



Contents

- [1 Why Exceptions?](#)
- [2 Don't Fight Exceptions](#)
- [3 When It's OK to Swallow](#)
- [4 Use Exceptions](#)
- [5 Where To Catch Exceptions](#)
- [6 Summary](#)

Why Exceptions?

Why does Java have "exceptions" and "try...catch" blocks? Plenty of other programming languages survive without them. C, for example.

C programs still have to deal with exceptions. At least, they have to deal with exceptional situations, such as a file or network connection failing to open. What C lacks, specifically, is *structured exception handling*.

You remember structured programming. Pascal, that stuff. Maybe some lecturer at university mentioned it. Maybe it sounded like something you do at university, and that real programmers don't do (kind of like having your hands in the "10 to 2" position when you drive a car). Maybe it sounded like something that has been superseded by object orientation.

It turns out that structured programming is actually really good. Object orientation doesn't replace it; it builds on top of it. Structured programming techniques are shown to reduce development times, reduce development costs, and increase readability and maintainability.

However... when it comes to exceptional situations, structured programming can really suck.

Structured programming, remember, means that we must stick to working in blocks, and cannot make arbitrary jumps from one place to another. We can't use "goto" (OK, it doesn't exist in Java, but it does in C and C++), or "break", or "continue", and we can't use "return" anywhere other than as the last line of the method.

In any piece of C code where you have to acquire multiple resources (allocating memory, opening file, connecting to databases, whatever), writing structured code usually means wrapping the useful bit of the code in a huge mess of nested "if" statements.

It's easy to end up with more code for handling failures than there is for handling success, and it can make the actual, useful code hard to spot.

C programmers therefore, sensibly, don't make their code unreadable with all this rubbish. They write unstructured code. However, one downside to this is that, often, resources are not cleaned up properly in exceptional situations, leading to problems such as memory leaks.

Structured exception handling (SEH) allows us to handle exceptions elegantly, in a relatively structured way, without having to write hideous, deeply nested code.

Don't Fight Exceptions

Many programmers see exceptions as "errors", something to be stamped out, killed, as quickly as possible. Since Java forces you to handle certain classes of exception, you quite often see code like this.

```
public boolean saveData(byte[] data) {
    try {
        RecordStore rs = RecordStore.openRecordStore(RS_NAME, false);
        rs.addRecord(data, 0, data.length);
        rs.closeRecordStore();
    } catch (RecordStoreException e) {
        return false;
    }
    return true;
}
```

Great, the exception is dealt with! Oh, but now we don't know if the record store couldn't be opened, or if the addition of the record failed. And if the addition of the record failed, then we left the record store open. OK...

```
public boolean saveData(byte[] data) {
    RecordStore rs = null;
    try {
        rs = RecordStore.openRecordStore(RS_NAME, false);
        rs.addRecord(data, 0, data.length);
    } catch (RecordStoreException e) {
        return false;
    } finally {
        if (rs != null) {
            try {
                rs.closeRecordStore();
            } catch (RecordStoreException e) {
                // ignore
            }
        }
    }
    return true;
}
```

Great. What a hideous mess.

Also, the code that calls this method is left with handling a true or false (success or fail) return value, meaning it has to wrap itself in "if" statements to handle any problems (or has to be unstructured). And so, we're back to The Time Before SEH, only now with even more confusing code.

RULE 1: Exceptions are not "errors". They are signals that something went wrong. Signals that can be used. Catch them where you can use them the best.

Instead of returning a boolean, we can just allow the exception to signal the caller that something went wrong.

```
public void saveData(byte[] data) throws RecordStoreException {
    RecordStore rs = RecordStore.openRecordStore(RS_NAME, false);
}
```

Exception_Handling_in_Java

```
try {
    rs.addRecord(data, 0, data.length);
} finally {
    rs.closeRecordStore();
}
}
```

This code is now simpler and more readable. It's safe, and always cleans up after itself. Instead of returning a boolean, it either throws an exception, or it doesn't.

The caller can use SEH to handle any problems from this, if it wants to. Or, it can let them propagate higher to where they can be handled.

(Handling in this case probably means reporting a problem to the user.)

The caller also has more information. A boolean can only signal "was OK", or "was not OK". An exception is an object, with a class that is part of a class heirarchy. For example, the code can see a `RecordStoreException`, and know there was a record store issue. Or, it can look further, and differentiate between a `RecordStoreFullException` or a `RecordStoreNotFoundException`.

When It's OK to Swallow

Sometimes, it's OK to swallow an exception with an empty `catch{}`.

```
try {
    Thread.sleep(someTime);
} catch (InterruptedException e) {
    // never thrown in CLDC-1.0
}
```

Don't make the `catch{}` completely empty. Put a comment in with a reason why you've swallowed the exception. Some intellectual honesty is required here: it must a *reason*, not an *excuse*!

Be careful of code like this:

```
private Image background;

try {
    background = Image.createImage("/background.png");
} catch (IOException e) {
    background = null;
}
```

Set variables to null only if you want them, specifically, to be null.

If you set a variable to null, just for something to do in a catch, then you run the risk of a `NullPointerException` later on. Getting the wrong exception in the wrong place will make it much harder to find out what the problem is.

If you intend the variable to be null, as a marker that this build doesn't contain a background image (perhaps it's a build for a device with limited heap, or a JAR size constraint), and you have code elsewhere like:

Exception_Handling_in_Java

```
if (background != null) {
    g.drawImage(background, 0, 0, Graphics.TOP|Graphics.CENTER);
} else {
    // this build has no background image
    g.setColor(BACKGROUND_COLOR);
    g.fillRect(0, 0, screenWidth, screenHeight);
}
```

then pat yourself on the back for writing flexible code!

Use Exceptions

```
/**
 * @return false if cannot be stopped
 */
public boolean stop() {
    if (!started) {
        return false;
    }
    started = false;
    return true;
}
```

When other programmers use your code, let them use SEH. Even better, make them!

RULE 2: Don't use return values to indicate success or failure.

Return values from methods are best used for:

- Functions: methods that calculate some value based on parameters.
- Getters: methods that return information about an object's state.

Return values that return error codes are often ignored, and an ignored return value is useless.

```
/**
 * @throws IllegalStateException if not started
 */
public void stop() throws IllegalStateException {
    if (!started) {
        throw new IllegalStateException("cannot stop, not started");
    }
    started = false;
}
```

Provided everyone keeps to RULE 1, exceptions won't get ignored as easily as a return value. This will make bugs in the code easier to spot.

Where To Catch Exceptions

Unless you can deal with (by which I mean: actually fix the problem!) an exception low down in the code, the throw it higher. If the situation is not fixable, then throw it all the way up to the user interface, so the problem

can be reported.

RULE 3: Even an uncaught exception that crashes your program can be more use than one that is ignored.

Exceptions tell you what's wrong with your program. If you ignore them, you're only making work for yourself.

Summary

Exceptions exist to make your life easier. If you use them, they will.