

Powerful Exceptions: an Overview

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W5S04



<http://www.pharo.org>



Exceptions

- Really powerful
- Can be resumed, restarted, and signaled as new exception
- Two important classes:
 - Error and Notification
- For more complete reference, read Deep into Pharo



What You Will Learn

- To raise and trap exceptions
- Some nice helper methods



API Overview

- Installing an handler

```
[ doSomething ] on: ExceptionClass do: [ :ex | something ]
```

- Raising an exception

```
anException signal
```

- defaultAction is executed when an exception occurs and it is not trapped

Convenient messages:

- ensure:, ifCurtailed:



Catching Example

```
[ do something ] on: ExceptionClass do: [ :ex | something ]
```

Example:

```
| x y |  
x := 7.  
y := 0.  
[ x / y ]  
  on: ZeroDivide  
  do: [ :exception | Transcript show: exception description; cr.  
      0 ]  
> 0
```

Signaling an Exception

To raise an exception:

- create an instance of exception
- send it messages `signal` or `signal: aMessage`

(`AuthorNameRequest` new initialAnswer: 'Stef') signal
(`Warning` new messageText: 'Pay attention') signal



Signaling an Exception

Usually classes propose a shortcut

`OutOfMemory` signal.

`Warning` signal: 'description of the exception'



Testing That an Exception Occurs

SUnit offers `should:raise:` and `shouldnt:raise:` to check occurrence of exceptions.

```
testNameOfMonth
```

```
self assert: (Date nameOfMonth: 1) equals: #January.
```

```
self
```

```
  shouldnt: [ Date nameOfMonth: 2 ]
```

```
  raise: SubscriptOutOfBounds.
```

```
self
```

```
  should: [ Date nameOfMonth: 13 ]
```

```
  raise: SubscriptOutOfBounds.
```


Kinds of Exceptions

- Error: all errors (subscript, message not understood, division by zero)
- Halt: to stop the execution (and get a debugger)
- Notification: non fatal exceptions (deprecation, warning, timeout)
- UnhandledError: when an error occurs and that it is *not* trapped



Exceptions are Real Objects

When you send an unknown message Point new
strangeAndBizarre

```
ProtoObject >> doesNotUnderstand: aMessage
```

```
^ MessageNotUnderstood new  
  message: aMessage;  
  receiver: self;  
  signal
```



Deprecation

To support API migration, Pharo uses deprecation. When the deprecation setting is on, a warning is raised when a deprecated method is executed.

```
MenuItem >> title: aString  
"Add a title line at the top of this menu."  
self deprecated: 'Use method addTitle: instead' on: '29  
september' in: #Pharo40.  
self addTitle: aString
```



Deprecation Implementation Use

Create an instance of Deprecation and signal it

```
deprecated: anExplanationString on: date in: version  
"Warn that the sending method has been deprecated"  
(Deprecation  
  method: thisContext sender method  
  explanation: anExplanationString  
  on: date  
  in: version) signal
```

Exception Sets

```
[ do some work ]  
  on: ZeroDivide, Warning  
  do: [ :ex | what you want ]
```

Or

```
| exceptionSets |  
exceptionSets := ExceptionSet with: ZeroDivide with: Warning.  
[do some work]  
  on: exceptionSets  
  do: [ :ex | what you want ]
```



A Nice Helper: ensure:

- How to ensure that an expression is **always executed** (even if the program fails before)?
- [doSomething] ensure: [alwaysExecuteThis]

spyOn: aBlock

"Profile system activity during execution of aBlock."

self startProfiling.

aBlock ensure: [**self** stopProfiling]



Another nice Helper ifCurtailed:

- How to ensure that an expression is **executed only if the program fails** or returns?
- [doSomething] ifCurtailed: [onProblem]

wait

"Schedule this Delay, then wait on its semaphore. The current process will be suspended for the amount of time specified when this Delay was created."

self schedule.

[delaySemaphore wait] ifCurtailed: [self unschedule]



Exception Lookup

- Each process has its own exception environment: an ordered list of active handlers
- Process starts with an empty list
- [aaaa] on: Error do: [bbb] **adds** Error,bbb to the beginning of the list
- When an exception is signaled, the system sends a message to the first handler
 - If the handler cannot handle the exception, the next one is asked
 - If no handler can handle the exception, then the default action is performed



Handling Exception

Just for your information ;)

Within an handler [aaa] on: anExceptionClass do: [anHandler], we can:

- **Return** an alternative result for the protected block (return:)
- **Retry** the protected block or a different block (retryUsing:)
- **Resume** the protected block at the failure point (resume:)
- **Pass** the caught exception to the enclosing handler (pass)
- **Resignal** a different exception (resignalAs:)



Returning From an Exception

```
[ Notification signal. 'Value from protected block' ]  
on: Notification  
do: [ :ex |ex return: 'Value from handler' ]  
  
> 'Value from handler'
```

We return a different string on normal or notification



Resuming from Resumable Exception

Warning, Notification and subclasses are resumable

```
[ Notification signal. 'Value from protected block' ]  
  on: Notification  
  do: [ :ex | ex resume: 'Value from handler' ]  
  
> 'Value from protected Block'.
```

- Notification signal raises an exception
- exception is handled
- resume: restores the context and the value returned normally as if the notification did not occur

What You Should Know

- Exceptions are powerful in Pharo.
- Offer a simple API

Raising

```
anException signal
```

Installing:

```
[ doSomething ] on: ExceptionClass do: [ :ex | something ]
```

- Helpers
 - [doSomething] ensure: [alwaysDoThis]
 - [doSomething] ifCurtailed: [onProblem]

A course by



and



in collaboration with



Inria 2016

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>