Exceptions

Murray Cole

Computer Science
School of Informatics
The University of Edinburgh

Stuff that goes wrong

Programming the **wrong thing**!

Programming the **thing wrong**, leading to

- **syntax errors** detected at **compile time**
- **logical errors** detected at **run time** through
  - unexpected/incorrect behaviour
  - program crash (extremely incorrect!)

The compiler forces us to fix all syntax errors. A well designed language can help us to find (sometimes) and handle (more often) some of our logical errors as well as various unpredictable situations.

Exceptions — reminders

We have already met the notion of *exception*. An exception (informally) is an event that occurs during the execution of a program, which disrupts the normal flow of instructions.

Many kinds of error 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.

Java provides language constructs for managing exceptions. These allows us to **create** exceptions, **throw** exceptions, and **catch** exceptions.

Error handling can lead to spaghetti code!

Suppose we are writing a function to read a file into memory.

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

This pseudo-code ignores many 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?

```
errorCodeType readFile(file) {
    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) {
            . . .
```

The perils of spaghetti code

The original 7 lines have been inflated to 29 lines of code!

There's so much error detection, reporting, and returning that the original 7 lines of code are lost in the clutter.

The logical flow of the code has also been lost in the clutter, making the program harder to read and understand.

Programmers in other languages often "solve" these problems by simply ignoring errors — errors are "reported" when their programs crash.

Error handling with exceptions

```
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;
    }
}
```

Exceptions in Java

Conceptually, an exception is a **description of problem**, or other **unusual situtation** which has arisen at run-time, and which the system is asking the program to handle.

If the program can't handle the exception, then the **system will crash the program** and report the execption and where it arose.

Formally, an exception is an **object** (like most things in Java!), with associated fields and methods which let us inspect and handle it.

An Uncaught Exception

```
[aubergine]mic: java Test
Exception in thread "main" java.lang.NullPointerException
        at Test.checkGrades(Test.java:10)
        at Test.analyseStudent(Test.java:5)
        at Test.main(Test.java:59)
```

Organization of Exceptions in Java

Java organizes the builtin exceptions in `java.lang`:

- All exceptions are subclasses of `Throwable`
- Exceptions are split into **errors**, subclasses of `Error`, and **exceptions**, subclasses of `Exception`.
- Exception has a special subclass `RuntimeException`.

Errors are usually fatal problems which cannot be rectified, (*e.g.* `LinkageError`, `VirtualMachineError`).

Exceptions are problems which can potentially be caught. The *runtime exceptions* are those problems which can be raised practically anywhere (*e.g.* `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `NullPointerException`).

Handling exceptions

Exceptions are handled with **try**, **catch**, **finally**. Notice that the **finally** block is *always* executed, however **try** exits: normally; with a caught/uncaught exception; or with **break**, **continue**, **return**.

Successive **catch** clauses are tested against an exception using the exception hiearchy.

```
try {
    // Some code that may raise exceptions
}
catch (SomeException e1) {
    // Process e1
}
catch (SomeOtherException e2) {
    // Process e2
}
finally {
    // Code that is always executed
}
```

What can you do with an Exception?

When you catch an exception object `e`, there are several things you can do with it:

1. Invoke one of the standard methods `e.getMessage()` or `e.printStackTrace()`

2. Print out some message, possibly with specific information pertaining to your special exception class. (You might add specific fields to your exception class to hold this information).

3. Take remedial action and continue the program.

Using try - catch

```
while (true) {
    try {
        Student s = (Student) in.readObject();
        System.out.println(s);
    } catch (EOFException e) {
        System.out.println("End of report");
        break;
    } catch (ClassCastException e) {
        System.out.println("Invalid record");
    }
}
```

Where is an exception caught?

Firstly, the **`catch`** clauses of the immediately enclosing **`try`** block are examined, in textual order, for a match (bearing in mind the exception hierarchy).

Then, any textually nested enclosing **`try`**-**`catch`** blocks are similarly investigated "inside out" by nesting order.

Lastly, any enclosing **`try`**-**`catch`** blocks on the call stack are investigated. A method which may throw an exception but doesn't catch it, has to say so with a **`throws`** clause.

The **`finally`** clause is executed as the last action of the block, either when the **`try`** completes normally, or after an exception has been caught locally, or before an exception is propagated further afield.

```
try {
    while (true) {
      try {
          Student s = (Student) in.readObject();
          System.out.println(s);
      } catch (ClassCastException e) {
          System.out.println("Invalid record");
      }
    }
} catch (EOFException e) {
    System.out.println("End of report");
}
```

Propagating Errors Up the Call Stack in Java

The Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception. A Java method can "duck" any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. 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;
}
```

Creating new exceptions

We can declare our own new exceptions simply by defining a class which **extends** one of the supplied base exception classes, usually the **Exception** class.

Here's an example which includes a constructor which is passed a string argument to create a customized error message:

```
public class WrongNumberException extends Exception {
    WrongNumberException(String message) {
        // Construct an exception message for
        // the toString method, using the given
        // message and the super-class constructor.
        super("Wrong number: " + message);
    }
}
```

Throwing exceptions

To throw an exception, we use Java's **throw** statement:

```
throw someThrowableObject;
```

The throwable object is an instance of any subclass of the `Throwable` class, typically freshly created with **new**. The constructor for the subclass may well include arguments (*e.g.* an error message).

```
if (args.length != 3)
  throw
   new WrongNumberException("3 command line args needed.");
```

Result:

```
Exception in thread "main" WrongNumberException:
     Wrong number: 3 command line args needed.
     at TestWrongNumberException.main(TestWrongNumberException.java:26)
```

Checked exceptions and the `throws` clause

Exceptions are split into **checked** and **unchecked** exceptions. `RuntimeExceptions` and errors are unchecked, all others are checked. Of course, **all** exceptions occur at run time, so the subclass name is very badly chosen!

The checked exceptions are ones which the Java compiler will force you either to handle with **`try-catch-finally`**, or to "duck" by specifying that your method **`throws`** that exception.

```
public String getWord() throws FileNotFoundException {
    ...
}
```

Notice that you may need to specify **`throws`** in your method even if you do not explictly **`throw`** the exception yourself. An exception must be specified whenever some method called within the scope of your method may throw that exception. Several exceptions (or their super-classes) may be listed.

The Output (figure 14.9 Deitel & Deitel)

```
Method throwException

Exception handled in method throwException

Finally executed in throwException

Exception handled in main

Method doesNotThrowException

Finally executed in doesNotThrowException

End of method doesNotThrowException
```