An Exceptional Class

by
Peter Lavin

June 1, 2004

Overview

When a method throws an exception, Java requires that it be caught. Some exceptions require action on the programmer’s part and others simply need to be reported to the user. The Java class that we will develop in this article is concerned with the latter type of exception.

Instead of rewriting code every time you need to catch an exception, you can create a class to do most of the work for you.

Introduction

In a previous article, *Java Help Files*, we discussed how to create a class for displaying help files. Here we will take the same approach to handling exceptions. In some cases, exceptions can be handled quietly in the background without notifying the end user. However, we are concerned with the situations where it makes sense to display error messages.

The code to do this is not very complicated but gets a bit more interesting when we try to format the way our error message is displayed. We will show a few different approaches to doing this. We might have some fun along the way – taking different methods out for a spin and seeing how they drive.

The Code

Briefly described, this class uses a JOptionPane to display a dialogue box showing the exception type in the title bar and the error message in the body. Error messages created by the Java API are fairly uniform but there are many occasions when you will use classes created by other programmers – database drivers for instance. Here you won't be sure how or if the messages have been formatted. Error strings can end up being very long and may require the insertion of newline characters in order to display properly. We will develop three different methods of doing this, but for the moment have a quick look at the code below.

```java
//////////////////////////////////////////////////////////
//ErrorDialogue class
//////////////////////////////////////////////////////////
//comments for javadoc below
/** class ErrorDialogue
* This class is for use in catch clauses. It will display
* a dialogue box showing the error message.
*/
//put imports here
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class ErrorDialogue{
//data members
private Component window;
private final int increment = 30;
//////////////////////////////////////////////////////////
//constructors
//////////////////////////////////////////////////////////
public ErrorDialogue(Exception e, Component window) {
    this.window = window;
    doDialogue(e);
}
//end constructors
/////////////////////////////////////////////////////////////
//private functions
/////////////////////////////////////////////////////////////
private void doDialogue(Exception e) {
    Class error = e.getClass();
    String errname = error.getName();
    String message=e.getMessage();
    String messagetwo = e.getMessage();
    //check if contains newline
    if (e.getMessage().indexOf('\n') == -1) {
        //find length
        int length = message.length();
        //break at intervals
        if (length > increment){
            message=quickAndDirty(message);
            messagetwo=insertNewline(messagetwo);
        }
    }
    JOptionPane.showMessageDialog(window, "Error - " + message,
        errname, JOptionPane.WARNING_MESSAGE);
    JOptionPane.showMessageDialog(window, "Error - " + messagetwo,
        errname, JOptionPane.ERROR_MESSAGE);


}
/////////////////////////////////////////////////////////////
/**
* First method of parsing the string
*/
private String quickAndDirty(String message){
    //find space closest to midpoint
    StringBuffer sb= new StringBuffer(message);
    Character space=new Character(' ');
    int strlength = message.length();
    int midpoint = strlength/2;
    for(int x= midpoint; x < strlength; x++){
        if(new Character(sb.charAt(x)).equals(space)){
            sb.insert(x,"\n");
            break;
        }
    }
    String newstring = new String(sb);
    return newstring;
}
/////////////////////////////////////////////////////////////
/**
* second method of parsing the string
*/
private String insertNewline(String message){
    String tail = "";
    String head = "";
    int newstart=29;
    int breakpoint=0;
    tail = message.substring(newstart);
    int length=message.length();
    head = message.substring(0,newstart);
    while(length>increment && tail.indexOf(" ")!=-1){
        //find next space, insert break and concatenate
        breakpoint = tail.indexOf(" ")+1;
        head += tail.substring(0,breakpoint);
        head += "\n";
        tail=tail.substring(breakpoint);
        length=tail.length();
        if (length > increment && tail.indexOf(" ",newstart)!=-1){
            head+=tail.substring(0,newstart);
            tail=tail.substring(newstart);
        }else{
            head+=tail;
            message = head;
            break;
        }
    }
    return message;
}
////////////////////////////////////////////////////////////
/**
* Third method of parsing the string
*/
private String insertNewlineToken(String message){
    StringTokenizer stk = new StringTokenizer(message, " ", true);
    String temp = "";
    String newstring = "";
    int maxlength = increment;
    while(stk.hasMoreTokens()){
        temp = stk.nextToken();
        newstring += temp;
        //add newline if longer and don't start with a space
        if (newstring.length() > maxlength && temp.equals(" ")){
            newstring += "\n";
            maxlength = newstring.length() + increment;
        }
    }
    return newstring;
}
public static void main (String [] args){
    String message = "Let's have a really long message here";
    message+=" so that we can test our class. ";
    message+="Blah blah blah blah ...";
    Exception e = new Exception(message);
    new ErrorDialogue(e, null);
    System.exit(0);
}
}//end class
```

General Comments

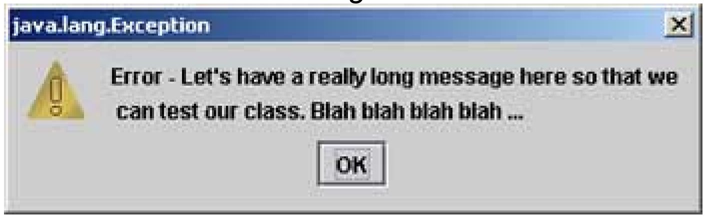
The constructor's first argument (line 21) is an object of the Exception class so this class or any of its subclasses may be passed in. It won't matter whether you pass in an IOException, an SQLException or any other subtype. The second argument is a Component that will act as the parent window for our dialogue box. Again we have chosen Component because it is the parent object of the various window subclasses.

On line 22 the constructor invokes a method, "doDialogue", to extract the error message and display it using a JOptionPane. The interesting part of this method is the call to a function to insert a newline character but let's first look at the output.

Output

Compiling and running the application results in the following output:

Image 1

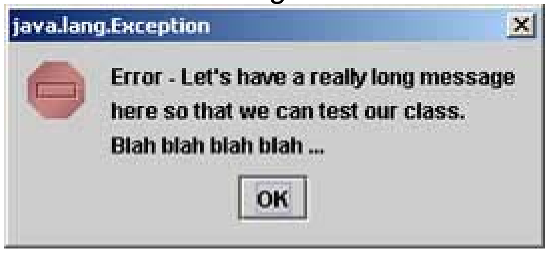


java.lang.Exception

Error - Let's have a really long message here so that we
can test our class. Blah blah blah blah ...

followed by:

Image 2



java.lang.Exception

Error - Let's have a really long message
here so that we can test our class.
Blah blah blah blah ...

This is the same message formatted in two different ways. Now that we've seen the output let's have a look at how it's done.

Parsing the String

First Method

The code to create our first dialogue box is found on lines 55-69. Because the String class does not have an “insertAt” method the error description is converted to a StringBuffer. The space character closest to the midpoint is found and a newline is inserted at that point. Basically the string is cut in half. Have a look at Image 1 above to see what it looks like.

This method is easy to understand and easy to code; hence the method name. Calling it “quickAndDirty” doesn’t quite do justice to this approach; it might well be suitable in other circumstances. Here though, any string of a few hundred characters would create problems. Pretty soon we’d have a dialogue box that stretched across or beyond the screen width.

Second Method

The shortcomings of the first method spawned method number two. The pseudo code for this method could be briefly written as:

```
get first portion of the string
get tail end
while not end of string
      find next most proximate space in tail end
      create substring of text up to this point
      concatenate this new string with original portion
      add on a newline
end while
```

A reasonable enough starting point but turning this into suitable code didn’t prove easy. Using almost twice as many lines of code as the “quickAndDirty” method, we end up with the dialogue box shown as Image 2.

This method does exactly what is wanted but the code doesn’t have the clarity of the first. This is important not only for writing the code but for maintaining it. This code works – but like the driver who refuses to ask for directions, we came the long way round. We grimly stayed behind the wheel and finally got there. There must be a better way.

The Better Way

This time, before starting out, we pulled over to the side of the road, got out the Java API and checked StringTokenizer for directions.

The first thing to notice is the StringTokenizer constructor that we chose to use (line 105).; it takes three arguments, the String to be tokenized, the character to use as a delimiter and finally a boolean that determines whether or not to return the delimiter as a token. This is ideal because we are not discarding the spaces between words.

The “hasMoreTokens” method is used to control our “while” loop (line 109). Inside the loop we simply reconstruct our string including spaces and add a newline when it exceeds the recommended length.

What could be simpler? Using one more line than the “quickAndDirty” method we have created more functional code without sacrificing clarity.

Larger Issues

At this point it is fairly obvious what the final form of our code should be. Drop the first two methods of parsing text, make a slight change to the “doDialogue” method and call only one method and finally, remove “main”. However, there is a larger lesson to be learned here.

Java is an object-oriented language with a large API. This means that much of the work you need to do may already have been done. Look around and take advantage of existing classes. In this case we found a class ideally suited to our needs – StringTokenizer – that performed the job in many fewer lines and made for more intelligible code. Programming with a high level language such as Java is more than just formulating your algorithm and implementing it. While nobody can know all of the classes available and all of their methods, a general knowledge of the language’s capability can be enormously helpful and a real timesaver. Do this and you won’t re-invent classes that already exist.

While you *can* create functional code without asking for directions, don’t. Roll down the window and ask ‘cause you’ll get there faster and more easily.

About the Author

Peter Lavin runs a Web Design/Development firm in Toronto, Canada. For more information visit http://www.softcoded.com. He may be reached at peterlavin@sympatico.ca.