Java

Exceptions

Presented by Bartosz Sakowicz

Handling errors with exceptions

Golden rule of programming:

Errors occur in software programs.

What really matters is what happens **after** the error occurs:

- How is the error handled?
- Who handles it?
- Can the program recover, or should it just die?

Exceptions - fundamentals

• An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

• **Many kinds of errors can cause exceptions**--problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element.

• When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system.

• The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called **throwing an exception.**

Exceptions – fund...(2)

• After a method throws an exception, the runtime system leaps into action to find someone to handle the exception. The set of possible "someones" to handle the exception is the set of methods in the call stack of the method where the error occurred. The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate **exception handler.**

• An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. The exception handler chosen is said to **catch the exception**.

• If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

Exceptions advantages

1. Separating error handling code from „regular” code.

Suppose that you have a function that reads an entire file into memory. In pseudo-code, your function might look something like this:

readFile { open the file; determine its size; allocate that much memory; read the file into memory; close the file; }

At first glance this function seems simple enough, but it ignores all of these **potential errors**:

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Exceptions advantages(2)

Your function would looking something like this:

errorCodeType **readFile {**

initialize errorCode = 0;

open the file; if (theFileIsOpen) {

determine the length of the file; if (gotTheFileLength) {

allocate that much memory; if (gotEnoughMemory) {

read the file into memory;

if (readFailed) { errorCode = -1; } } else { errorCode = -2; } } else { errorCode = -3; }

close the file;

if (theFileDidntClose && errorCode == 0) { errorCode = -4; }

else { errorCode = errorCode and -4; } }

else { errorCode = -5; } return errorCode; **}**

Exceptions advantages(3)

If your read_file function used exceptions instead of traditional error management techniques, it would look something like this:

readFile {

try {

open the file; determine its size; allocate that much memory; read the file into memory; close the file;

} catch (fileOpenFailed) { doSomething;

} catch (sizeDeterminationFailed) { doSomething;

} catch (memoryAllocationFailed) { doSomething;

} catch (readFailed) { doSomething;

} catch (fileCloseFailed) { doSomething; }

}

The bloat factor for error management code is about 250 percent--compared to 400 percent in the previous example.

Exceptions advantages(4)

2. Propagating errors up the call stack

Lets suppose than only *method1* is interested in errors occured during reading the file:

method1 { call method2; }

method2 { call method3; }

method3 { call readFile; }

Traditional error notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1- the only method that is interested in them.

Exceptions advantages(5)

method1 {

errorCodeType error; error = call method2;

if (error) doErrorProcessing; else proceed; }

errorCodeType method2 {

errorCodeType error; error = call method3;

if (error) return error; else proceed; }

errorCodeType method3 {

errorCodeType error; error = call readFile;

if (error) return error; else proceed; }

Exceptions advantages(6)

The Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception. Thus only the methods that care about errors have to worry about detecting errors.

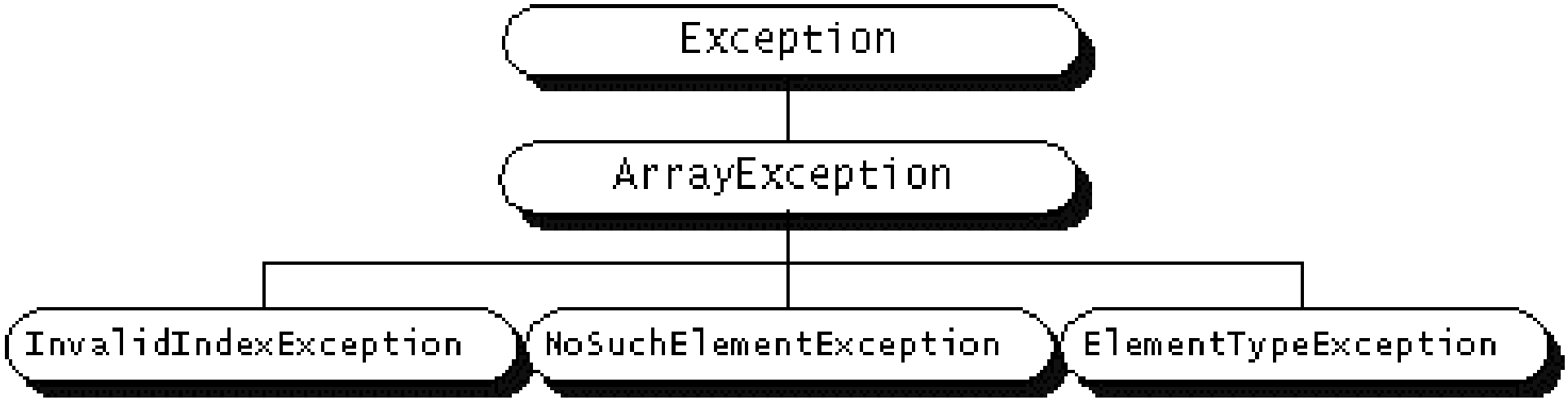
```
method1 {
try {
        call method2;
         } catch (exception) { doErrorProcessing; }
}
method2 throws exception {
call method3; }
method3 throws exception {
call readFile; }
```

Exceptions advantages(7)

3. Grouping Error Types and Error Differentiation.

Lets imagine a group of exceptions, each of which represents a specific type of error that can occur when manipulating an array: the index is out of range for the size of the array, the element being inserted into the array is of the wrong type, or the element being searched for is not in the array.

For example ArrayException is a subclass of Exception (a subclass of Throwable) and has three subclasses:



Exceptions advantages(8)

An exception handler that handles only invalid index exceptions has a catch statement like this:

catch (InvalidIndexException e) { . . . }

To catch all array exceptions regardless of their specific type, an exception handler would specify an ArrayException argument:

catch (ArrayException e) { . . . }

It is possiblel to set up an exception handler that handles any Exception with this handler:

catch (Exception e) { . . . }

It is not recommended writing general exception handlers as a rule.

Understanding exceptions

Let's look at following code:

public class InputFile {

private FileReader in;

public InputFile(String filename) {

in = new FileReader(filename); }

What should the FileReader do if the named file does not exist on the file system?

- Should the FileReader kill the program?
- Should it try an alternate filename?
- Should it just create a file of the indicated name?

There's no possible way the FileReader implementers could choose a solution that would suit every user of FileReader.

By throwing an exception, FileReader allows the calling method to handle the error in way is most appropriate for it.

Catch or specify requirement

Java requires that a method either catch or specify all checked exceptions that can be thrown within the scope of the method.

Catch - a method can catch an exception by providing an exception handler for that type of exception.

Specify - if a method chooses not to catch an exception, the method must specify that it can throw that exception.

Runtime and checked exceptions

Runtime exceptions are those exceptions that occur within the Java runtime system. This includes exceptions like:

•arithmetic exceptions (such as when dividing by zero)

•pointer exceptions (such as trying to access an object through a null reference)

• indexing exceptions (such as attempting to access an array element through an index that is too large or too small)

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous**.** Thus the compiler does not require that you catch or specify runtime exceptions, although you can.

Checked exceptions are exceptions that are not runtime exceptions and are checked by the compiler; the compiler checks that these exceptions are caught or specified.

The finally block

```
public void writeList() {
PrintWriter out = null;
try {
        out = new PrintWriter( new FileWriter("OutFile.txt"));
        for (int i = 0; i < size; i++)
        out.println("Value at: " + i + " = " + vector.elementAt(i));
} catch (ArrayIndexOutOfBoundsException e) {
System.err.println("Caught ArrayIndexOutOfBoundsException: " +
e.getMessage());
 } catch (IOException e) {
System.err.println("Caught IOException: " + e.getMessage());
 } finally {  // (3 possible versions of runtime – only one close)
if (out != null) out.close(); } } }
```

Specifying the Exceptions Thrown by a Method

Sometimes, it's appropriate for your code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception.

To specify that method throws exceptions, you add a **throws** clause to the method signature. The throws clause is composed of the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The throws clause goes after the method name and argument list and before the curly bracket that defines the scope of the method. **Example:**

public void writeList() throws IOException, ArrayIndexOutOfBoundsException {

Choosing the exception type to throw

You should write your own exception classes if you answer "yes" to any of the following questions. Otherwise, you can use java predefined exceptions:

- Do you need an exception type that isn't represented by those in the Java development environment?
- Would it help your users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will your users have access to those exceptions? A similar question is: Should your package be independent and self-contained?

Choosing the exception type to throw(2)

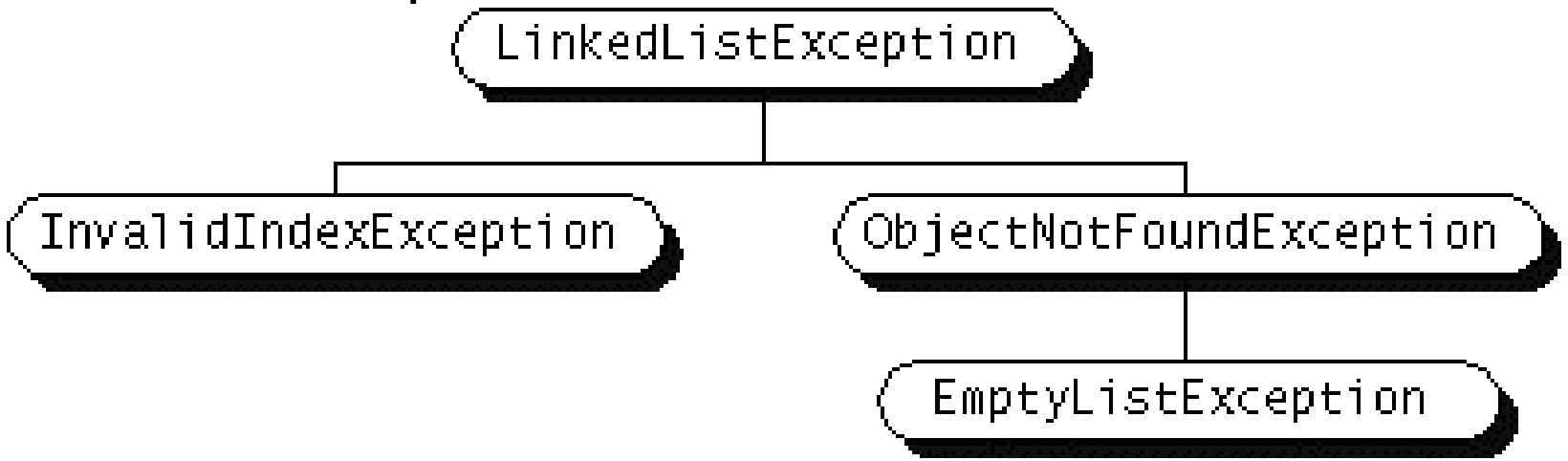
Suppose you are writing a linked list class that you're planning to distribute as freeware. Among other methods, your linked list class supports these methods:

objectAt(int *n*) -Returns the object in the nth position in the list.

firstObject - Returns the first object in the list.

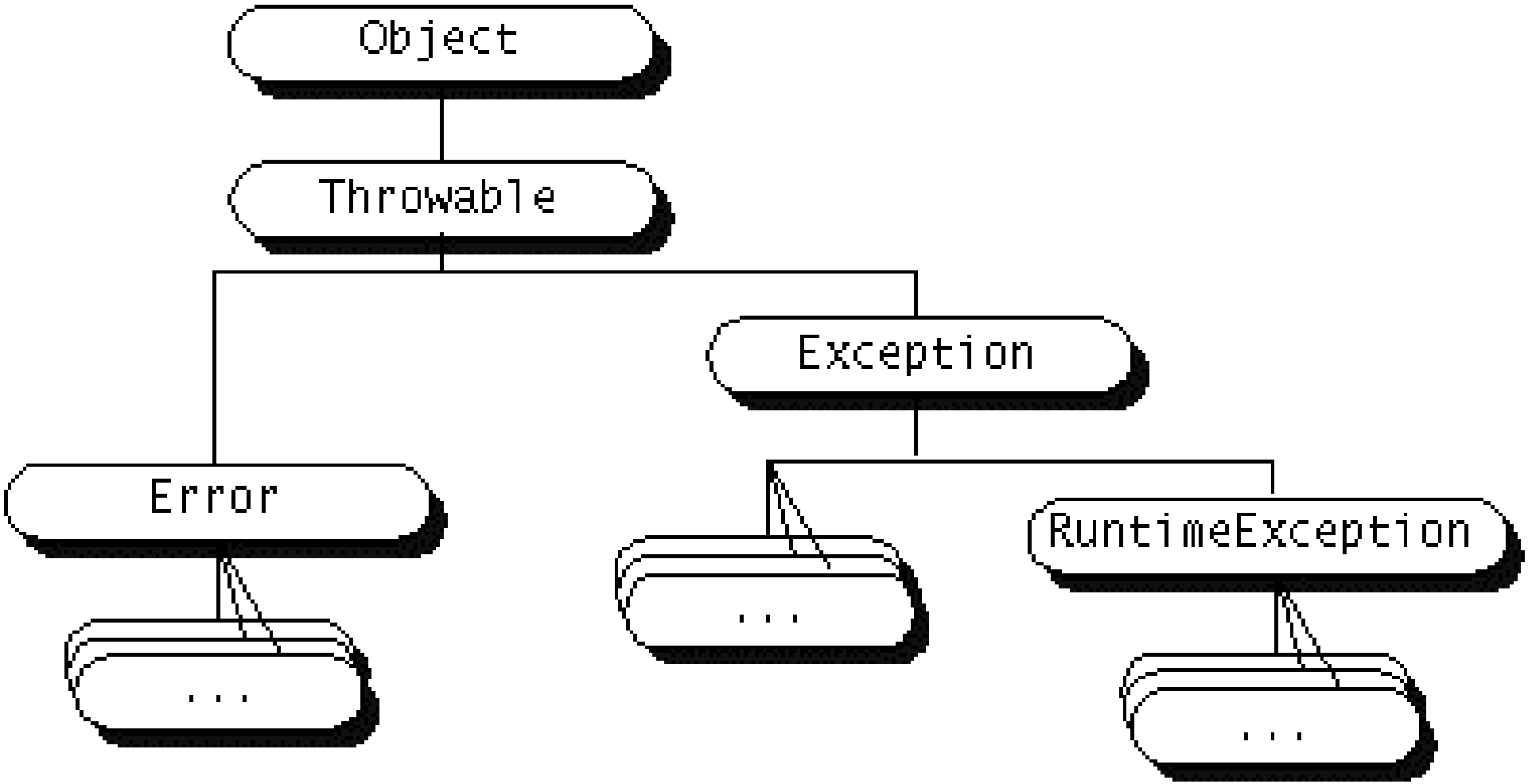
indexOf(Object *n*) - Searches the list for the specified Object and returns its position in the list.

Your structure of exceptions should be like this:



Choosing the superclass

Java exceptions must be Throwable objects (they must be instances of Throwable or a subclass of Throwable).



Choosing the superclass(2)

- **Errors** are reserved for serious hard errors that occur deep in the system.
- You shouldn't subclass **RuntimeException** unless your class really is a runtime exception.
- Generally you should subclass **Exception**.

Naming Conventions

It's good practice to append the word "Exception" to the end of all classes that inherit (directly or indirectly) from the Exception class. Similarly, classes that inherit from the Error class should end with the string "Error".

Choosing the superclass(3)

- A method can detect and throw a RuntimeException when it's encountered an error in the virtual machine runtime. However, it's typically easier to just let the virtual machine detect and throw it. Normally, the methods you write should throw Exceptions, not RuntimeException.
- Similarly, you create a subclass of RuntimeException when you are creating an error in the virtual machine runtime (which you probably aren't). Otherwise you should subclass Exception.
- Do not throw a runtime exception or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Implementing Exceptions

The simplest exception is:

```
class SimpleException extends Exception { }
```

But exceptions can be more complicated (and useful):

```
class MyException extends Exception {
private int i;
public MyException() { }
public MyException(String msg) {
        super(msg); }
public MyException(String msg, int x) {
        super(msg); i = x; }
public int val() { return i; } }
```

Implementing Exceptions(2)

Usage of MyException:

```
public class ExtraFeatures {
public static void f() throws MyException {
        System.out.println( "Throwing MyException from f()");
        throw new MyException(); }
public static void h() throws MyException {
        System.out.println( "Throwing MyException from h()");
        throw new MyException( "Originated in h()", 47); }
public static void main(String[] args) {
try { f(); } catch(MyException e) { e.printStackTrace(System.err); }
try { h(); } catch(MyException e) { e.printStackTrace(System.err);
System.err.println("e.val() = " + e.val());
} } }
```

Implementing Exceptions(3)

The output is:

Throwing MyException from f()

MyException

at ExtraFeatures.f(ExtraFeatures.java:22)

at ExtraFeatures.main(ExtraFeatures.java:34)

Throwing MyException from h()

MyException: ***Originated in h()***

at ExtraFeatures.h(ExtraFeatures.java:30)

at ExtraFeatures.main(ExtraFeatures.java:44)

e.val() = 47

System.out and System.err

- System.err is better place to send error information than System.out, which may be redirected.

- If you send output to System.err it will not be redirected along with System.out so the user is more likely to notice it.

Methods inherited from Throwable

String getMessage( )
String getLocalizedMessage( )

Gets the detail message, or a message adjusted for this particular locale.

String toString( )

Returns a short description of the Throwable, including the detail message if there is one.

Methods inherited from Throwable(2)

void printStackTrace( )

void printStackTrace(PrintStream)

void printStackTrace(PrintWriter)

Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown. The first version prints to standard error, the second and third prints to a stream of your choice

Throwable fillInStackTrace( )

Records information within this Throwable object about the current state of the stack frames. Useful when an application is rethrowing an error or exception.

Exceptions - questions

Is there anything wrong with this exception handler as written? Will this code compile?

...

} catch (Exception e) { ...

} catch (ArithmeticException a) { ... }

Answer

This first handler catches exceptions of type Exception; therefore, it catches any exception, including ArithmeticException. The second handler could never be reached. This code will not compile.

Exceptions – questions(2)

1. Match each situation in the first column with an item in the second column.

a. int[] A;
A[0] = 0;

b. The Java VM starts running your program, but the VM can't find the Java platform classes. (The Java platform classes reside in classes.zip or rt.jar.)

c. A program is reading a stream and reaches the end of stream marker.

d. Before closing the stream and after reaching the end of stream marker, a program tries to read the stream again.

1. error
2. checked exception
3. runtime exception
4. no exception

Answers: a-3 b-1 c-4 d-2

Presented by Bartosz Sakowicz
DMCS TUL Lodz