chapter 3

Advanced Exception Handling Concepts

Introduction

The last two chapters covered foundation concepts of using exceptions. They focused on the "hows" and "whys" of exception handling, showed how exceptions work and discussed how to deal with them in code. In this chapter, we move on to more advanced topics. Specifically, we discuss:

- How to create and use custom exceptions.
- Exception support for chaining and localization.
- How exceptions can be used in abstract classes and interfaces.
- Exception requirements for overridden methods.
- How exception handling code is represented in bytecodes.
- The efficiency of exception handling operations in an application.

These topics represent the more unusual tools in your exception handling toolbox. You probably won't need to use them in every single project, but you'll be glad to have them around for the more challenging programming jobs.

Custom Exceptions

So far, we've talked about how to handle exceptions when they're produced by calling methods in Java APIs. If you need to, Java also lets you create and use custom exceptions—classes of your own used to represent errors. You heard right. You can create brand new exceptions and use them within your applications.

Now that you know you *can* do this, a reasonable question is why would you want to? Why would you define new exception categories? W*ould you do this for fun*? P*rofit*? F*ame*? E*xcitement*? Normally, you create a custom exception to represent some type of error within your application—to give a new, distinct meaning to one or more problems that can occur within your code. You may do this to show similarities between errors that exist in several places throughout your code, to differentiate one or more errors from similar problems that could occur as your code runs, or to give special meaning to a group of errors within your application.

As an example, think about any kind of server. Its basic job is to handle communication with clients. If you used standard Java APIs (classes in the `java.io` and `java.net` packages, for instance) to write your server, you would create code that could throw `IOExceptions` in a number of places. You could throw `IOExceptions` when setting up the server, while waiting for a client connection, and when you get streams for communication. You could also throw `IOExceptions` during communication and when trying to break the connection. In short, virtually everything a server does could cause an `IOException`.

Do these `IOExceptions` all mean the same thing to the server? Probably not. Although they're all represented by the same type of exception, there could be a different business meaning (and different reporting and recovery actions) associated with each exception. You might associate one set of exceptions with problems in server configuration and startup, another set with the actual act of communication with a client, and a third set with the tasks associated with server shutdown. Custom exceptions give you the freedom to represent errors in a way that's meaningful to your application.

It's fairly easy to create and use a custom exception. There are three basic steps that you need to follow.

1. Define the exception class

You typically represent a custom exception by defining a new class.[7] In many cases, all you need to do is to create a subclass of an existing exception class:

```
public class CustomerExistsException extends Exception{
  public CustomerExistsException(){}
  public CustomerExistsException(String message){
    super(message);
  }
}
```

At a minimum, you need to subclass Throwable or one of its subclasses. Often, you'll also define one or more constructors to store information like an error message

7. It's also possible to use an existing exception class if it meets your needs. Developers often like to use a new exception class, because they can subsequently check for the exception in a `try-catch` block.

in the object, as shown in lines 2-4. When you subclass any exception, you automatically inherit some standard features from the Throwable class, such as:

1. Error message
2. Stack trace
3. Exception wrappering

Of course, if you want to add additional information to your exception, you can add variables and methods to the class:

```
public class CustomerExistsException extends Exception{
  private String customerName;
  public CustomerExistsException(){}
  public CustomerExistsException(String message){
    super(message);
  }
  public CustomerExistsException(String message, String customer){
    super(message);
    customerName = customer;
  }
  public String getCustomerName(){
    return customerName;
  }
}
```

This example shows how you might modify the class `CustomerExistsException` to provide support for an additional property in the exception. In this case, you could associate a String called `customerName` with the exception—the name of the customer for the record that caused the exception.

2. Declare that your error-producing method throws your custom exception

This step is actually the same as the “declare” part of the “handle or declare” rule. In order to use a custom exception, you must show classes that call your code that they need to plan for this new type of exception. You do this by declaring that one or more of your methods throws the exception:

```
public void insertCustomer(Customer c) throws CustomerExistsException{
  // The method stores customer information in the database.
  // If the customer data already exists, the method creates
  // and throws the CustomerExistsException.
}
```

3. Find the point(s) of failure in your error-producing method, create the exception and dispatch it using the keyword "throw"

The third and final step is to actually create the object and propagate it through the system. To do this, you need to know where your code will fail in the method. Depending on the circumstances, you may decide to use some or all of the following conditions to indicate a failure point in your code:

External Problems

- Exceptions produced in the application
- Failure codes returned by other methods

Internal Problems

- Inconsistent application state
- Problems with processing in the application

In our example, we encounter a failure scenario when we are unable to create a new customer. As a result, we create an exception to represent the problem and throw it. You can see an example in the following sample method:

```
public void insertCustomer(Customer c)
  throws CustomerExistsException, SQLException {
  String selectSql =
    "SELECT * FROM Customer WHERE first_name=? AND last_name=?";
  String insertSql = "INSERT INTO Customer VALUES(?, ?)";
  try{
    Connection conn = dbmsConnectionFactory.getConnection();
    PreparedStatement selStmt = conn.prepareStatement(selectSql);
    selectStmt.setString(1, c.getFirstName());
    selectStmt.setString(2, c.getLastName());
    ResultSet rs = selStmt.executeQuery();
    if (rs.next()){
      // In this case, the failure condition is produced if you
      //  can already locate a metching record in the database.
     throw new CustomerExistsException("Customer exists:" + c, c);
    }
    else{
      PreparedStatement insStmt = conn.prepareStatement(insertSql);
      insStmt.setString(1, c.getFirstName());
      insStmt.setString(2, c.getLastName());
      int status = insStmt.executeUpdate();
    }
  }
  catch (SQLException exc){
      Logger.log(exc);
      throw exc;
    }
  }
```

The Java keyword "throw" propagates the new exception object to this method's caller. Once you've followed these three steps, you've created a custom exception. Any objects that call this method will subsequently have to follow the handle or declare rule—unless, of course, you subclassed an unchecked exception class, such as RuntimeException or Error.

This raises an interesting point: What should you subclass when defining a custom exception? Well, you *must* subclass something in the Throwable class hierarchy, or you won't be able to propagate the exception in your application. Beyond that, you should never subclass Throwable directly. The Throwable class is intended to provide a behavioral base to the two main categories of problem—Exception and Error—and you shouldn't define new branches of the inheritance tree. It's also generally not a good idea to directly subclass Error or any of its subclasses, since custom exceptions don't usually fit the criteria for errors: "Serious problems that a reasonable application should not try to catch."

That leaves the Exception class hierarchy. You should generally define a custom exception as a subclass of whichever exception class that is a more general category of your own failure state. In our example, the `ServerConnectionException` subclasses `java.io.IOException`, since it represents a more specialized kind of that exception.

There's some question about whether it's good coding practice to define an exception that subclasses something in the RuntimeException tree. By doing this, you effectively circumvent the exception mechanism, since classes do not have to explicitly handle your exception even if you declare it.[8]

This example shows how to create a basic custom exception. In many cases, it will easily meet your needs. The custom exception class, and perhaps the message, are often the only things you need to use the exception in your application. Sometimes you may need to support more advanced features, though—and that usually means more code, right? There are two properties that you might want to use in some exceptions—chaining and localization.

8. There are a few Java APIs that use RuntimeException subclasses as a way to provide a flexible programming model—to ensure that the API can be expanded without forcing a developer to write an additional exception handling framework.

Chaining Exceptions

Starting with JDK1.4, the Throwable class introduced support for something called "exception chaining." Basically, chaining lets you set up an association between two exceptions. Prior to version 1.4, a few Java APIs used chaining as a way to group a set of related exceptions together. In JDBC, for example, `SQLException` objects can be chained so that it's possible to send back a variable number of exceptions associated with problems during the same database operation.

While you can certainly use chaining to set up groups of related exceptions in your code, a more common application is to set up a "root cause" association between exceptions. Suppose you catch an exception in code and want to throw a custom exception in response to the error. Normally, you'd lose all information from the original exception, or you'd have additional work to store additional information. If you use chaining, you can associate the original exception with your custom exception as the root cause of your problem. If you need any of the original information, you can get it by simply calling the method `getCause( )` on your custom exception.

There are two ways that you can set up a root cause for an exception: you can pass it in as a constructor argument, or you can set it after object creation by calling the `initCause(Throwable)` method. You can only set the root cause once, though. Once a value for the root cause has been set, the exception object you've created will hold onto that same root cause for the rest of its life. Setting the root cause of an exception in the constructor tends to be the more common way to establish it. It's quick, efficient, and leads to more compact code:

```
try{
  // Lots of risky I/O operations!
}
catch (java.io.IOException rootCause){
  throw new ConnectionException(rootCause);
}
```

Of course, you'd need to write a version of the constructor that would take Throwable as an argument and pass it to the superclass, in order to chain exceptions this way:

```
public class ConnectionException extends java.io.IOException{
  public ConnectionException(Throwable cause){
    initCause(cause);
  }
}
```

The main reason you'd use the `initCause` method would be to chain a custom exception class that predates JDK1.4. If you were working with a legacy exception class, it might not have a constructor that supported exception chaining. Since `initCause` is defined at the Throwable class level, any exception can call it, and be able to save an associated exception after its creation:

```
try{
  // Even more risky I/O operations!
}
catch (java.io.IOException rootCause){
  throw new LegacyException("Old exception").initCause(rootCause);
}
```

As mentioned, you can use root causes to "chain" a bunch of exceptions together. When you have a series of errors in your code, you can feed each one into the next as a root cause, and so create a group of exceptions that are related to each other. Naturally, the hope is that you don't have many cases where you'd have to do something like this, but it's a handy ability to have for more complex systems. When processing chained exceptions, you can write handler code to "walk" a set of chained exceptions and use the data that each object contains:

```
try{
  troublesomeObject.riskyMethod();
}
catch (ChainedException exc){
  System.out.println("We're in trouble now!!!");
  Throwable currentException = exc;
  do{
     System.out.println(currentException.toString());
     currentException = currentException.getCause();
  }
  while (currentException != null);
}
```

Exception Localization and Internationalization

Starting with JDK1.1, Throwable defined the method `getLocalizedMessage()`. By default, this method returns the same value as that provided by calling `getMessage()`—it returns whatever message is associated with the Throwable object. If you choose to, you can redefine this method in a custom Exception class to support locale-specific error messages. This lets you set up exceptions that support localization (l10n) and internationalization (i18n) in your code. These two characteristics allow applications to be used in multiple nations or regions without forcing everybody in that region to learn English (or even American).

In many cases, there isn't a great need to support exception localization. Normally, applications only provide for i18n of GUIs or standard program output. Even if an application supports different languages, many products use one standard language for the exceptions, since software products frequently have technical support provided by a development team that is located in a single country.

For those cases where you need to localize the text associated with an exception, `getLocalizedMessage()` can be overridden in a custom exception class, so that you can load locale-specific exception messages.

Java supports l10n and i18n with classes in the `java.util` and `java.text` packages.[9] The two key classes (from our point of view, at least) are java.util.ReourceBundle and java.util.Locale. The ResourceBundle class is a localizable wrapper around a collection. It allows you to store a series of key-value pairs, and to associate these values with a standard ISO-based locale naming scheme.[10] The locale mapping allows you to define a region based on a combination of language and location codes. For instance:

Code	Language and Region
de	German
en-CA	Canadian English
en-GB	British English
en-US	American English
fr	French

The Locale class is Java's way of representing these codes. A Locale object maps to a specific language or region, and allows you to look up the "standard" preferences for that region. The following steps show how to create a custom exception class that uses a localized message.

1. Create a ResourceBundle subclass to hold your message

The ResourceBundle class is used as a holding place for your resources. If you're interested in storing exception messages, an easy way to hold them is to subclass `ListResourceBundle`, a class that provides a base for managing arrays of elements.

9. The subclasses of `java.text.Format` are the key classes in the `java.text` package. They allow a developer to obtain locale-specific formatting for values such as numbers, dates, weights and measures and language syntax.

10. Actually, there's a good chance you have seen this kind of scheme if you've done work on the Web—the specification for language identification in HTML (RFC1766) is based on ISO639 (language abbreviations) and ISO3166 (country codes).

```
import java.util.ListResourceBundle;
public class ExceptionResourceBundle extends ListResourceBundle{
  private static final Object [][] contents = {
    {"criticalException", "A critical exception has occurred"}
  };
  public Object [][] getContents(){ return contents; }
}
```

2. Subclass the ResourceBundle for your different region(s)

The first ResourceBundle class you create will act as a default, the class to hold the messages that are used any time that Java cannot find a resource that is more suitable for the locale. To support other languages or regions, all you need to do is to create subclasses of the original ResourceBundle. When you do this, you're free to override some or all of the messages for the new language or region.

```
public class ExceptionResourceBundle_fr extends ExceptionResourceBundle{
  private static final Object [][] contents = {
    {"criticalException", "Il y'a quelque chose qui cloche!"}
  };
  public Object [][] getContents(){ return contents; }
}
```

3. Create a custom exception class that overrides `getLocalizedMessage` and uses the ResourceBundle to retrieve its message

The key modification for the custom exception class is to override the `getLocalizedMessage` method. If you plan to use localization, you need to call the static method `getBundle` from your ResourceBundle subclass. This method retrieves the appropriate `ResourceBundle` object based on the localization information used as input. Once that's done, you can call the method `getString` to retrieve a specific String stored in the bundle.

```
import java.util.Locale;
import java.util.ResourceBundle;
public class LocalizedException extends Exception{
  public static final String DEFAULT_MSG_KEY = "criticalException";
  private String localeMessageKey = DEFAULT_MSG_KEY ;
  private String languageCode;
  private String countryCode;
  public LocalizedException(String message){
    super(message);
  }
  public LocalizedException(Throwable cause, String  messageKey){
    super(cause);
    if (isValidString(messageKey)){
      localeMessageKey = messageKey;
    }
  }
  public LocalizedException(String defaultMessage, Throwable cause,
  String messageKey){
    super(defaultMessage, cause);
    if (isValidString(messageKey)){
      localeMessageKey = messageKey;
    }
  }
  public LocalizedException(String defaultMessage, String messageKey,
  String language, String country){
    super(defaultMessage);
    if (isValidString(messageKey)){
      localeMessageKey = messageKey;
    }
    if (isValidString(country)){
      countryCode = country;
    }
    if (isValidString(language)){
      languageCode = language;
    }
  }
  public void setLocaleMessageKey(String messageKey){
    if (isValidString(messageKey)){
      localeMessageKey = messageKey;
    }
  }
  public String getLocaleMessageKey(){
    return localeMessageKey;
  }
  public String getLocalizedMessage(){
    ResourceBundle rb = null;
    Locale loc = getLocale();
    rb = ResourceBundle.getBundle("ExceptionResourceBundle", loc);
    return rb.getString(localeMessageKey);
  }
  private Locale getLocale(){
    Locale locale = Locale.getDefault();
    if ((languageCode != null) && (countryCode != null)){
      locale = new Locale(languageCode, countryCode);
    }
    else if (languageCode != null){
      locale = new Locale(languageCode);
    }
    return locale;
  }
  private boolean isValidString(String input){
    return (input != null) && (!input.equals(""));
  }
}
```

When you do this, you've provided support for localization in an exception class. If you plan to use a number of localizable exception objects, it's probably worthwhile to define a localizable class that you can subclass for all of them. The best feature of the ResourceBundle is that it automatically defaults to the closest language/region match when you use any of the `getBundle` methods to perform a lookup. If we throw the following three `LocalizedException` objects

```
throw new LocalizedException("", "criticalException", "fr", "FR");
throw new LocalizedException("", "criticalException", "jp", "");
throw new LocalizedException("Internal application error");
```

and subsequently call the `getLocalizedMessage()` method in a handler block, we potentially get three different values for the exception message. In the first case, the call to getBundle uses the Locale object for French, and defaults to the closest match, `ExceptionResourceBundle_fr`. If we had defined `ExceptionResourceBundle_fr_FR`, it would have been used instead. In the second case, there is no Japanese version of the class,[11] so the class used defaults to the base resource bundle class, `ExceptionResourceBundle`. In the third example, the locale used is whatever is defined as the default for the system where the exception is created. If the application were run on a machine configured for a French speaker, the class returned would be `ExceptionResourceBundle_fr`. Otherwise, the base class `ExceptionResourceBundle` would be used.

Subclassing

When you create a subclass, you're free to redefine methods from the parent class, modifying the code. This is called overriding a method, and it gives you the freedom to modify how a method does its job so that it suits your needs for the subclass.

There are some fairly strict rules you have to observe when you override a method. You have to duplicate the method signature: the method's name, inputs and output have to match. You can never make the overriding method more private than its parent's method. You also must ensure that your method does not throw any exceptions other than those that are declared in your parent's method. This means that when you override a method, you can throw the same exceptions, or a subset of the exceptions,

11. Unfortunately, the author only knows enough Japanese to order sushi.

that were originally declared. You can *never* throw different exceptions without first defining them in your parent class method.

Think about why it's so important to throw the same exceptions that were declared in the parent class. Whenever a class defines a method, the exceptions declared in that method (apart from unchecked exceptions) represent the total set of things that can go wrong. When you write code that calls the method, you plan for that set of possibilities, structuring your handler code accordingly.

You can substitute a subclass for its parent. After all, the child class is a subclass of its parent, so you could potentially replace the parent with the child class anywhere in your code. If you call the overridden method, it will be the child's version of that method that is run.

Now, if the child's method can only produce a subset of its parent's exceptions, that's absolutely fine. If a calling method has a handler block for an exception that can never be produced, that isn't really a problem. You need to write some handling code for a problem that can never occur, but won't lead to application failure.

On the other hand, if a child class *could* override its parent's method and throw a different set of exceptions,[12] you could generate exceptions for which the handler code would be totally unprepared. Fundamentally, you'd be able to propagate errors that would have no recovery in your code, possibly halting your application and defeating the entire purpose of exception handling. For this reason, it's important that the compiler strictly enforce the "same or fewer" exception rule in overridden methods.

Exception Declaration for Interfaces and Abstract Classes

It's fairly obvious what an exception means when it's declared for a method of a class, but what does it mean when an exception is declared for an abstract method, or an interface? Keep in mind, after all, that an abstract method only provides a signature for a method; it can't actually define any code. And an interface is basically composed of nothing but abstract methods.

In cases like these, what does it mean to say that a specific exception can be thrown? Whenever you define an exception for such a method, you're basically setting an expectation of what could be expected to go wrong when the method is eventually implemented.

```
public interface CommunicationServer{
  public void processClient() throws ServerConnectionException;
  public void listen();
}
```

12. This could *never* happen in Java—it would result in a compiler error. For the sake of argument, let's imagine that it could be done, to see what the result would be.

If you create a class that implements `CommunicationServer`, you'll write your own version of the `processClient` method. Because the interface declares that the method can throw a `ServerConnectionException`, you're free to throw that exception in your code if you need to. More importantly, you know that if you decide to throw the exception, any other class that's using the `CommunicationServer` interface will have to handle or declare this exception. Even if you write an implementation of `CommunicationServer` many months after the interface was created, you can be certain that the classes using the interface are prepared to address the possible `ServerConnectionException`.

```
public class ServerManager{
  private CommunicationServer commoServer = new CommunicationServerImpl();
  private boolean shutdown;
  public void handlerCycle(){
    while (!shutdown){
      commoServer.listen();
      try{
        commoServer.processClient();
      }
      catch (ServerConnectionException exc){
        Logger.log("Client connection error", exc);
      }
    }
  }
}
```

Exception Stack Traces

In object-oriented programming, most complex actions are performed as a series of method calls. This is a natural consequence of two programming goals: the desire to define reusable units of code, and the desire to progressively break complicated tasks into smaller, easier to manage subtasks. Combine this practice with the tendency to define a number of objects that work together to accomplish a programming task, and you have a programming model where a number of objects communicate with each other through a series of method calls to perform their work. It's normal for a chain of method calls, commonly referred to as a "call stack," to occur as an object-oriented application runs.

For a demonstration of this, think about a business task that a server might perform. For instance, how would the server handle a response to add a new customer record to a database? Although it would be possible for you to write the code for this entire behavior in a single method, the result would be *extremely* hard to understand, difficult to maintain, and probably impossible to reuse. Instead, it would be preferable to divide the work up among several objects. A Server would manage the overall server lifecycle, a Communication Delegate would handle communication with a client, a Business Delegate would interpret and act on client requests, a Customer Handler would hold the methods to handle any Customer business actions, and a `DataAccessObject` would communicate with the database. Table 3-1 shows a series of calls within a server as it processes a client's request to create a new customer account. The entries on the left show which class contains the method, and the ones on the right show the method call that has been made.

Table 3–1 Call stack to create a new customer account

Class	Method
Server	main
Server	configure
Server	connect
CommunicationDelegate	run
CommunicationWorkThread	run
CommunicationWorkThread	readInput
ShoppingService	processInput
CustomerAccountService	createCustomer
CustomerAccountService	isCustomerValid
CustomerAccountService	createAccount
AccountDAO	insertCustomer
AccountDAO	insertAddress
AccountDAO	insertPaymentInfo
ShoppingService	processOutput
CommunicationWorkThread	writeOutput

In this example, you can clearly see the series of method calls, as the main method runs the handler method, then receives a request from a client, then identifies it as a request to create a new customer. What happens when an exception occurs as a method in the call stack, such `insertCustomer`, is run? If the exception were handled in a `try-catch-finally` block of that method, nothing would happen. The `insertCustomer` method would handle the exception normally, and the code would continue to run.

If `insertCustomer` declared the exception, the situation would be quite a bit different. In that case, the exception would cause `insertCustomer` to stop execution, and the exception would be propagated to the method's caller, the method `createCustomer`. That method would have the same choice—to handle or declare the exception—and if it declared the exception, that method would halt, and the exception would be propagated further up the call stack.

As developers, we have the freedom to handle an exception at any point within a call stack that best suits our needs for an application. We can handle a problem in the method that produces an exception, or we can handle it at some remote point in the call sequence.

As an exception is propagated upward in a call chain,[13] it maintains a structure called a **"stack trace."** A stack trace is a record of every method that failed to handle the exception, as well as the line in code where the problem occurred. When an exception is propagated to the caller of a method, it adds a line to the stack trace indicating the failure point in that method. In the previous example, a stack trace might look something like this:

```
SQL Exception: The statement was aborted because it would have caused
  a duplicate key value in a unique or primary key constraint defined
  on 'CUSTOMER(LAST_NAME)'.
    at c8e.p.i._f1(Unknown Source)
    at c8e.p.q._b84(Unknown Source)
    at c8e.p.q.handleException(Unknown Source)
    at c8e.p.n.handleException(Unknown Source)
    at c8e.p.p.handleException(Unknown Source)
    at c8e.p.j.executeStatement(Unknown Source)
    at c8e.p.g.execute(Unknown Source)
    at c8e.p.g.executeUpdate(Unknown Source)
    at
RmiJdbc.RJPreparedStatementServer.executeUpdate(RJPreparedStatementServer.java:
74)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.ja
va:25)
    at java.lang.reflect.Method.invoke(Method.java:324)
    at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:261)
    at sun.rmi.transport.Transport$1.run(Transport.java:148)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.rmi.transport.Transport.serviceCall(Transport.java:144)
    at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
    at
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
    at java.lang.Thread.run(Thread.java:534)
    at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall
.java:247)
```

13. In this case, "up" means toward the method that was originally used to start the application.

```
    at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:223)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:133)
    at RmiJdbc.RJPreparedStatementServer_Stub.executeUpdate(Unknown Source)
    at RmiJdbc.RJPreparedStatement.executeUpdate(RJPreparedStatement.java:80)
    at AccountDAO.insertCustomer(AccountDAO.java:69)
    at CustomerAccountService.createAccount(CustomerAccountService.java:13)
    at CustomerAccountService.createCustomer(CustomerAccountService.java:6)
    at ShoppingService.processInput(ShoppingService.java:6)
    at CommunicationWorkThread.run(CommunicationWorkThread.java:21)
    at java.lang.Thread.run(Thread.java:534)
```

Looks a little imposing, doesn't it? As promised, it holds a record of every point within the application where the execution stopped. Once you know the basic format of an exception stack trace, it's a lot easier to understand. The first few lines document the exception that was thrown—specifically, they show the exception class type and the detail message of the exception. Next, the stack trace documents the stopping points in your code. Each line in the stack trace shows a place where execution has halted in a called method. It indicates the class, the method name within the class, and the line in the file that corresponds to the failure point.[14] As the lines progress, they work from the innermost called method "upward" toward the starting point of the business action. In this case, the business action was performed by a thread, so the last entry is the thread's run method.

In a complex enterprise application, there can be many, many levels in a call stack. In this example, the business code in the example communicates with a database using a JDBC driver—an adapter that manages communications with the DBMS. Most of the entries in the call stack correspond to the internal code of the database driver. It isn't until you look at the bottom of the call stack that you can see the failure point in the application code: the `insertCustomer` method is the first method from the application code that has an error.

If you know how to interpret the information, a stack trace can be an exceptionally valuable tool when debugging code. For instance, if you look closely at the output from the earlier example, you can learn a few things about the problem. First, the exception was thrown because a database insert would have resulted in a duplicate primary key in the database—specifically, with the `last_name` field defined in the Customer table. Second, the point of origin in the application code was in the `insertCustomer` method. If you put these two facts together, you might think that the exception was thrown because of the customer entry for the account already existed.[15] You could subsequently develop a test for the theory and correct the problem in your code.

14. In some cases, the exact position within the source code may not be available; in that case, you will see the indicator Unknown Source if debug information is not available for the class file, or Native Method if the call corresponds to a method in native code.

15. And you would be right!

Low-Level Exception Handling

What exactly goes on in Java when you use exceptions? When you handle or declare an exception in your code, exactly what are you doing? To answer this question, you really have to look at the bytecodes that are produced when you compile a Java class. Java provides a utility called `javap`—a general purpose tool to *profile decompiled* class files—which can also interpret bytecodes when it's run with the correct switch. To get a good idea of what different exception handling choices mean in bytecode, we need to compare between three scenarios:

- A situation where no exception is produced
- A situation where an exception is declared
- A situation where an exception is handled

To ensure that we can see the differences in a uniform code example, we'll use an unchecked exception to demonstrate. This simple example takes an argument from standard input and tries to convert it into an integer by using the `Integer.parseInt` method. If you pass any non-integer value as an argument, the method will produce a `NumberFormatException`, which is a type of RuntimeException.

```
public class BytecodeExample{
  public static void main(String [] args){
    BytecodeExample app = new BytecodeExample();
    if (args.length > 0){
      app.readIntTryCatch(args[0]);
      app.readIntDeclare(args[0]);
        app.readInt(args[0]);
    }
  }

  public int readInt(String input){
    int returnValue = 0;
    returnValue = Integer.parseInt(input);
    return returnValue;
  }

  public int readIntTryCatch(String input){
    int returnValue = 0;
    try{
      returnValue = Integer.parseInt(input);
    }
    catch(NumberFormatException exc){}
    return returnValue;
  }

  public int readIntDeclare(String input) throws NumberFormatException{
    int returnValue = 0;
    returnValue = Integer.parseInt(input);
    return returnValue;
  }
}
```

After compiling this code, you can produce a text file with the interpreted bytecodes by executing the following command:

```
javap -c -verbose BytecodeExample > bytecodes.txt
```

This command runs the `javap` utility on `BytecodeExample.class` with bytecodes interpretation (`-c` switch) providing verbose output (`-verbose switch`) and saving the output into the file `bytecodes.txt`. When you run this command, you'll see the following result:

```
Compiled from BytecodeExample.java
public class BytecodeExample extends java.lang.Object {
    public BytecodeExample();
   /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
   /* Stack=3, Locals=2, Args_size=1 */
    public int readInt(java.lang.String);
   /* Stack=1, Locals=3, Args_size=2 */
    public int readIntTryCatch(java.lang.String);
   /* Stack=1, Locals=4, Args_size=2 */
    public int readIntDeclare(java.lang.String) throws
java.lang.NumberFormatException;
   /* Stack=1, Locals=3, Args_size=2 */
}

Method int readInt(java.lang.String)
   0 iconst_0
   1 istore_2
   2 aload_1
   3 invokestatic #7 <Method int parseInt(java.lang.String)>
   6 istore_2
   7 iload_2
   8 ireturn

Method int readIntTryCatch(java.lang.String)
   0 iconst_0
   1 istore_2
   2 aload_1
   3 invokestatic #7 <Method int parseInt(java.lang.String)>
   6 istore_2
   7 goto 14
  10 astore_3
  11 goto 14
  14 iload_2
  15 ireturn
Exception table:
   from   to  target type
     2     7    10   <Class java.lang.NumberFormatException>

Method int readIntDeclare(java.lang.String)
   0 iconst_0
   1 istore_2
   2 aload_1
   3 invokestatic #7 <Method int parseInt(java.lang.String)>
   6 istore_2
   7 iload_2
   8 ireturn
```

If you look at the listing, you'll notice a few interesting differences for the methods that work directly with the `NumberFormatException`. The `readIntDeclare` method has the same bytecodes produced as the standard `readInt` method, but the exception throw clause is declared in the method lookup table for the class. The handle example, `readIntTryCatch`, actually has several additional bytecode instructions and an "exception table" associated with the method. The exception table controls code routing if the exception is produced. In this case, the code silently handles the exception, so the method basically behaves the same as the other two methods.

By this time, I'll guarantee that a few people are wondering about how this affects performance. You've seen that handling or declaring an exception really **does** result in different bytecodes—so what's the cost in terms of runtime efficiency? It's a lot like taking a date to an expensive restaurant where they don't put the prices on the menu. At some point during the meal, you're bound to think, "This is all very nice, but what will it cost me?"

To get a general idea of the relative cost of using exceptions, we can run our `BytecodeExample` using a homemade profiler. The profiler doesn't have to be very complicated—it just needs to be able to keep track of how long it takes to execute a method. The source code for a sample profiler is shown below:

```
import java.util.Date;
import java.text.SimpleDateFormat;
public class Profile{
  private Runtime runtime;
  private long startTime, stopTime, timeElapsed;
  private SimpleDateFormat sdf = new SimpleDateFormat("mm:ss.SSSS");
  public Profile(){
    runtime = Runtime.getRuntime();
  }
  public void startTimer(){
    startTime = System.currentTimeMillis();
  }
  public void stopTimer(){
    stopTime = System.currentTimeMillis();
    timeElapsed += stopTime - startTime;
  }
  public void clearTimer(){
    startTime = stopTime = timeElapsed = 0;
  }
  public long getStartTimeMillis(){
    return startTime;
  }
  public long getStopTimeMillis(){
    return stopTime;
  }
  public long getStartStopTimeMillis(){
    return stopTime - startTime;
  }
  public long getTimeElapsedMillis(){
    return timeElapsed;
  }
  public Date getStartTime(){
    return new Date(startTime);
  }
  public Date getStopTime(){
    return new Date(stopTime);
  }
  public Date getStartStopTime(){
    return new Date(stopTime - startTime);
  }
  public Date getTimeElapsed(){
    return new Date(stopTime);
  }
  public String getTimeElapsedAsString(){
    StringBuffer buffer = new StringBuffer();
    return sdf.format(new Date(timeElapsed));
  }
  public String getStartStopTimeAsString(){
    StringBuffer buffer = new StringBuffer();
    return sdf.format(new Date(stopTime - startTime));
  }
```

```
}
```

To get a general idea of the relative cost of each exception handling option, we ran a profiler with each option 1,000,000 times, which produced the following results:

1,000,000 iterations, no exception produced

Handle:	0.433 s total, 0.433 s/method call
Declare:	0.411 s total, 0.411 s/method call
No Handling:	0.414 s total, 0.414 s/method call

1,000,000 iterations, `NumberFormatException` produced

Handle:	10.93 s total, 10.93 s/method call
Declare:	10.52 s total, 10.52 s/method call
No Handling:	10.49 s total, 10.49 s/method call

Based on the test results,[16] you can see that there really isn't a substantial performance difference between the different options. Even in situations where you execute the same code many, many times (keep in mind that the profiler ran the examples 1,000,000 times) there isn't a major performance gap between the three exception handling options. Naturally, you see a gap in application performance when you compare a situation where an exception is produced to one where no exception is generated, but that's to be expected.

The good news about all of this is that it essentially leaves you free to follow best programming practices; you aren't faced with unpleasant trade-offs between effective coding and performance. This means that our basic rule for working with exceptions can still stand. When working with exception-producing code,[17] the order of preference could still be the following:

1. Avoid throwing an exception if you can.
2. Handle an exception if you can.
3. Declare an exception if you must.

16. For detailed information about the test as well as complete code, refer to Appendix A.
17. Of course, this may not be the case when the exception has to be propagated across a network!