Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-seven)

7.2: Exception Handling

Imagine this: you're writing a class that represents a database of students and their grades, schedules, etc. The constructor for this class accepts the name of a disk file as an argument; it reads the student information out of the file. What should the constructor do if the file does not exist? Remember, a constructor has no return value.

One possible solution is to set a special boolean "dead" member variable inside the object. Each of the object's member functions would then have to check the "dead" member before doing anything; if the flag was true, they could return a error value. For example, if the database class had an n_students( ) member, it could return -1 if the object was dead. Of course all the code that calls n_students( ) would need to check to see if it returned -1, and if so, it would have to handle things in some reasonable way. Whether the caller checked the error return value or not, though, at least the database class would always behave correctly. This solution is workable, but it adds a lot of essentially useless error-handling code to the application and to the database class.

Another possibility would be to set a dead flag, as above, in the database class, and trust the code that creates the database object to immediately discard the database if it turned out to be dead. This simplifies the code in the database class, because the individual methods don't need to check the dead flag, and in the application, because no dead database object should ever be used. But the word "trust" always should send your eyebrows shooting up. A class should never trust its caller to always do the right thing, as we've discussed in the past. A truly robust class doesn't trust code outside the class to do anything right!

The right answer in this case is neither of the two solutions above. Java provides a far more elegant solution: the use of exceptions.

```
import java.io.FileNotFoundException;

...

Database db = null;
try
  {
     db = new Database("Students.dbf");
  }
catch (FileNotFoundException fnfe)
  {
     System.out.println("Database not found: "
                         + fnfe);
     System.out.println("Using default database");
     db = new Database("Default.dbf");
  }
   System.out.println("Database Initialized");

   ...
```

```
    public class Database
    {
      public Database(String filename)
             throws FileNotFoundException
      {
      if (!loadFile(filename))
         throw new FileNotFoundException("Cannot
                                  load
                                  Database");
      }
      // lots of code omitted, including
      //boolean loadFile(String name)
}
```

There are a lot of things going on here, so we'll look at it one step at a time, in the order that the code might execute.

The try keyword means precisely that: try to do something that might fail. In this case, we're going to try to create a Database object from the file Students.dbf. The flow of control transfers to the constructor function in class Database. Note that what follows the try keyword must be a block enclosed in curly brackets, even though it is just a single line of code. Note also that we must first declare the db variable outside of the try block if we are going to use it later in the program, so that it is not local to the try block!

In the Database constructor, we call a member function loadFile( ), which presumably returns true on success. Let us first imagine that loadFile returns true. In this case, the if( ) test does not succeed, and the constructor returns normally. This puts us back in the mainline code, which assigns the newly constructed Database to the db variable. Flow of control proceeds then to the line that prints the "Database initialized" message.

But if instead loadFile( ) returns false, something very different happens. In this case, the if( ) succeeds, and we find ourselves at the line of code beginning with throw. We see a java.io.FileNotFoundException object being created (it's just an ordinary class object). What becomes of this object?

It helps to envision Robinson Crusoe stranded on his desert island. His only hope of communicating with the outside world is to write a message on a scrap of paper, stuff it into a bottle, and throw the bottle into the ocean. That is exactly what is happening here. By creating an exception object and throwing it, the constructor is sending a message to any code that advertises an interest in hearing about what went wrong. The Java runtime system takes that exception object and looks in the function that called this function for a *catch* block. A catch block is essentially a target for messages in bottles that might be thrown by code in trouble. The catch block in the mainline code above specifies that it is only interested in FileNotFoundException messages (actually, it would also receive subclasses of this exception type, if there were any). If there were not such a catch block in this function, Java would look in the function that called this function, and so on, until either such a block was found or until there were no more functions to check. (When that happens, the program is halted and a stack trace is printed out.)

A catch block must appear directly after the try block in which the code that might fail occurs. You can specify a number of catch blocks, one after another, all associated with a single try block. Each catch block must specify a different exception type.

Notice that the catch block looks a little like a function definition:

```
catch (FileNotFoundException fnfe) { ... }

int ketch (SomeOtherObjectType soot) { ... }
```

The resemblance is functional as well as visual. When the code in the catch block's pair of curly braces is executed, the variable fnfe has been initialized to refer to the thrown FileNotFoundException object. It's just like a function's formal argument list, except a catch block always has only one argument. Notice how the catch block above prints out the FileNotFoundException object.

A few more words about syntax are in order. First, a method must declare all the exceptions that it can throw, just as the Database constructor declares that it can throw FileNotFoundException. The compiler enforces this.

Exception classes such as FileNotFoundException are ordinary Java classes, although to be thrown, a class must inherit from java.lang.Throwable. Exceptions representing severe problems generally inherit from java.lang.Error, while those representing somewhat less extraordinary conditions inherit from java.lang.Exception. Both Error and Exception are subclasses of Throwable. If you define your own exception classes (you are encouraged to), they should generally be subclasses of java.lang.Exception.

Exceptions can be placed in two broad categories: unchecked (those that inherit from Error or from RuntimeException, a subclass of Exception) and checked (everything else.)

A method must either catch all checked exceptions thrown by other methods it calls, or declare that it itself throws those same exception types. The compiler enforces this also. It's actually fairly helpful in this regard, because it tells you exactly which exception types you must deal with. Not catching an exception is precisely the same as throwing one. This is why I previously told you to declare that your main() routine throws IOException; having declared that it threw IOException, main() didn't need to bother to catch it when any other functions threw it. Almost all of the methods in the java.io package can throw IOExceptions, as you can see in the API documentation.

You are not required (by the compiler, anyway) to declare or to catch unchecked exceptions. This is primarily for pragmatic reasons, since the JVM itself can throw many kinds of unchecked exceptions (ArrayIndexOutOfBoundsException, ClassCastException, NullPointerException, OutOfMemoryError) during the execution of ordinary code. Some unchecked exceptions should nonetheless always be caught and handled; NumberFormatException is one example.

There is one more interesting wrinkle: the *finally* clause. A set of try and catch blocks can be followed by a finally block:

```
SomeResource sr = new SomeResource();
try
    {
       sr.manipulate();
```

```
    }
catch (Failure f)
    {
       System.out.println("Failed!");
       return;
    }
finally
    {
       sr.dispose();
    }
System.out.println("Done!");
```

The code in the finally block is executed after one of the try or catch blocks runs to completion (only one may), *even* if *that block returns from the method!* In the above example, even though the catch block returns, the sr.dispose( ) call would be made. Note that only if the try block runs to completion will the message "Done!" be printed. finally clauses let you put cleanup code that must be executed regardless of success or failure in one place, instead of duplicating it in each try and catch block.