

Introduction to Computation and Problem Solving

Class 25: Error Handling in Java

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Goals

In this session we are going to explore better and worse ways to handle errors.

In particular, we are going to learn about the *exception mechanism* in Java.

After learning about exceptions, you are going to rework a specialized editor application to make the error handling more user friendly and effective.

3 Approaches to Error Handling

- The error code
- The error flag in memory, cf. UNIX `errno`
- Exceptions – the ejector seat

3

Using Error Codes

```
public static double average( double [] dArray )
{
    if ( dArray.length == 0 )
        return ???;

    double sum = 0.0;
    for ( int i = 0; i < dArray.length; i++ )
        sum += dArray[ i ];
    return sum / dArray.length;
}
```

4

Using Error Codes, 2

- You can return `Double.NaN`.

- You must test it as follows:

```
double[] darray = ...;
double d = average( darray );
if ( Double.isNaN( d ) )
    // error
else
    // OK
```

5

Using an Error Flag

```
public class MyMath
{
    static public final int ILLEGAL_ARGUMENT = -1;
    static public int errflag = 0;

    public static double average( double [] dArray )
    {
        if ( dArray.length == 0 ) {
            errflag = ILLEGAL_ARGUMENT;
            return ???;
        }

        double sum = 0.0;
        for ( int i = 0; i < dArray.length; i++ )
            sum += dArray[ i ];
        return sum / dArray.length;
    }
}
```

6

Using Exceptions

Advantages:

- Can not be confused with a normal return.
- Programmer has to work to ignore it.
- Error can be dealt with anywhere (up the call stack).
- Error conditions can be intelligently grouped using a variation of inheritance.

Now for the mechanism:

7

Throwing an Exception

```
public static double average( double [] dArray )
    throws IllegalArgumentException
{
    if ( dArray.length == 0 )
        throw new IllegalArgumentException();

    double sum = 0.0;
    for ( int i = 0; i < dArray.length; i++ )
        sum += dArray[ i ];
    return sum / dArray.length;
}
```

8

Declaring an Exception

- If a method can throw an exception, you can always declare the type of the exception in the header after the keyword `throws`

```
public static double average( double [] dArray )
    throws IllegalArgumentException
```
- The compiler requires you to declare the possible exception throw if the exception class is not derived from `RuntimeException` or `Error`. These are called *checked exceptions* (checked by the compiler).
- Exceptions derived from `RuntimeException` or `Error` are called *unchecked exceptions* and their declaration is optional.
- `IllegalArgumentException` is actually unchecked.

9

Checked vs. Unchecked Exceptions

- The distinction between checked and unchecked exceptions is fairly arbitrary.
- In principle, checked exceptions are those which you, the programmer, are supposed to be able to fix at runtime, such as a `FileNotFoundException`. Very often these are generated in system code by user, not programmer, error.
- Unchecked exceptions are supposed to be able to occur anywhere (so hard to check for) and to be the result of programmer error (so the best way of handling them is to fix the program). Good examples, `NullPointerException`, `ArrayIndexOutOfBoundsException`. Bad example, `NumberFormatException`, thrown e.g. by `Integer.parseInt(String s)`.

10

Creating Exception Instances

- Exceptions are objects. You must create a new instance of an exception before you can throw it

```
if ( dArray.length == 0 )
    throw new IllegalArgumentException();
```
- Exceptions can be arbitrarily complex, but the exceptions supplied with the JDK possess a default constructor and a constructor that takes a single error message string.

11

Catching Exceptions

- Normal execution of a method ceases at the point an exception is thrown.
- The runtime environment then looks for an enclosing `try` block with a matching `catch` clause.
- After executing the `catch` clause, the program resumes with the first statement after the `catch` block.

12

throw/try/catch Pattern

```
try {
    if ( error )
        throw new MyException();
    //skip further execution
}
catch ( MyException e ) {
    // handle exception e
}
catch( MyException2 e ) { ... } // optional
...
finally { // optional
    ... // always executed if present
}

//resume execution
```

13

Catching Exceptions, 2

- If there is no enclosing `try` block in the current method, or one with an appropriately typed `catch` clause, then the Java Virtual Machine goes hunting up the call stack for a matching `try/catch` pair.
- If the JVM can't find such a pair on the call stack, the default behavior is to print out an error message and halt the thread (the whole program if a console app; otherwise the program "hangs").

14

Catching Exception Up the Call Stack

```
double [] myDoubles = { ... };
double a = 0.0;
try {
    a = average( myDoubles );
}
catch ( IllegalArgumentException e )
{
    // do something about it
}
System.out.println( "Average = " + a );
```

15

Not Catching Exception Up the Call Stack

```
import javax.swing.*;

public class BadArgument {
    public static void main( String [] args ) {
        while ( true ) {
            String answer = JOptionPane.showInputDialog(
                "Enter an integer" );
            int intAnswer = Integer.parseInt( answer );
            // What happens if the user types %!Z$
            if ( intAnswer == 42 )
                break;
        }
        System.exit( 0 );
    }
}
```

16

Better BadArgument Implementation

```
public class BadArgument {
    public static void main( String [] args ) {
        while ( true ) {
            String answer = JOptionPane.showInputDialog(
                "Enter an integer" );
            int intAnswer;
            try {
                intAnswer = Integer.parseInt( answer );
            }
            catch ( NumberFormatException e ) {
                JOptionPane.showMessageDialog( null, "Not an integer" );
            }
            if ( intAnswer == 42 ) break;
        }
        System.exit( 0 );
    }
}
```

17

Writing Your Own Exception Classes

- Writing your own exception class is simple.
- New exception classes allow you to handle a new type of error separately.
- Exception classes extend `java.lang.Exception`.

```
public class DataFormatException
    extends java.lang.Exception {
    public DataFormatException()
    { super(); }
    public DataFormatException(String s)
    { super( s ); }
}
```

18

What an Exception Object Contains

- All Exception objects implement a `toString()` method that will print out the exception type and the exception message, if any.
- You can also retrieve the error message (the `String` argument, if present) by using the method:

```
public String getMessage()
```

19

Exceptions and Error Handling

- The Zen of error handling starts with the realization that the place you discover an error is almost never the place where you can fix it.
- Older languages like C use the horrible kludge of error codes and flags as we have seen.
- C++ introduced exceptions into the C family.
- But many programmers simply don't test for errors.

20

The Exception Strategy

The goal of using exceptions is to split the problem of *detecting* errors from the problem of *handling* them.

21

Exceptions in the Calculator

```
From CalculatorController:doOp()  
try { ...  
    switch( eT )  
    {  
        case Calculator.I_DIV:  
            setAcc( model.div() );  
            break;  
        ...  
    }  
    catch ( EmptyStackException e )  
    { error(); }  
}
```

22

Exceptions in the Calculator, 2

Where does the exception get thrown?
Not in CalculatorModel.

```
public double div()
    throws EmptyStackException
{
    double bot = stack.pop();
    double top = stack.pop();
    return top / bot;
}
```

23

Exceptions in the Calculator, 3

CalculatorController:
public double doOp()
does and puts FSM in
ERROR state

CalculatorModel:
public double div()
doesn't catch it!

ArrayStack<E>:
public E pop()
throws
EmptyStackException

24

Exceptions in the Calculator, 4

- Despite the fact that `EmptyStackException` is an unchecked exception, it is caused here by user error (not entering enough operands) not programmer error.
- `ArrayStack` doesn't know how to fix it so it throws it.
- `CalculatorModel` doesn't know how to fix it, so it throws it.
- Finally, `CalculatorController` can do something about it so it catches the exception and puts the whole calculator into the `Error` state until the user clears it.

25

Exceptions and Inheritance

- Since exceptions are instances of classes, exception classes may use inheritance. A `FileNotFoundException` is a derived class of `IOException`.
- When an error is detected, you should create and throw a new instance of an appropriate type of exception.
- The stack is searched for the nearest catch statement matching the exception class or one of its **superclasses**.

26

Exception Inheritance Example

```
try
{
    FileReader in = new FileReader( "MyFile.txt" );
    // read file here
}
catch ( FileNotFoundException e )
{
    // handle not finding the file (bad file name?)
}
catch ( IOException e )
{
    // handle any other read error
}
```

27

BirthdayApp Exercise, 1

- Download Lecture25.zip from the class web site.
- These files implement a birthday list editor.
- Examine the classes in the 3 files:
 - BirthdayApp: This is the application class with the main() method.
 - BirthdayView: This is the combined view and controller class.
 - BirthdayModel: This class maintains the growing list of birthdays. Don't worry about the methods labelled "Related to MVC" towards the end of the file. They update the JList when a new birthday is added.
 - Birthday: contains the name and date information.

28

BirthdayApp Exercise, 2

- Compile these classes and run the main method of `BirthdayApp`. This will open a window on the screen that contains a single birthday and a button.
- Click the button. This will prompt you to enter a name. Enter your name. Next, it will prompt you to enter the date of your birthday, enter the following exactly:
`January 1, 2002`
- You should see that your name and January 1, 2002 have been added to the list of Birthdays. Click the button again. This time, enter a name and the following exactly:
`january 1, 2002`
- You should see that the program has exited completely. Why did this happen? Examine the `addBirthday()` method in the `BirthdayModel` class closely. You should see that if the `addBirthday()` method has any trouble parsing the `String bdayDate`, it invokes `System.exit()`. Your first goal in this active learning session is to modify `addBirthday()` so that it does not call `System.exit()`. Instead, it should throw an exception when it cannot correctly parse the `String bdayDate`.

29

BirthdayApp Exercise, 3

- Your second goal in this active learning session is to modify the `actionPerformed()` method in the `BirthdayView` class so that it can react to exceptions thrown by `addBirthday()`.
- When you are completely done with the program, your final solution should take input from the user in exactly the same way. However, when the user enters something wrong, the program should not quit. Instead, it should tell the user that they did something wrong.
- Make your error handling as useful as possible. Try and let the user know not only that an error has occurred but give him or her a chance to fix it.

30