

Exceptions

Great Exceptions

Here we'll just cover the basic features and uses of exceptions.

Pre-Exceptions

A program has to encode two ideas -- how to do the main computation, and how to identify and handle exceptional (rare) failure conditions that might arise during the computation.

Error Code Checking

The old way to do this is with error codes...

```
int err1 = foo(a, &b);
if (err1 < 0) { ... error handling ... }
int err2 = bar(b, &c);
if (err2 M 0) { ... error handling ... }
```

Error Code Problems

1. The main computation and the error handling are mixed together -- less convenient, less readable
2. The error must be handled where it occurs, or we must manually pass back error codes of some sort. The error-code-return coding is a bore. Often, the error occurs at a low level, but we want to handle it at a higher level (a few methods up the call chain).
3. The error handling, such as it is, depends on programmer diligence -- the compiler is not enforcing anything. This is especially dangerous for code paths that, as a practical matter, may never execute until the rare error condition actually happens. (e.g. running out of disk space midway through save operation)

Exceptions

Exceptions try to formalize and separate the error handling code from the main code, and to do it in a standard, structured way.

Java Exception

At runtime, an "exception" represents an error condition that halts the current computation.

Many common exceptions are detected by the VM -- `NullPointerException`, `ArrayIndexOutOfBoundsException`...

```
String s = null;
is (s.length() > 0) { ... // throws NullPointerException
```

Exception classes

An Exception is a Java object used to represent the error condition.

Contains a "message" human readable string -- set in the ctor.

The Throwable class is the superclass of all exceptions

If you want a catch() clause (below) that catches **everything**, declare it to catch Throwable.

Error subclass

low-level errors, such as OutOfMemory error. Generally, code does not declare or deal with these. They are fatal.

Exception subclass

The commonly used exceptions are subclasses of Exception.

A method that throws an Exception must declare it with throws clause (below)

Also known as a "checked" exception

RuntimeException subclass

Subclass of Exception.

A method that throws a RuntimeException does not need to declare it

Also known as an "unchecked" exception

Exception vs. RuntimeException

If it makes sense for the caller to be aware of the exception and have a handling strategy, use Exception.

If the caller does not have any special handling for the exception, then use RuntimeException, and at runtime the exception will halt execution.

Exception throw

Throw an exception object at runtime.

throw <exception>

e.g. if (!(array.length >= 2)) throw new IllegalArgumentException("array length must be >= 2");

e.g. throw (new RuntimeException("oh no!"));

Throw course

The throw stops executing the method, exits the method, and its caller, and so on, releasing locks and dealing with catch() and finally() clauses on its way out (below).

If A calls B call C which throws an exception, we exit C, then B, then A -- this is known as a "stack unwind" in programming language circles.

Methods vs. Exceptions

If a method calls something that might throw an exception, then...

1. The method must "eat" the exception with a try/catch clause
2. -or- The method may declare the exception with a "throws" clause

Does it make sense to force the caller, at compile time, to acknowledge the possibility of the exception, or should it be handled/eaten within the method?

Method throws Example

Here the fileRead() method declares that it throws IOException.

Any code calling `fileRead()` will need to deal with the exception case -- more inconvenient, but ultimately more robust.

```
public void fileRead(String fname) throws IOException {           // NOTE throws
    // this is the standard way to read a text file...
    FileReader reader = new FileReader(new File(fname));
    BufferedReader in = new BufferedReader(reader);

    String line;
    while ((line = in.readLine()) != null) {
        ...
        // readLine() etc. can fail in various ways with
        // an IOException
    }
}
```

try / catch

A try/catch block attempts to execute a block of statements. If an exception occurs in some method called from within the try block, then the catch() clause can intercept the exception stack unwind.

Having intercepted the exception, the catch clause can handle it in any number of ways (see below).

There can be multiple catch clauses, in which case they are searched from top to bottom, and the first that matches the exception gets it.

If the catch clause matches the exception, it halts the stack unwind. Otherwise, the exception keeps going up the call stack. Catch Throwable to get all possible exceptions -- more likely, you catch a specific exception you are care about, and let the others pass on through.

try / catch example

Here, the `fileRead()` method uses a try/catch to catch the `IOException` internally and print an error message. Note that the `IOException` is caught inside `fileRead()`, so `IOException` is not declared in a "throws" in the method prototype.

```
public void fileRead(String fname) {           // NOTE no throws

    try {
        // this is the standard way to read a text file...
        FileReader reader = new FileReader(new File(fname));
        BufferedReader in = new BufferedReader(reader);

        String line;
        while ((line = in.readLine()) != null) {
            ...
            // readLine() etc. can fail in various ways with
            // an IOException        }
        }

        // Control jumps to the catch clause on an exception
        catch (IOException e) {
            // a simple handling strategy -- see below for better strategies
            e.printStackTrace();
        }
    }
}
```

```
}
}
```

Exception handling with catch()

What to do in the catch handler...

1. Eat silently

This is the worst strategy. It would only be appropriate if you are just writing junk code or a lecture example or something where the exception is not meant to be handled. It is polite to name the exception variable "ignored" to emphasize that it is not really being handled. This is never appropriate in production code. This strategy can make debugging more difficult, since runtime errors are eaten silently which is unfortunate since seeing the exception is often a big help with debugging.

```
try {
    // file operations
}
catch(IOException ignored) { } // silently eat the exception
```

2. e.printStackTrace()

Exceptions have a built in `printStackTrace()` behavior. If you are writing code without a clear exception plan, this is a good default strategy. It does not handle the exception meaningfully, but it does produce debugging output. If the code does find itself actually being used, the unhandled exception will announce itself if it ever happens.

```
try {
    // file operations
}
catch (IOException e) {
    e.printStackTrace();
    // we have not really handled the error, but at least we produce debugging output

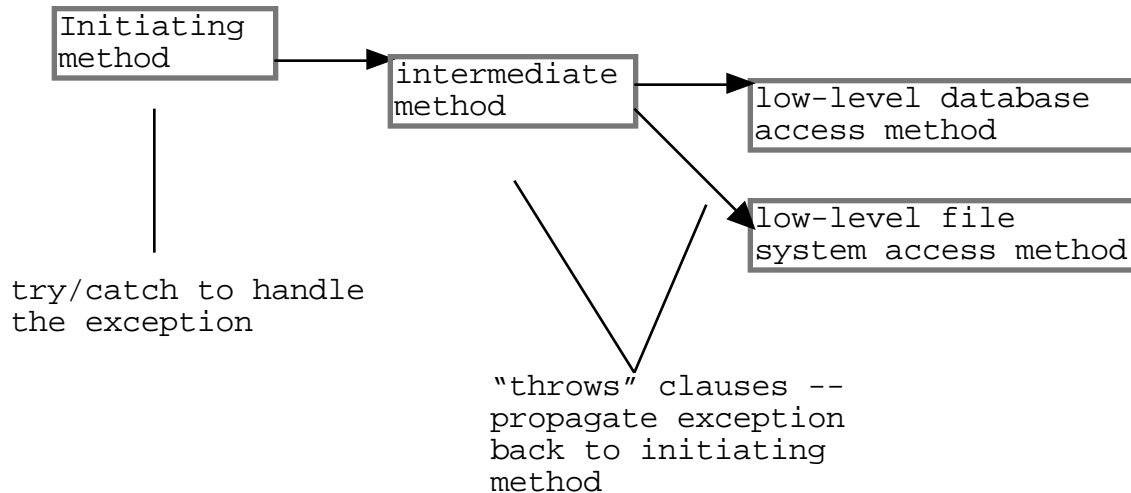
    // could allow execution to continue, or could System.exit(-1);
}
```

3. Pass it along

Rather than handle the exception internally, we might make a design decision that it is appropriate for the caller to decide how to handle the exception. In that case, add a "throws" clause to the method prototype (see above) If the exception is meaningful, hiding it from the client is probably a mistake. The client will need to declare their own try/catch logic to express how they would like to treat the exception condition.

Exception patterns

1. Inner throws, Outer handler



Suppose there is some outer GUI or controller code that initiates operations, and it calls lower level methods to do the computation.

1. Suppose the lower level methods can fail with `IOExceptions` and `DatabaseExceptions`.
2. The low-level and intermediate methods declare `IOException` and `DatabaseException` in their "throws" clauses. The exceptions are, realistically, something the caller needs to know about.
3. The outer, initiating method uses `try/catch` to bring up an error dialog or write an error log or whatever to deal with the error.

Swing Thread Example

The Swing thread is a simple example of an "outer" caller using `try/catch`. The swing thread pulls a task off its queue and executes it with a `try/catch` similar to the following.

In this way, if your button handler or whatever gets a `NullPointerException`, the Swing loop prints the exception, but does not let the exception mess up the swing thread itself. After the exception, the swing thread loops around to handle the next task.

```

loop processing Swing GUI tasks {
    Runnable task = nextTask();
    try {
        task.run();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
  
```

2. No: try/catch at every level

If every layer in the call chain has its own try/catch -- that's a bad sign. The low-level methods should just identify the error and pass back the information that it happened.

The more complex, try/catch code to do something about the error should be concentrated in one place.

A try/catch may be used at a lower level if the method deals with the error on its own, and the upper layers do not need to know about it.

3. Multiple Catch Clauses

It is possible to have multiple catch clauses to use different strategies for the different types of exceptions...

Here's an example of some SAX XML loading code which can fail with either an IOException or a SAXException...

```
private void loadXML(File file) {
    try {
        // file opening and XML parsing code
    }
    catch (SAXException e) {
        System.err.println("XML parse err:" + e.getMessage());
    }
    catch (IOException e) {
        System.err.println("IO err:" + e.getMessage());
    }
}
```

4. Clean try/catch

Ideally, the try/catch can be written so that the data is left in a good state regardless -- either the operation succeeds, or on the exception, we clean up the state so its still valid.

Wrong -- unclean

Suppose we have an HTTPTester class that makes tests many url connections, and saves the result strings in an array. Suppose there is a URL class that responds to a open() and getData() messages, but they may throw a URLException.

This code does not work quite right -- if the exception throws out of connect(), resultCount will already have been incremented. If it throws out of open(), it happens to work ok.

```
class HTTPTester {
    private String[] results;
    private int resultCount;
```

```

// Attempts a connection to the given url and adds the result to the array.
// Suppose that url responds to a connect() message
public void test(URL url) {

    try {
        url.connect();    // may throw
        resultCount++;
        results[resultCount-1] = url.getData();    // may throw
    }
    catch (ConnectException e) {
        // log the exception
    }
}

...
}

```

Correct -- fail safe

This version works correctly -- it tries the unsafe operations first. If one of them fails, the receiver is ok, since we have not yet adjusted any of its state. No cleanup is required. It's a little subtle -- a later code maintainer will need to realize that the order of operations is this way for a reason.

```

class HTTPTester {
    private String[] results;
    private int resultCount;

    // Attempts a connection to the given url and adds the result to the array.
    // Suppose that url responds to a connect() message
    public void test(URL url) {

        try {
            // do all the unsafe operations first, store results on the stack
            // not into ivars
            url.connect();    // may throw
            String result = url.getData();    // may throw

            // if we get here, store into the ivars
            resultCount++;
            results[resultCount-1] = result;
        }
        catch (ConnectException e) {
            // log the exception
        }
    }

    ...
}

```

Correct -- clean up

This version works correctly -- it includes code in the catch clause to detect and undo the incomplete operation. This style makes me a little nervous. The catch() clause has to deal with the object in different possible states of disrepair,

and testing all the cases is going to be very difficult. I prefer the above fail-safe strategy where possible.

```
class HTTPTester {
    private String[] results;
    private int resultCount;

    // Attempts a connection to the given url and adds the result to the array.
    public void test(URL url) {
        int oldCount = resultCount;
        try {
            url.connect();    // may throw
            resultCount++;
            results[resultCount-1] = url.getData();    // may throw
        }
        catch (ConnectException e) {
            // log the exception

            // specifically detect and undo the partial operation
            if (resultCount > oldCount) {
                results[resultCount-1] = null;
                resultCount = oldCount;
            }
        }
    }
    ...
}
```

Finally

The "finally" clause defines code that always executes when the block exits -- normally or due to an exception.

In the following example, the finally clause sets processing to false as the method exits -- either normally (exiting, calling return), or because of an exception.

Style: I have some reservations about the finally clause. I think code that executes for both the normal and exception cases is not what most programmers are going to expect, although it's nice to not have to duplicate code for the two cases.

```
public void processFile() {
    processing = true;
    try {
        ...
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        processing = false;
    }
}
```