ONJava Topics
All Articles
Best Practices
Enterprise JavaBeans
Java and XML
Java Data Objects
Java EE (Enterprise)
Java IDE Tools
Java Media
Java SE (Standard)
Java Security
Java SysAdmin
JDO/JDBC/SQLJ
JSP and Servlets
Open Source Java
P2P Java
Web Services
Wireless Java

Best Practices for Exception Handling

Print
Subscribe to ONJava
Subscribe to Newsletters
ShareThis

by Gunjan Doshi
11/19/2003

One of the problems with exception handling is knowing when and how to use it. In this article, I will cover some of the best practices for exception handling. I will also summarize the recent debate about the use of checked exceptions.

We as programmers want to write quality code that solves problems. Unfortunately, exceptions come as side effects of our code. No one likes side effects, so we soon find our own ways to get around them. I have seen some smart programmers deal with exceptions the following way:

```
public void consumeAndForgetAllExceptions(){
    try {
        ...some code that throws exceptions
    } catch (Exception ex){
        ex.printStacktrace();
    }
}
```

What is wrong with the code above?

Once an exception is thrown, normal program execution is suspended and control is transferred to the catch block. The catch block catches the exception and just suppresses it. Execution of the program continues after the catch block, *as if nothing had happened*.

How about the following?

```
public void someMethod() throws Exception{
}
```

This method is a blank one; it does not have any code in it. How can a blank method throw exceptions? Java does not stop you from doing this. Recently, I came across similar code where the method was declared to throw exceptions, but there was no code that actually generated that exception. When I asked the programmer, he replied "I know, it is corrupting the API, but I am used to doing it and it works."

It took the C++ community several years to decide on how to use exceptions. This debate has just started in the Java community. I have seen several Java programmers struggle with the use of exceptions. If not used correctly, exceptions can slow down your program, as it takes memory and CPU power to create, throw, and catch exceptions. If overused, they make the code difficult to read and frustrating for the programmers using the API. We all know frustrations lead to hacks and code smells. The client code may circumvent the issue by just ignoring exceptions or throwing them, as in the previous two examples.

The Nature of Exceptions

Broadly speaking, there are three different situations that cause exceptions to be thrown:

- **Exceptions due to programming errors:** In this category, exceptions are generated due to programming errors (e.g., `NullPointerException` and `IllegalArgumentException`). The client code usually cannot do anything about programming errors.
- **Exceptions due to client code errors:** Client code attempts something not allowed by the API, and thereby violates its contract. The client can take some alternative course of action, if there is useful information provided in the exception. For example: an exception is thrown while parsing an XML document that is not well-formed. The exception contains useful information about the location in the XML document that causes the problem. The client can use this information to take recovery steps.
- **Exceptions due to resource failures:** Exceptions that get generated when resources fail. For example: the system runs out of memory or a network connection fails. The client's response to resource failures is context-driven. The client can retry the operation after some time or just log the resource failure and bring the application to a halt.

Types of Exceptions in Java

Java defines two kinds of exceptions:

- **Checked exceptions:** Exceptions that inherit from the `Exception` class are checked exceptions. Client code has to handle the checked exceptions thrown by the API, either in a `catch` clause or by forwarding it outward with the `throws` clause.
- **Unchecked exceptions:** `RuntimeException` also extends from `Exception`. However, all of the exceptions that inherit from `RuntimeException` get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions.

By way of example, Figure 1 shows the hierarchy for `NullPointerException`.

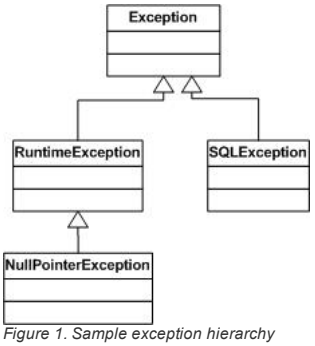

Figure 1. Sample exception hierarchy

In this diagram, `NullPointerException` extends from `RuntimeException` and hence is an unchecked exception.

I have seen heavy use of checked exceptions and minimal use of unchecked exceptions. Recently, there has been a hot debate in the Java community regarding checked exceptions and their true value. The debate stems from fact that Java seems to be the first mainstream OO language with checked exceptions. C++ and C# do not have checked exceptions at all; all exceptions in these languages are unchecked.

A checked exception thrown by a lower layer is a forced contract on the invoking layer to catch or throw it. The checked exception contract between the API and its client soon changes into an unwanted burden if the client code is unable to deal with the exception effectively. Programmers of the client code may start taking shortcuts by suppressing the exception in an empty catch block or just throwing it and, in effect, placing the burden on the client's invoker.

Checked exceptions are also accused of breaking encapsulation. Consider the following:

```
public List getAllAccounts() throws
    FileNotFoundException, SQLException{
    ...
}
```

The method `getAllAccounts()` throws two checked exceptions. The client of this method has to explicitly deal with the implementation-specific exceptions, even if it has no idea what file or database call has failed within `getAllAccounts()`, or has no business providing filesystem or database logic. Thus, the exception handling forces an inappropriately tight coupling between the method and its callers.

Best Practices for Designing the API

Having said all of this, let us now talk about how to design an API that throws exceptions properly.

1. When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"

If the client can take some alternate action to recover from the exception, make it a checked exception. If the client cannot do anything useful, then make the exception unchecked. By useful, I mean taking steps to recover from the exception and not just logging the exception. To summarize:

Client's reaction when exception happens	Exception type
Client code cannot do anything	Make it an unchecked exception
Client code will take some useful recovery action based on information in exception	Make it a checked exception

Moreover, *prefer unchecked exceptions for all programming errors*: unchecked exceptions have the benefit of not forcing the client API to explicitly deal with them. They propagate to where you want to catch them, or they go all the way out and get reported. The Java API has many unchecked exceptions, such as `NullPointerException`, `IllegalArgumentException`, and `IllegalStateException`. I prefer working with standard exceptions provided in Java rather than creating my own. They make my code easy to understand and avoid increasing the memory footprint of code.

2. Preserve encapsulation.

Never let implementation-specific checked exceptions escalate to the higher layers. For example, do not propagate `SQLException` from data access code to the business objects layer. Business objects layer do not need to know about `SQLException`. You have two options:

- Convert `SQLException` into another checked exception, if the client code is expected to recuperate from the exception.
- Convert `SQLException` into an unchecked exception, if the client code cannot do anything about it.

Most of the time, client code cannot do anything about `SQLException`s. Do not hesitate to convert them into unchecked exceptions. Consider the following piece of code:

```
public void dataAccessCode(){
    try{
        ..some code that throws SQLException
    }catch(SQLException ex){
        ex.printStacktrace();
    }
}
```

This `catch` block just suppresses the exception and does nothing. The justification is that there is nothing my client could do about an `SQLException`. How about dealing with it in the following manner?

```
public void dataAccessCode(){
    try{
        ..some code that throws SQLException
    }catch(SQLException ex){
        throw new RuntimeException(ex);
    }
}
```

This converts `SQLException` to `RuntimeException`. If `SQLException` occurs, the `catch` clause throws a new `RuntimeException`. The execution thread is suspended and the exception gets reported. However, I am not corrupting my business object layer with unnecessary exception handling, especially since it cannot do anything about an `SQLException`. If my `catch` needs the root exception cause, I can make use of the `getCause()` method available in all exception classes as of JDK1.4.

If you are confident that the business layer can take some recovery action when `SQLException` occurs, you can convert it into a more meaningful checked exception. But I have found that just throwing `RuntimeException` suffices most of the time.

Pages: **1**, 2

Next Page ▶

About O'Reilly
Sign In
Academic Solutions
Jobs
Contacts
Corporate Information
Press Room
Privacy Policy
Terms of Service
Writing for O'Reilly

Community
Authors
Community & Featured Users
Forums
Membership
Newsletters
O'Reilly Answers
RSS Feeds
User Groups

Partner Sites
makezine.com
makerfaire.com
craftzine.com
igniteshow.com
PayPal Developer Zone
O'Reilly Insights on Forbes.com

Shop O'Reilly
Customer Service
Contact Us
Shipping Information
Ordering & Payment
The O'Reilly Guarantee