Exception Handling

Consider the following class:

public class ErrorTest {

// return the average of the numbers (represented as strings) in the array

public static int average(String[] values) {

int total = 0;

for (int i = 0; i < values.length; i++) {

total += Integer.parseInt(values[i]);

}

return total / args.length;

}

public static void main(String[] args) {

System.out.println(average(args));

}

}

Since we have a main method, this program can be run stand alone, and it averages (using int average) the command line arguments

java ErrorTest 4 5 6

> 5

What can go wrong?

1. If the user does not enter an int (for example 4.5 or apple). In this case, the program will generate a NumberFormatException.

2. If the user does not enter anything. Then args.length is a 0, and the program will generate an ArithmeticException.

How to deal with errors:

1. Print an error message. This solution should only be used in routines that are directly interacting with the user. Otherwise the error message will be ignored.

2. Return a special "error" value. This solution should only be used when either the error value makes sense or there is no possibility that the error value could be confused for legitimate output.

3. Use a separate channel to send an error indication.

For this technique, Java uses exceptions.

We have already seen several types of exceptions:

NumberFormatException, ArithmeticException, NullPointerException, IndexOutOfBoundsException

Each of these exceptions are called unchecked exceptions. All unchecked exceptions are subclasses of either Error or RunTimeException.

With unchecked exceptions, the programmer does not have to explicitly state what to do if an exception occurs (is thrown).

The default is to stop the method and throw the exception on the the calling method.

The other types of exceptions (for example IOException), are checked exceptions. With checked exceptions, the programmer must specify what to do if the exception is thrown.

It is a compiler error to use code that can generate a checked exception but not explicitly state how to deal with it.

Key point: exceptions are just objects and they otherwise behave exactly like all other objects in Java.

Exceptions are not errors. Throwing an exception is just another way (besides the return statement) of getting data out of a method.

How to Handle an Exception

There are two things the programmer can do for an exception: handle the exception or throw the exception on to the calling method.

To throw the exception, we can do either or both of these:

1. Place "throws ExceptionType" in the header of the method. See the lab example with IOException. This must be done if throwing a checked exception.

2. Explicitly throw the exception with the throw statement.

throw e;

throw new ExceptionType();

To handle the exception, we use the try/catch statement.

try {

- code that could throw an exception

}

catch (ExceptionType e) {

- code that is executed if an exception of type ExceptionType occurs inside the try block

- e is a variable that stores the exception object address, and it exists inside this block

}

finally {

- code that is always executed upon exit of the try and catch blocks

}

There may be 0 or more catch blocks with a try and at most one finally block. There must be at least one catch block or a finally block with the try statement.

Note that the catch statement is a variable declaration. The variable e will hold the address of the exception that was thrown in the try block code.

Example: Dealing with the integer divide by 0:

The divide by 0 occurs in the line:

return total / values.length;

We can place a try/catch block around it and catch ArithmeticExceptions

try {

return total / values.length;

}

catch (ArithmeticException e) {

return 0;

}

Example 2: Dealing with the NumberFormatException

If the input contains a String that does not represent a number, a NumberFormatException occurs.

We catch the exception and try formatting the input as a double. (Note we could have just formatted it as a double to begin with, but then we would not have as much fun dealing with errors.)

Then, we need a second try/catch block inside the catch block in case the number is also not a double.

We can use more than one catch statement with a try:

try {

}

catch (ExceptionType1 e) {

}

catch (ExceptionType2 e) {

}

On an exception, Java will run through each type from the top to the bottom and stop at the first one that matches.

It is important that if one of the exception types is the parent of the other, the child type must come first.

Otherwise both parent and child types will match the first parent type declaration, and the catch block of the second child type declaration will not be executed.

Another important point is that this will catch multiple exceptions that occur in the try block. An exception that occurs inside the first catch block is not caught by the second catch.

Anything caught must be inside a try, and so to catch the error that occured inside the catch, we needed to nest another try/catch inside the catch block.

Example 3: A suggestion from class

One student wanted to have the NumberFormatException both ignore the non-number as well as give an error message.

Placing an error message inside the average method is not a good solution because it violates the rule above that only routines that get user input should print output to the user.

Our solution is to create and throw our own exception type: NumberFormatResultException

This new exception will store the result of an expression.

Code that handles this exception can both print an error message and get the result of the computation from the exception object. Please see the code example.

Because we made the NumberFormatResultException extend Exception, it is a checked exception, and we had to add "throws NumberFormatResultException" to the header of our average method.

We also could have made it extend NumberFormatException, and that would make it an unchecked exception.

In summary: exceptions provide two big benefits. They report errors on a separate channel from the normal return value of the method, and they let you deal

with the error where it makes the most sense, not necessarily where the error occurs.

Exceptions have one huge drawback. Throwing exceptions is like doing a "break" in a loop. It is another way that execution can jump out of a method. As a result, it makes reasoning logically about our code a little more challenging.