Best practices in EJB exception handling

Learn to code for faster problem resolution on EJB-based systems

Srikanth Shenoy

May 01, 2002

As J2EE has become the enterprise development platform of choice, more and more J2EE-based applications are going into production. One important component of the J2EE platform is the Enterprise JavaBeans (EJB) API. Together, J2EE and EJB technology offer many advantages, but with these advantages come new challenges. In particular, any problem in an enterprise system must be resolved quickly. In this article, Enterprise Java programming veteran Srikanth Shenoy reveals his best practices in EJB exception handling for faster problem resolution.

Exception handling is simple enough in a hello-world scenario. Whenever you encounter an exception in a method, you catch the exception and print the stack trace or declare the method to throw the exception. Unfortunately, this approach isn't sufficient to handle the types of exceptions that arise in the real world. In a production system, when an exception is thrown it's likely that the end user is unable to process his or her request. When such an exception occurs, the end user normally expects the following:

- A clear message indicating that an error has occurred
- A unique error number that he can use upon accessing a readily available customer support system
- Quick resolution of the problem, and the assurance that his request has been processed, or will be processed within a set time frame

Ideally, an enterprise-level system will not only provide these basic services to the customer, but will also have a few essential back-end mechanisms in place. The customer service team should, for example, receive immediate error notification, so that the service representative is aware of the problem before the customer calls for resolution. Furthermore, the service representative should be able to cross-reference a user's unique error number and the production logs for quick identification of the problem -- preferably up to the exact line number or the exact method. In order to provide both the end user and the support team with the tools and services they need, you must have a clear picture, as you are building a system, of everything that can go wrong with it once it is deployed.
In this article we'll talk about exception handling in EJB-based systems. We'll start with a review of exception-handling basics, including the use of logging utilities, then move quickly into a more detailed discussion of how EJB technology defines and manages different types of exception. From there, we'll use code examples to look at the pros and cons of some common exception-handling solutions, and I'll reveal my own best practices for making the most of EJB exception handling.

Note that this article assumes you are familiar with J2EE and EJB technologies. You should understand the difference between entity beans and session beans. It will also be helpful if you have some knowledge of what bean-managed persistence (BMP) and container-managed persistence (CMP) mean in the entity bean context. See the Related topics section to learn more about J2EE and EJB technologies.

Exception-handling basics
The first step in resolving a system error is to set up a test system with the same build as the production system and trace through all the code that led up to that exception being thrown, as well as all the various branches in the code. In a distributed application, chances are that the debugger doesn't work, so you'll likely be using System.out.println() methods to track the exception. While they come in handy, System.out.println()s are expensive. They synchronize processing for the duration of disk I/O, which significantly slows throughput. By default, stack traces are logged to the console. But browsing the console for an exception trace isn't feasible in a production system. In addition, they aren't guaranteed to show up in the production system, because system administrators can map System.outs and System.errs to ‘ ’ on NT and dev/null on UNIX. Moreover, if you're running the J2EE app server as an NT service, you won't even have a console. Even if you redirect the console log to an output file, chances are that the file will be overwritten when the production J2EE app servers are restarted.

### Principles of exception handling
The following are some of the generally accepted principles of exception handling:

1. If you can't handle an exception, don't catch it.
2. If you catch an exception, don't swallow it.
3. Catch an exception as close as possible to its source.
4. Log an exception where you catch it, unless you plan to rethrow it.
5. Structure your methods according to how fine-grained your exception handling must be.
6. Use as many typed exceptions as you need, particularly for application exceptions.

Point 1 is obviously in conflict with Point 3. The practical solution is a trade-off between how close to the source you catch an exception and how far you let it fall before you've completely lost the intent or content of the original exception.

**Note:** These principles are not particular to EJB exception handling, although they are applied throughout the EJB exception-handling mechanisms.

For these reasons, rolling your code into production with System.out.println()s included isn't an option. Using them during testing and then removing them before production isn't an elegant solution either, because doing so means your production code won't function the same as your test...
code. What you need is a mechanism to declaratively control logging so that your test code and your production code are the same, and performance overhead incurred in production is minimal when logging is declaratively turned off.

The obvious solution here is to use a logging utility. With the right coding conventions in place, a logging utility will pretty much take care of recording any type of messages, whether a system error or some warning. So, we'll talk about logging utilities before we go any further.

**Logging landscape: A bird's eye view**

Every large application uses logging utilities in development, testing, and production cycles. The logging landscape today has a handful of players, and among them two are most widely known. One is Log4J, an open source project from Apache under Jakarta. The other is a recent entry that comes bundled with J2SE 1.4. We'll use Log4J to illustrate the best practices discussed in this article; however, these best practices aren't specifically tied to Log4J.

Log4J has three main components: layout, appender, and category. *Layout* represents the format of the message to be logged. *Appender* is an alias for the physical location at which the message will be logged. And *category* is the named entity; you can think of it as a handle for logging. Layouts and appenders are declared in an XML configuration file. Every category comes with its own layout and appender definitions. When you get a category and log to it, the message ends up in all the appenders associated with that category, and all those messages will be represented in the layout format specified in the XML configuration file.

Log4J assigns four priorities to messages: they are ERROR, WARN, INFO, and DEBUG. For the purpose of this discussion all exceptions are logged with ERROR priority. When logging an exception in this article, we will find the code that gets the category (using the `Category.getInstance(String name)` method) and then invoke the method `category.error()` (which corresponds to the message with the priority of ERROR).

While logging utilities help us to log the message to appropriate persistent location(s), they cannot fix the root of the problem. They cannot pinpoint an individual customer's problem report from the production logs; this facility is left up to you to build into the system you are developing.

For more information about Log4J or the J2SE logging utility, see the **Related topics** section.

**Exception categories**

Exceptions are classified in different ways. Here, we'll talk about how they're classified from an EJB perspective. The EJB spec classifies exceptions into three broad categories:

- **JVM exceptions**: This type of exception is thrown by the JVM. An `OutOfMemoryError` is one common example of a JVM exception. There is nothing you can do about JVM exceptions. They indicate a fatal situation. The only graceful exit is to stop the application server, maybe beef up the hardware resources, and restart the system.
- **Application exceptions**: An application exception is a custom exception thrown by the application or a third-party library. These are essentially checked exceptions; they denote that
some condition in the business logic has not been met. Under these conditions, the caller of the EJB method can gracefully handle the situation and take an alternative path.

- **System exceptions**: Most often system exceptions are thrown as subclasses of RuntimeException by the JVM. A NullPointerException, or an ArrayOutOfBoundsException, for example, will be thrown due to a bug in the code. Another type of system exception occurs when the system encounters an improperly configured resource such as a misspelled JNDI lookup. In this case, it will throw a checked exception. It makes a lot of sense to catch these checked system exceptions and throw them as unchecked exceptions. The rule of thumb is, if there isn’t anything you can do about an exception, it’s a system exception and it should be thrown as an unchecked exception.

**Note**: A **checked exception** is a Java class that subclasses java.lang.Exception. By subclassing java.lang.Exception, you are forced to catch the exception at compile time. In contrast, an **unchecked exception** is one that subclasses java.lang.RuntimeException. Subclassing java.lang.RuntimeException ensures you will not be forced by the compiler to catch the exception.

### How the EJB container handles exceptions

The EJB container intercepts every method call on the EJB component. As a result, every exception that results in a method call is also intercepted by the EJB container. The EJB specification deals only with handling two types of exception: application exceptions and system exceptions.

An **application exception** is defined by the EJB spec as any exception declared on the method signatures in the remote interface (other than RuntimeException). An application exception is a special scenario in the business workflow. When this type of exception is thrown, the client is given a recovery option, usually one that entails processing the request in a different way. This does not, however, mean that any unchecked exception declared in the throws clause of a remote-interface method would be treated as an application exception. The spec states clearly that application exceptions should not extend RuntimeException or its subclasses.

When an application exception occurs, the EJB container doesn't roll back the transaction unless it is asked to do so explicitly, with a call to the setRollbackOnly() method on the associated EJBContext object. In fact, application exceptions are guaranteed to be delivered to the client as is: the EJB container does not wrap or massage the exception in any way.

A **system exception** is defined as either a checked exception or an unchecked exception, from which an EJB method cannot recover. When the EJB container intercepts an unchecked exception, it rolls back the transaction and does any necessary cleanup. Then the container wraps the unchecked exception in a RemoteException and throws it to the client. Thus the EJB container presents all unchecked system exceptions to the client as RemoteExceptions (or as a subclass thereof, such as TransactionRolledbackException).

In the case of a checked exception, the container does not automatically perform the housekeeping described above. To use the EJB container's internal housekeeping, you will have to have your checked exceptions thrown as unchecked exceptions. Whenever a checked system
exception (such as a NamingException) occurs, you should throw javax.ejb.EJBException, or a subclass thereof, by wrapping the original exception. Because EJBException itself is an unchecked exception, there is no need to declare it in the throws clause of the method. The EJB container catches the EJBException or its subclass, wraps it in a RemoteException, and throws the RemoteException to the client.

Although system exceptions are logged by the application server (as mandated by the EJB specification) the logging format will differ from one application server to another. Often, an enterprise will need to run shell/Perl scripts on the generated logs in order to access needed statistics. To ensure a uniform logging format, it is better to log the exceptions in your code.

**Note**: The EJB 1.0 spec required that checked system exceptions be thrown as RemoteExceptions. Starting with EJB 1.1, the spec has mandated that EJB implementation classes should not throw RemoteException at all.

**Common exception-handling strategies**

In the absence of a strategy for exception handling, different developers on the project team will likely write code to handle exceptions differently. At the very least, this can result in confusion for the production support team, since a single exception may be described and handled differently in different areas of the system. Lack of strategy also results in logging at multiple places throughout the system. Logging should be centralized or broken out into multiple manageable units. Ideally, exception logging should occur in as few places as possible without compromising the content. In this section and the ones that follow, I’ll demonstrate coding strategies that can be implemented in a uniform way throughout an enterprise system. You can download the utility classes developed in this article from the Related topics section.

Listing 1 shows a method from a session EJB component. This method deletes all orders placed by a customer before a particular date. First, it gets Home Interface for OrderEJB. Next, it fetches all orders from the particular customer. When it encounters an order placed before a particular date, it deletes the order item, then deletes the order itself. Note that three exceptions are thrown, and three common exception-handling practices are shown. (For simplicity, assume that compiler optimizations are not in use.)

**Listing 1. Three common exception-handling practices**

```java
try {
    OrderHome homeObj = EJBHomeFactory.getInstance().getOrderHome();
    Collection orderCollection = homeObj.findByCustomerId(id);
    Iterator orderIter = orderCollection.iterator();
    while (orderIter.hasNext()) {
        Order orderRemote = (OrderRemote) orderIter.getNext();
        OrderValue orderVal = orderRemote.getValue();
        if (orderVal.getDate() < "mm/dd/yyyy") {
            OrderItemHome itemHome =
                EJBHomeFactory.getInstance().getItemHome();
            Collection itemCol = itemHome.findByOrderId(orderId);
            Iterator itemIter = itemCol.iterator();
            while (itemIter.hasNext()) {
                OrderItem item = (OrderItem) itemIter.getNext();
                item.remove();
            }
            orderRemote.remove();
        }
    }
}
```

Best practices in EJB exception handling
Now, let's use the code illustration above to look at the flaws in the three demonstrated exception-handling practices.

**Throwing/rethrowing an exception with an error message**

A `NamingException` can occur on line 101 or 108. When a `NamingException` occurs, the caller of this method gets a `RemoteException` and can track the exception back to line 119. The caller cannot tell if the actual `NamingException` occurred on line 101 or line 108. Since the exception content isn't preserved until it is logged, the source of the problem is untraceable. In this type of scenario, the content of the exception is said to have been "swallowed." As this example shows, throwing or rethrowing an exception with a message is not a good exception-handling solution.

**Logging to the console and throwing an exception**

A `FinderException` can occur on line 102 or 109. Since the exception is logged to the console, however, the caller can trace back to lines 102 or 109 only if the console is available. Obviously, this isn't feasible, so the exception can only be traced back to line 122. The reasoning is the same here as above.

**Wrapping the original exception to preserve the content**

A `RemoteException` can occur on line 102, 106, 109, 113, or 115. It is caught in `catch` block on line 123. It is then wrapped in an `EJBException`, so it can remain intact wherever the caller logs it. While better than the previous two, this approach demonstrates the absence of a logging strategy. If the caller of the `deleteOldOrders()` method logs the exception, it will result in duplicate logging. And, in spite of the logging, the production logs or console cannot be cross-referenced when the customer reports a problem.

**EJB exception-handling heuristics**

Which exceptions should EJB components throw and where should you log them in your system? These two questions are intricately linked and should be addressed together. The answer depends on the following factors:

- **Your EJB system design**: In good EJB design, clients never invoke methods on entity EJB components. Most entity EJB method invocation occurs in session EJB components. If your design is along these lines, you should log your exception with the session EJB components. If the client invokes entity EJB methods directly, then you should log the messages in the entity EJB components, as well. There's a catch, however: the same entity EJB methods may be invoked by session EJB components, too. How do you prevent duplicate logging in such a
scenario? Similarly how do you prevent duplicate logging when one session EJB component invokes other? We'll soon explore a generic solution to handle both of these cases. (Note that EJB 1.1 does not architecturally prevent clients from invoking methods on entity EJB components. In EJB 2.0 you can mandate this restriction by defining local interfaces for entity EJB components.)

- **The extent of planned code reuse**: The issue here is whether you plan to add logging code in multiple places or to redesign and refactor your code for reduced logging code.
- **The type of clients you want to serve**: It is important to consider whether you will be serving a J2EE Web tier, stand-alone Java applications, PDAs, or other clients. Web-tier designs come in all shapes and sizes. If you're using the Command pattern, in which the Web tier invokes the same method in the EJB tier by passing in a different command every time, it is useful to log the exception in the EJB component, where command execution occurs. In most other Web-tier designs it is easier and better to log the exceptions in the Web tier itself, since you need to add the exception logging code in fewer places. You should consider the latter option if you have co-located Web tiers and EJB tiers and you don't have a requirement to support any other type of client.

- **The type of exception (application or system) you'll be dealing with**: Handling application exceptions is significantly different from handling system exceptions. System exceptions occur without the intention of the EJB developer. Because the intent of a system exception is unclear, the content should indicate the context of the exception. As you've seen, this is best handled by wrapping the original exception. On the other hand, application exceptions are explicitly thrown by the EJB developer, often by wrapping a message. Because the intent of an application exception is clear, there is no reason to preserve its context. This type of exception need not be logged in the EJB tier or the client tier; rather, it should be presented to the end user in a meaningful way, with an alternate path to resolution provided. System exception messages need not be very meaningful to the end user.

### Handling application exceptions

In this section and several that follow we'll look more closely at EJB exception handling for application exceptions and system exceptions, as well as Web tier designs. As part of this discussion, we'll explore different ways of handling the exceptions thrown from session and entity EJB components.

### Application exceptions in entity EJB components

Listing 2 shows an ejbCreate() method on an entity EJB. The caller of this method passes in an OrderItemValue and requests the creation of an OrderItem entity. Because OrderItemValue doesn't have a name, a CreateException is thrown.

#### Listing 2. Sample ejbCreate() method in an entity EJB component

```java
public Integer ejbCreate(OrderItemValue value) throws CreateException {
    if (value.getItemName() == null) {
        throw new CreateException("Cannot create Order without a name");
    }
    ..
    ..
    return null;
}
```
Listing 2 shows a very typical usage of a CreateException. Similarly, a finder method will throw a FinderException if the input arguments for a method do not have right values.

If you're using container-managed persistence (CMP), however, the developer doesn't have control over the finder method and FinderException may never be the thrown by the CMP implementation. Nonetheless, it is better to declare the FinderException in the throws clause for the finder methods on the Home interface. RemoveException is another application exception that is thrown when the entity is deleted.

Application exceptions thrown from entity EJB components are fairly limited to these three types (CreateException, FinderException, and RemoveException) and their subclasses. Most of the application exceptions originate from session EJB components because that's where intelligent decision making happens. Entity EJB components are generally dumb classes whose sole responsibility is to create and fetch data.

**Application exceptions in session EJB components**

Listing 3 shows a method from a session EJB component. The caller of this method tries to order n quantities of an item of a particular type. The SessionEJB() method figures out that there aren't enough quantities in stock and throws a NotEnoughStockException. The NotEnoughStockException applies to a business-specific scenario; when this exception is thrown, an alternative route is proposed to the caller, enabling him to order a smaller number of items.

Listing 3. Sample container callback method in a session EJB component

```java
public ItemValueObject[] placeOrder(int n, ItemType itemType) throws NotEnoughStockException {
    //Check Inventory.
    Collection orders = ItemHome.findByItemType(itemType);
    if (orders.size() < n) {
        throw NotEnoughStockException("Insufficient stock for " + itemType);
    }
}
```

**Handling system exceptions**

System exception handling is a more involved discussion than application exception handling. Because session EJB components and entity EJB components handle system exceptions similarly, we'll focus on entity EJB components for the examples throughout this section, but keep in mind that most of the examples can also be applied to working with session EJB components.

Entity EJB components encounter RemoteExceptions when they refer to other EJB remote interfaces, NamingExceptions while looking up other EJB components, and SQLExceptions if they use bean-managed persistence (BMP). Checked system exceptions like these should be caught and thrown as either an EJBException or one of its subclasses. The original exception should be wrapped. Listing 4 shows a way of handling system exceptions that is congruent with EJB container behavior for system exceptions. By wrapping the original exception and rethrowing it in the entity EJB component, you ensure you can access the exception when you want to log it.
**Listing 4. A common way of handling system exceptions**

```java
try {
    OrderHome orderHome = EJBHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (NamingException ne) {
    throw new EJBException(ne);
} catch (SQLException se) {
    throw new EJBException(se);
} catch (RemoteException re) {
    throw new EJBException(re);
}
```

**Avoiding duplicate logging**

Normally, exception logging occurs in session EJB components. But what if the entity EJB components are accessed directly outside of the EJB tier? Then you have to log the exception in an entity EJB component and throw it. The problem here is that the caller has no way of knowing that the exception has been logged and will likely log it again, which will result in duplicate logging. More importantly, the caller has no way of accessing the unique ID generated during the initial logging. Any logging without a mechanism to cross-reference is no good.

Consider this worst-case scenario: a method, `foo()`, in an entity EJB component is accessed by a stand-alone Java application. The same method is accessed in a session EJB method, called `bar()`. A Web-tier client invokes the method `bar()` on the session EJB component, and also logs the exceptions. If an exception occurs in the entity EJB method `foo()` when the session EJB method `bar()` is invoked from the Web-tier, the exception will have been logged in three places: first in the entity EJB component, then in the session EJB component, and finally in the Web tier. And not one of the stack traces can be cross-referenced!

Fortunately, addressing these problems is fairly easy to do in a generic way. All you need is a mechanism for the caller to:

- Access the unique ID
- Find out if the exception has already been logged

You can subclass `EJBException` to store this information. Listing 5 shows the `LoggableEJBException` subclass:

**Listing 5. LoggableEJBException -- a subclass of EJBException**

```java
public class LoggableEJBException extends EJBException {
    protected boolean isLogged;
    protected String uniqueID;

    public LoggableEJBException(Exception exc) {
        super(exc);
        isLogged = false;
        uniqueID = ExceptionIDGenerator.getExceptionID();
    }
    ..
    ..
}
```
The class \texttt{LoggableEJBException} has an indicator flag (\texttt{isLogged}) to check if the exception has been logged. Whenever you catch a \texttt{LoggableEJBException}, see if the exception has already been logged (\texttt{isLogged == false}). If it is false, log the exception and set the flag to true.

The \texttt{ExceptionIDGenerator} class generates the unique ID for the exception using the current time and host name of the machine. You can use fancy algorithms to generate the unique ID if you like. If you log the exception in the entity EJB component, it will not be logged elsewhere. If you throw the \texttt{LoggableEJBException} in the entity EJB component without logging, it will be logged in the session EJB component but not in the Web tier.

Listing 6 shows Listing 4 rewritten using this technique. You can also extend the \texttt{LoggableException} to suit your needs (by assigning an error code to the exceptions and so on).

\textbf{Listing 6. Exception handling with LoggableEJBException}

```java
try {
    OrderHome orderHome = EJBAHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (NamingException ne) {
    throw new LoggableEJBException(ne);
} catch (SQLException se) {
    throw new LoggableEJBException(se);
} catch (RemoteException re) {
    Throwable t = re.detail;
    if (t != null && t instanceof Exception) {
        throw new LoggableEJBException((Exception) re.detail);
    } else {
        throw new LoggableEJBException(re);
    }
}
```

\textbf{Logging a RemoteException}

From Listing 6 you can see that naming and SQL exceptions are wrapped in \texttt{LoggableEJBException} before being thrown. But \texttt{RemoteException} is handled in a slightly different -- and slightly more labor intensive -- manner.

\textbf{System exceptions in session EJB components}

If you decide to log session EJB exceptions, use the logging code shown in Listing 7, otherwise, throw the exception as shown in Listing 6. You should note that there is one way that session EJB components might handle exceptions differently from entity EJB components: because most EJB systems are accessible only from the Web tier and a session EJB can serve as a facade to the EJB tier, it is actually possible to defer the logging of session EJB exceptions to the Web tier.

It's different because in a \texttt{RemoteException}, the actual exception will be stored in a public attribute called \texttt{detail} (which is of type \texttt{Throwable}). Most of the time, this public attribute holds an exception. If you call a \texttt{printStackTrace} on a \texttt{RemoteException}, it prints the stack trace of the exception itself, in addition to the stack trace of the detail. You don't need the stack trace of the \texttt{RemoteException} as such.

To isolate your application code from the intricacies such as that of \texttt{RemoteException}, these lines are refactored into a class called \texttt{ExceptionLogUtil}. With this class, all you need to do
is Call ExceptionLogUtil.createLoggableEJBException(e) whenever you need to create a LoggableEJBException. Note that the entity EJB component doesn't log the exceptions in Listing 6; however, this solution works even if you decide to log the exceptions in the entity EJB components. Listing 7 shows exception logging in an entity EJB component:

**Listing 7. Exception logging in an entity EJB component**

```java
try {
    OrderHome orderHome = EJBHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (RemoteException re) {
    LoggableEJBException le = ExceptionLogUtil.createLoggableEJBException(re);
    String traceStr = StackTraceUtil.getStackTrace(le);
    Category.getInstance(getClass().getName()).error(le.getUniqueID() + " : " + traceStr);
    le.setLogged(true);
    throw le;
}
```

What you see in Listing 7 is a foolproof exception logging mechanism. Upon catching a checked system exception, create a new LoggableEJBException. Next, get the stack trace for the LoggableEJBException as a string, using the class StackTraceUtil. Then log the string as an error, using the Log4J category.

**How the StackTraceUtil class works**

In Listing 7, you saw a new class called StackTraceUtil. Because Log4J can only log String messages, this class addresses the problem of converting stack traces into Strings. Listing 8 illustrates the workings of the StackTraceUtil class:

**Listing 8. StackTraceUtil class**

```java
public class StackTraceUtil {
    public static String getStackTrace(Exception e) {
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        return sw.toString();
    }
}
```

The default printStackTrace() method in java.lang.Throwable logs an error message to the System.err. Throwable also has an overloaded printStackTrace() method to log to a PrintWriter or a PrintStream. The above method in StackTraceUtil wraps the StringWriter within a PrintWriter. When the PrintWriter contains the stack trace, it simply calls toString() on the StringWriter to get a String representation of the stack trace.

**EJB exception handling for the Web tier**

In a Web tier design, it is often easier and more efficient to place your exception logging mechanism on the client side. For this to work, the Web tier must be the only client for the EJB tier. In addition, the Web tier must be based on one of the following patterns or frameworks:
• **Patterns**: Business Delegate, FrontController, or Interception Filters
• **Frameworks**: Struts or any similar MVC framework that contains hierarchies

Why should exception logging take place on the client side? Well, first, the control hasn't passed outside of the application server yet. The so-called client tier, which is composed of JSP pages, servlets or their helper classes, runs on the J2EE app server itself. Second, the classes in a well-designed Web tier have a hierarchy (for example, in the Business Delegate classes, Intercepting Filter classes, http request handler classes, JSP base class, or in the Struts Action classes) or single point of invocation in the form of a FrontController servlet. The base classes of these hierarchies or the central point in Controller classes can contain the exception logging code. In the case of session EJB-based logging, each of the methods in the EJB component must have logging code. As the business logic grows, so will the number of session EJB methods, and so will the amount of logging code. A Web-tier system will require less logging code. You should consider this alternative if you have co-located Web tier and EJB tiers and you don't have a requirement to support any other type of client. Regardless, the logging mechanism doesn't change; you can use the same techniques as described in previous sections.

**Real-world complexities**

Until now you’ve seen the exception-handling techniques in straightforward scenarios for session and entity EJB components. Some combinations of application exceptions can, however, be more confusing and open to interpretation. Listing 9 shows an example. The ejbCreate() method of the OrderEJB tries to get a remote reference on a CustomerEJB, which results in a FinderException. Both OrderEJB and CustomerEJB are entity EJB components. How should you interpret this FinderException in ejbCreate()? Do you treat it as an application exception (because the EJB spec defines it as standard application exception), or do you treat it as system exception?

**Listing 9. FinderException in ejbCreate() method**

```java
public Object ejbCreate(OrderValue val) throws CreateException {
    try {
        if (value.getItemName() == null) {
            throw new CreateException("Cannot create Order without a name");
        }
        String custId = val.getCustomerId();
        Customer cust = customerHome.findByPrimaryKey(custId);
        this.customer = cust;
    } catch (FinderException ne) {
        //How do you handle this Exception ?
    } catch (RemoteException re) {
        //This is clearly a System Exception
        throw ExceptionLogUtil.createLoggableEJBException(re);
    }
    return null;
}
```

While there is nothing to prevent you from treating this as an application exception, it's better to treat the FinderException as a system exception. Here's why: EJB clients tend to treat EJB components as black boxes. If the caller of the createOrder() method gets a FinderException, it doesn't make sense to the caller. The fact that OrderEJB is trying to set a customer remote reference is transparent to the caller. From the client perspective, the failure simply means that the order can't be created.
Another example of this type of scenario is one where a session EJB component attempts to create another session EJB and gets a `CreateException`. A similar scenario is one where an entity EJB method tries to create a session EJB component and gets a `CreateException`. Both of these exceptions should be treated as system exceptions.

Another challenge you may encounter is one where a session EJB component gets a `FinderException` in one of its container callback methods. You have to handle this type of scenario on a case-by-case basis. You may decide to treat the `FinderException` as an application exception or a system exception. Consider the case in Listing 1, where a caller invokes a `deleteOldOrder` method on a session EJB component. Instead of catching the `FinderException`, what if we throw it? In this particular case, it seems logical to treat the `FinderException` as a system exception. The reasoning here is that session EJB components tend to do a lot of work in their methods, because they handle workflow situations and act as a blackbox to the caller.

On the other hand, consider a scenario where a session EJB is handling an order placement. To place an order, the user must have a profile -- but this particular user doesn't have one. The business logic may want the session EJB to explicitly inform the user that her profile is missing. The missing profile will most likely manifest as a `javax.ejb.ObjectNotFoundException` (a subclass of the `FinderException`) in the session EJB component. In this case, the best approach is to catch the `ObjectNotFoundException` in the session EJB component and throw an application exception, letting the user know that her profile is missing.

Even with good exception handling strategies, there is another problem that often occurs in testing, and more importantly in production. Compiler and runtime optimizations can change the overall structure of a class, which can limit your ability to track down an exception using the stack trace utility. This is where code refactoring comes to your rescue. You should split large method calls into smaller, more manageable chunks. Also, whenever possible, have your exceptions typed as much as needed; whenever you catch an exception, you should be catching a typed exception rather than a catch-all.

**Conclusion**

We've covered a lot of ground in this article, and you may be left wondering if all the up-front design we've discussed is worth it. It is my experience that even in a small- to medium-sized project, the effort pays for itself in the development cycle, let alone the testing and production cycles. Furthermore, the importance of a good exception-handling architecture cannot be stressed enough in a production system, where downtime can prove devastating to your business.

I hope you will benefit from the best practices demonstrated in this article. To follow up on some of the information presented here, check the listings in the Related topics section.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-ejbexcept.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- You can find more information on the EJB architecture by reading the EJB specification from Sun Microsystems.
- The Struts framework is another gem from the Jakarta project. Struts is based on the MVC architecture and provides a clean decoupling of a system's presentation layer from its business-logic layer.
- Sun's J2EE Patterns web site focuses on patterns, best practices, design strategies, and proven solutions using J2EE technologies.
- The tutorial "Java design patterns 101" (developerWorks, January 2002) is an introduction to design patterns. Find out why patterns are useful and important for object-oriented design and development, and how patterns are documented, categorized, and cataloged. The tutorial includes examples of important patterns and implementations.
- See the developerWorks tutorials page for a complete listing of free tutorials from the developerWorks Java technology zone.