Writing applications that integrate disparate systems is a challenge when it comes to handling unexpected events. In a single system that you fully control, you can handle these events and recover. But systems that are integrated over the network have additional risks: the network connection could be broken, a remote system might not respond in a timely manner, or it might even fail for no apparent reason. Even on your local server, unexpected events can occur, such as the server's disk filling up or running out of memory. Regardless which errors occur, your application should be prepared to handle them.

In these situations log files are often the only evidence of the unexpected event, so logging is important. Camel has extensive support for logging and handling errors to ensure your integrated application can continue to operate.

**Understanding error handling**

Before jumping into the world of error handling with Camel, we need to take a step back and look at errors more generally. Specifically, we need to look at the two main categories of errors, recoverable and irrecoverable, and we need to look at where and when error handling starts because there are some prerequisites.

**Recoverable and irrecoverable errors**

When it comes to errors, we can categorize them into recoverable and irrecoverable errors, as illustrated in figure 1.

An *irrecoverable error* is an error that remains an error no matter how many times you try to perform the same action. In the integration space, that could mean trying to access a database table that does not exist, which would cause the JDBC driver to throw back a `SQLException`.

A *recoverable error*, on the other hand, is a temporary error that might not cause a problem on the next attempt. A good example of such an error is a problem with the network connection resulting in a `java.io.IOException`. On a subsequent attempt, the network issue could be resolved and your application could continue to operate.
Figure 1 Errors can be categorized as either recoverable or irrecoverable. Irrecoverable errors continue to be errors on subsequent attempts; recoverable errors may be quickly resolved on their own.

In our daily lives as Java developers, we don’t encounter this division of errors into recoverable and irrecoverable very often. Generally, exception handling code uses one of the two patterns illustrated in the following two code snippets.

The first snippet illustrates a common error-handling idiom, where all kinds of exceptions are considered irrecoverable and we give up immediately throwing the exception back to the caller, very often wrapped.

```java
public void handleOrder(Order order) throws OrderFailedException {
    try {
        service.sendOrder(order);
    } catch (Exception e) {
        throw new OrderFailedException(e);
    }
}
```

The next snippet improves this situation by adding a bit of logic to handle redelivery attempts before eventually giving up:

```java
public void handleOrder(Order order) throws OrderFailedException {
    boolean done = false;
    int retries = 5;
    while (!done) {  #1
        try {
            service.sendOrder(order);
            done = true;
        } catch (Exception e) {
            if (--retries == 0) {
                throw new OrderFailedException(e);
            }
        }
    }
}
```

Around the invocation of the service is the logic that attempts redelivery (#1), in case an error occurred. After five attempts, we give up and throw the exception.

What the preceding example lacks is logic to determine whether the error is recoverable or irrecoverable and to react accordingly. In the recoverable case, we could try again and, in the irrecoverable case, we could give up immediately and rethrow the exception.

In Camel, a recoverable error is represented as a plain `Exception` that can be set or accessed from `org.apache.camel.Exchange` using one of the following two methods:

```java
void setException(Exception cause);
```

or

```java
Exception getException();
```
An irrecoverable error is represented as a fault message that can be set or accessed from `org.apache.camel.Exchange`. For example, to set "Unknown customer" as a fault message, you will have to do the following:

```java
Message fault = Exchange.getOut();
fault.setFault(true);
fault.setBody("Unknown customer");
```

So why are the two types of errors represented differently? There are two reasons. First, the Camel API was designed around the Java Business Integration (JBI) concepts, which include a fault message concept. Second, Camel has error handling build into its core so, whenever an exception is thrown back to Camel, it catches it and sets the thrown exception on the `Exchange` as a recoverable error, as illustrated here:

```java
try {
    processor.process(exchange);
} catch (Exception e) {
    exchange.setException(e);
}
```

Using this pattern allows Camel to catch and handle all exceptions that are thrown. Camel's error handling can determine how to deal with the errors—retry, propagate the error back to the caller, or do something else.

End users of Camel can set irrecoverable errors as fault messages, and Camel can react accordingly and stop routing the message.

Now that you've seen recoverable and irrecoverable errors in action, let's summarize how they are represented in Camel:

- Exceptions are represented as recoverable errors.
- Fault messages are represented as irrecoverable errors.

Now let's look at when and where Camel's error handling applies.

### Where Camel's error handling applies

The Camel error handling does not apply everywhere. To understand why, take a look at figure 2.

![Figure 2 Camel error handling only applies within the lifecycle of an Exchange.](http://en.wikipedia.org/wiki/Java_Business_Integration)

Figure 2 shows a very simple route that translates files. We have a file consumer and a file producer as the input and output facilities and, in between, we have the Camel routing engine, which routes messages encompassed in an `Exchange`. It's during the lifecycle of this `Exchange` that the Camel error handling applies. That leaves a little room on the input side where this error handling cannot operate—the file consumer must be able to successfully read the file, instantiate the `Exchange`, and start the routing before the error handling can function. This applies to any kind of Camel consumer.

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/ibsen/](http://www.manning.com/ibsen/)
So what happens if the file consumer cannot read the file? The answer is component specific, and each Camel component must deal with this in its own way. Some components will ignore and skip the message, others will retry a number of times, and still others will gracefully recover.

NOTE
A number of Camel components provide minor error-handling features: file, ftp, mail, ibatis, rss, atom, jpa, and snmp. These components are based on the ScheduledPollConsumer class, which offers a pluggable PollingConsumerPollStrategy that you can use to create your own error-handling strategy. You can learn more about this on the Camel website, at http://camel.apache.org/polling-consumer.html.

That’s enough background information—let’s dig into how error handling in Camel works. In the next section, we look at the different error handlers Camel provides.

Error handlers in Camel
We’ve just learned that Camel regards all exceptions as recoverable and stores them on the exchange using the setException(Exception cause) method. This means error handlers in Camel will only react to exceptions set on the Exchange. They will not react if an irrecoverable error has been set as a fault message.

The rule of thumb is that error handlers in Camel only trigger when exchange.getException() != null.

Camel provides a range of error handlers. They’re listed in table 1.

Table 1 The error handlers provided in Camel

<table>
<thead>
<tr>
<th>Error handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultErrorHandler</td>
<td>This is the default error handler that is automatically enabled, in case no other has been configured.</td>
</tr>
<tr>
<td>DeadLetterChannel</td>
<td>This is the error handler that implements the Dead Letter Channel and Dead Letter Queue EIP patterns.</td>
</tr>
<tr>
<td>TransactionErrorHandler</td>
<td>This is a transaction-aware error handler extending the default error handler.</td>
</tr>
<tr>
<td>NoErrorHandler</td>
<td>This handler is used to disable error handling altogether.</td>
</tr>
<tr>
<td>LoggingErrorHandler</td>
<td>This error handler just logs the exception.</td>
</tr>
</tbody>
</table>

At first glance, having five error handlers may seem overwhelming, but you will learn that the default error handler is used in most cases.

The first three error handlers in table 1 all extend the RedeliveryErrorHandler class. That class contains the majority of the error handling logic which the first three error handlers all leverage. The latter two has limited functionality and therefore does not extend that class. We’ll look at each of the error handlers in table 5.1 in turn.

Default error handler
Camel is preconfigured to use the DefaultErrorHandler, which covers most uses cases. To understand it, consider the following example route:

```java
from("direct:newOrder")
    .beanRef("orderService, "validate")
    .beanRef("orderService, "store");
```

For Source Code, Sample Chapters, the Author Forum and other resources, go to http://www.manning.com/ibsen/
The default error handler is preconfigured and doesn’t need to be explicitly declared in the route. So what happens if an exception is thrown from the `validate` method on the order service bean? To answer this, we need to dive into the inner processing of Camel, where the error handler lives. In every Camel route, there is a `Channel` that sits between each node in the route graph, as illustrated in figure 3.

![Figure 3 A detailed view of a route path, where channels act as controllers between the processors](http://www.manning.com/ibsen/)

The `Channel` is woven in between each node in the route path, which ensures it can act as a controller that monitors and controls the routing at runtime. This is the feature that allows Camel to enrich the route with error handling, message tracing, interceptors, and much more. For now, you just need to know that this is where the error handler lives.

Turning back to our route, imagine that an exception was thrown from the order service bean during invocation of the `validate` method. In figure 3, the processor (#3) would throw an exception, which would then be propagated back to the previous channel (#2), where the error handler would catch it. This gives Camel the chance to react accordingly. For example, Camel could try again (redeliver), or it could route the message to another route path (detour using exception policies), or it could simply give up and propagate the exception back to the caller.

And this is what happens with the default settings — Camel will propagate the exception back to the caller.

The default error handler is configured with these default settings:

- No redelivery
- Exceptions are propagated back to the caller

These settings match what happens when working with exceptions in Java, so Camel’s behavior won’t surprise Camel end users.

Let’s continue with the next error handler, the dead letter channel.

**Dead letter channel error handler**

The dead letter channel error handler is very similar to the default error handler except for the following differences:

- The dead letter channel is the only error handler that supports a dead letter queue.
- Unlike the default error handler, the dead letter channel will, by default, handle exceptions and send them to the dead letter queue.
- The dead letter channel supports using the original input message when a message is moved to the dead letter queue.

Let’s look at each of these in a bit more detail.

**The Dead Letter Queue**

The Dead Letter Channel is an error handler that implements the principles of the Dead Letter Channel pattern in the EIP book[^2]. This pattern states that, if a message cannot be processed or delivered, it should be moved to a dead letter channel. Figure 4 illustrates this pattern.

[^2]: http://www.enterpriseintegrationpatterns.com/
As you can see, the consumer (#1) consumes a new message, which is supposed to be routed to the processor (#3). The channel (#2) controls the routing between (#1) and (#3) and, if the message can't be delivered to (#3), the channel invokes the Dead Letter Channel error handler, which moves the message to the dead letter channel (#4). This keeps the message safe in the dead letter channel and allows the application to continue operating.

This pattern is often used with messaging. Instead of allowing a failed message to block new messages from being picked up, the message is simply moved to a dead letter queue to get it out of the way.

The same idea applies to the dead letter channel error handler in Camel. This error handler also has an associated dead letter queue, which is based on an endpoint, allowing you to use any Camel endpoint you choose. For example, you can use a database or a file or just log the failed messages.

When you choose to use the dead letter channel error handler, you must configure the dead letter endpoint so the handler knows where to move the failed messages. This is done a bit differently in Java DSL and Spring XML. For example, here is how we would log the message at ERROR level in Java DSL:

```java
errorHandler(deadLetterChannel("log:dead?level=ERROR"));
```

And here is how we would do it in Spring XML:

```xml
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
  deadLetterUri="log:dead?level=ERROR"/>
```

Now, let's look at how the dead letter channel error handler handles exceptions when it moves the message to the dead letter endpoint.

**Handling exceptions by default**

By default, Camel handles exceptions by suppressing them; it removes the exceptions from the Exchange and stores them as properties on the Exchange. After a message has been moved to the dead letter endpoint, Camel stops routing the message and the caller regards it as processed.

When a message is moved to the dead letter channel, you can obtain the exception from the exchange using the Exchange.CAUSED_EXCEPTION property.

```java
Exception e = exchange.getProperty(Exchange.CAUSED_EXCEPTION,
  Exception.class);
```

Next, let's look at using the original message.

**Using original message with dead letter channel**

Suppose you have a route in which the message goes through a series of processing steps, each altering a bit of the message before it reaches its final destination, as in the following code:

```java
errorHandler(deadLetterChannel("jms:queue:dead"));
from("jms:queue:inbox")
  .beanRef("orderService", "decrypt")
  .beanRef("orderService", "validate")
```

Figure 4 The Dead Letter Channel EIP pattern moves failed messages to a dead letter channel.
Now, imagine that an exception occurs at the validate method, and the dead letter channel error handler moves the message to the dead letter channel. Suppose a new message arrives and an exception occurs at the enrich method, and this message is also moved to the same dead letter channel. If we want to retry these messages, can we just drop them in the inbox queue?

In theory, we could do this but the messages that were moved to the dead letter channel no longer match the messages that originally arrived at the inbox queue—they were altered as the messages were routed. What we want instead is for the original message content to be moved to the dead letter channel. The useOriginalMessage option instructs Camel to use the original message when it moves messages to the dead letter channel. You configure the error handler to use the useOriginalMessage option as follows:

```java
ErrorHandler(deadLetterChannel("jms:queue:dead").useOriginalMessage());
```

And in Spring XML you would do:

```xml
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
    deadLetterUrl="jms:queue:dead" useOriginalMessage="true"/>
```

Let’s move on to the transaction error handler.

**Transaction error handler**

The transaction error handler is built on top of the default error handler and offers the same functionality tailored to support transacted routes. The remaining two error handlers are seldom used and are much simpler.

**No error handler**

The NoErrorHandler is used to disable error handling. Camel’s current architecture mandates that an error handler must be configured so, if we want to disable error handling, we need to provide an error handler that basically is an empty shell with no real logic. That's the NoErrorHandler.

Finally, let’s look at the logging error handler.

**Logging error handler**

The logging error handler logs the failed message along with the caused exception. The logger uses standard log format from log kits such as Log4j, commons logging, or the Java Util Logger.

Camel will, by default, log the failed message and the caused exception using the log name org.apache.camel.processor.LoggingErrorHandler at ERROR level. You can, of course, customize this.

That covers the five error handlers provided with Camel. Let’s now look at the major features these error handlers provide.

**Features of the error handlers**

The default, dead letter channel, and transaction error handlers are all built on the same base, org.apache.camel.processor.RedeliveryErrorHandler, so they all have several common major features. These features are listed in table 2.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redelivery policies</td>
<td>Redelivery policies allow you to define policies for whether or not redelivery should be attempted. The policies also define settings such as maximum number of redelivery attempts, delays between attempt, and so on.</td>
</tr>
</tbody>
</table>

For Source Code, Sample Chapters, the Author Forum and other resources, go to [http://www.manning.com/ibsen/](http://www.manning.com/ibsen/)
**Scope**
Camel error handlers have two possible scopes: context (high level) and route (low level). The context scopes allow you to reuse the same error handler for multiple routes, whereas route scope is per route only.

**Exception policies**
Exception policies allow you to define special policies for individual exceptions. This is extremely useful for defining policies that are only applied for certain exceptions.

**Error handling**
Option to specify whether or not the error handler should handle the error. This allows you to let the error handler deal with the error or to leave it for the caller to handle.

**Summary**
In this article, we saw how recoverable and irrecoverable errors are represented in Camel. We also looked at all of the provided error handlers, focusing on the most important of them. We also saw how Camel can control how exceptions should be dealt with using redelivery policies to set the scene and using exception policies to handle specific exceptions differently.