *final* class we can't derive *testInteger* from it, but must encapsulate Integer and bring some of its methods to the surface of the enclosing class:

```
class testInteger {
   //an integer class that tells you whether the current
   //integer is a legal number or not
   //since Integer is a final class
   //we encapsulate an Integer and use a few of its methods
   private boolean legal;
   Integer test;

   public testInteger(String s)    {
      try{
         test = new Integer(s);
         legal = true;
      }
      catch(NumberFormatException e) {
         legal = false;
      }
   }
   //-----------------------------
   public boolean isLegal()    {
      return legal;
   }
   //-----------------------------
   public int intValue()    {
      return test.intValue();
   }
}
```

Keeping with the Times

There is nothing special about how we get the club token. If there aren't any tokens left, the legal flag is set to false. The time method works much like the age conversion, however, where we catch the float conversion exception and set the resulting time to zero. We don't make the entire kid's data illegal however, because we can keep the kid and correct the time value later:

```
private void getTime()   {
     Float tim;
     try{
      tim = new Float(tok.nextToken());
      kid_time = tim.floatValue();
     }
     catch (NumberFormatException e)      {
       kid_time = 0;
     }
     catch(NoSuchElementException e)
       {kid_time = 0;}
  }
```

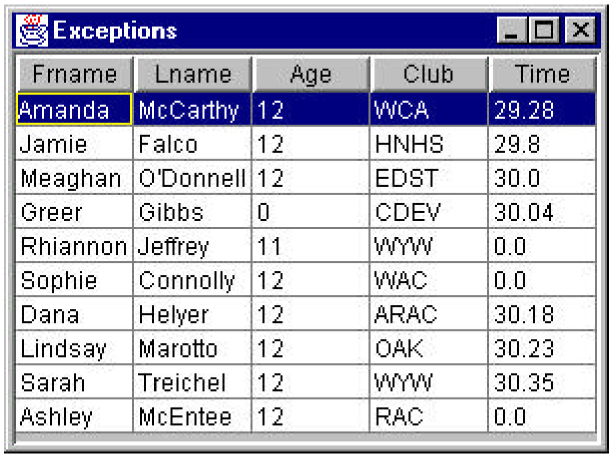
Note again that if we run out of tokens in the tokenizer we catch the NoSuchElement exception and go on with a zero time being recorded.

Showing the Results

Our main program loads all of these Kid objects into a Table Data model

```
KidModel tmodel = new KidModel(); //table data model
InputFile f = new InputFile("50free.txt");
String s = f.readLine();
while( s != null)
  {
  k = new Kid(s);
  if(k.isLegal())
     tmodel.add(k);
  s = f.readLine();
 }
JTable table = new JTable(tmodel);
```

and displays them as shown below:



Frname	Lname	Age	Club	Time
Amanda	McCarthy	12	WCA	29.28
Jamie	Falco	12	HNHS	29.8
Meaghan	O'Donnell	12	EDST	30.0
Greer	Gibbs	0	CDEV	30.04
Rhiannon	Jeffrey	11	WYW	0.0
Sophie	Connolly	12	WAC	0.0
Dana	Helyer	12	ARAC	30.18
Lindsay	Marotto	12	OAK	30.23
Sarah	Treichel	12	WYW	30.35
Ashley	McEntee	12	RAC	0.0

Note that we have successfully rejoined the two last names, have indicated the 4th age as zeroand the 5th, 6th and 10th times as zero, meaning that they probaby require manual intervention to correct.

You may note that we introduced the InputFile class above. This is itself a simple class thatencloses file I/O exceptions and we have discussed it previously (3).

Summary

Exceptions don't always lead to rapid program exits. If they did we could just leave them uncaught. Instead we've seen here how to use exception handling to improve error checking and the robustness of your code in the face of faulty user input.

References

1. Peter Haggar, "Java Exception Handling" IBM e-Business and Networking Systems Technical Converence, Las Vegas, September, 1998.

2. James Gosling, Frank Yellin, and The Java Team, *The Java Application Programmng Interface, Vol. 1, Core Packages,* Addison-Wesley, 1996.

3. James Cooper, *Principles of Object-Oriented Programming in Java 1.1*, Ventana/Coriolis, 1997.