

User name:

Book: SQL in a Nutshell, 2nd Edition

---

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

---

## 5.8. Error Handling

With everything that can go wrong in a database application, the correct handling of errors is a topic large enough for its own chapter (or book!). We're lucky that the APIs covered in this book, built on the modern Java language and on the .NET framework, contain error handling mechanisms that make recovering from errors easy.

### 5.8.1. Error Handling in ADO.NET

ADO.NET's error handling model is based on .NET exceptions. Functions or methods that can fail because of an error must be wrapped in a `try/catch/finally` block. The difficult part of handling exceptions is determining what to do when an error occurs and freeing resources for objects that were created. The following code example shows the error handling pattern that can be successfully used with ADO.NET to clean up all resources in the event of an error:

```
{Odbc|OleDb|Sql}Connection connection = null;
{Odbc|OleDb|Sql}Command statement = null;
{Odbc|OleDb|Sql}DataReader resultSet = null;
try {
    // Create and use the Connection, Command, and DataReader objects
    connection = new SqlConnection(connection_string);
    connection.Open( );

    statement = connection.CreateCommand( );
    statement.CommandText = SQL;

    resultSet = statement.ExecuteReader( );
} catch( Exception e ) {
    // Notify user that an error occurred
} finally {
    if( resultSet != null ) resultSet.Close( );
    if( statement != null ) statement.Dispose( );
    if( connection != null ) connection.Close( );
}
```

If an error occurs in the `try` block and an exception (of type `Exception` in this case) is thrown, the exception will be caught and the code in the `catch` block will be executed. The `finally` block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources. Which resources to free is difficult to determine if you don't know the point of failure. For example, if the failure occurs when executing a SQL statement, then the only resources that need to be freed are the `statement` and `connection` objects. In the code fragment above, an exception thrown in the `ExecuteReader` method would prevent the `resultSet` object from being closed, since it would still be set to its initial value of `null`.



It is good programming practice to free resources in the opposite order in which they were allocated.

### 5.8.2. Error Handling in JDBC

JDBC's error handling model is based on Java exceptions; therefore, the standard `try/catch/finally` error handling paradigm of Java can be applied to JDBC applications. The following Java code fragment is an example of how to use a `try/catch/finally` code block to gracefully handle application exceptions and free allocated resources:

```
Connection connection = null;
Statement statement = null;
ResultSet resultSet = null;
```

```

try {
    // Create and use the Connection, Statement, and ResultSet objects
    connection = DriverManager.getConnection( connection_string );
    statement = connection.createStatement( );
    resultSet = statement.executeQuery( SQL );

} catch( Exception e ) {
    // Then, notify the user of an error.
} finally {
    if( resultSet != null )
        try{resultSet.close( );} catch(Exception e) {}
    if( statement != null )
        try{statement.close( );} catch(Exception e) {}
    if( connection != null )
        try{connection.close( );} catch(Exception e) {}
}

```

If an error occurs in the `try` block and an exception (of type `Exception` in this case) is thrown, the exception will be caught and the code in the `catch` block will be executed. The `finally` block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources held by the `connection`, `result set`, and `statement` objects. Note that each object is first tested for a null value before freeing the resources held by the object. This NULL test is to make the code applicable to all error conditions — the code functions if the exception occurs while opening the connection as well as processing the `result set` data. The `close` methods on the JDBC objects can also generate exceptions when freeing resources, so they must be wrapped in their own `try/catch` blocks. This makes coding awkward, but even so, it will be relatively simple if you follow the pattern given here.



It is good programming practice to free resources in the opposite order from what they were allocated.