Table of Contents

Exception handling is very important for applications. It's necessary to visualize unexpected exceptions and to handle expected exceptions. Exception handling should be easy, centralized, and shouldn't blow up the application code. Isn't it boring to catch exceptions after unimportant code blocks?

This could be boilerplate code:

```
try
{
    Email email = prepareEmail(dataBook);

    sendEmail(email);
}
catch (MessagingException me)
{
    showError(me);
}
```

If you catch exceptions to show error messages, it's boilerplate code. If you catch exceptions to do specific error handling, it's not. Usually an application has both use-cases.

With JVx, we tried to reduce code for exception handling to a bare minimum. We delegate “all” framework exceptions to an ExceptionHandler. The ExceptionHandle is more or less a delegator.

The ExceptionHandler doesn't visualize an exception. It forwards exceptions to a listener list and is responsible for logging. The application itself is responsible for visualization because only the application knows how to visualize an exception - as a popup, as a status in a specific area, etc.

The standard JVx application already supports visualization of exceptions because it's a registered exception listener. The method

```
public void handleException(Throwable pThrowable)
{
    error(pThrowable);

    try
    {
        if (error == null || error.isClosed())
        {
            error = createError();
            error.eventWindowClosed().addListener(this, "doErrorClosed");

            configureFrame(error);

            invokeLater(this, "showErrorDialog");
        }

        error.addError(pThrowable);
    }
    catch (Throwable th)
    {
```

```
        //forwarding to the launcher is the only possibility
        getLauncher().handleException(th);

        getLauncher().handleException(pThrowable);
    }
}
```

will do the job. Simply override the method to do what you want or register your own listener and remove the application and exception listener.

The default implementation makes it super easy to show exceptions without additional code. Suppose we have the following action:

```
UIButton button = new UIButton("Send E-Mail");
button.eventAction().addListener(this, "doSendEmail");

public void doSendEmail()
{
    getConnection().callAction("sendEmail");
}
```

The callAction method will throw throwable because it's a remote procedure call and all remote procedure calls will throw throwable instead of an exception. JVx doesn't define custom exceptions because Java has enough, and it wasn't necessary to wrap exceptions for remote calls. The application is able to define custom exceptions in remote procedure calls, but JVx doesn't wrap exceptions. It's nice for an application developer to get the thrown exception instead of unwrapping a caught exception to get the original exception.

To make our action working, we could do the following:

```
public void doSendEmail() throws Throwable
{
    getConnection().callAction("sendEmail");
}
```

It's good enough to add throws throwable to the method definition. The event mechanism of JVx will do the rest because it forwards exceptions to ExceptionHandler:

```
@Override
public Object dispatchEvent(Object... pEventParameter)
{
    try
    {
        return super.dispatchEvent(pEventParameter);
    }
    catch (Throwable pThrowable)
    {
        ExceptionHandler.raise(pThrowable);
        return null;
    }
```

```
}
```

If you need custom exception handling, simply do it:

```
public void doSendEmail() throws Throwable
{
    try
    {
        getConnection().callAction("sendEmail");
    }
    catch (MessagingException me)
    {
        showEmailProblem(me);
    }
}
```

This will handle MessagingException and all other exceptions will be handled from the default implementation. It's also possible to ignore the default implementation:

```
public void doSendEmail()
{
    try
    {
        getConnection().callAction("sendEmail");
    }
    catch (MessagingException me)
    {
        showEmailProblem(me);
    }
    catch (Throwable th)
    {
        handleProblem(th);
    }
}
```

Please, centralize your custom exception handling to keep things simple and to reduce complexity.

It's also possible to use standard exception visualization from your custom code:

```
public void doSendEmail()
{
    try
    {
        getConnection().callAction("sendEmail");
    }
    catch (MessagingException me)
    {
        showEmailProblem(me);
    }
    catch (Throwable th)
    {
```

```
            ExceptionHandler.show(th);
        }
    }
```

The ExceptionHandler has two relevant methods for your exceptions:

```
public static void raise(Throwable pThrowable)
public static void show(Throwable pThrowable)
```

The raise method will interrupt code execution while show will continue.

The ExceptionHandler knows one specific exception which will be handled separately. It's the SilentAbortException. If you throw a SilentAbortException, no listener will be notified about the exception. This also means that your application doesn't visualize the exception.