Exceptions en Smalltalk

S.Ducasse – stephane.ducasse@free.fr
A. Bergel – alexandre@bergel.eu

Les applications réelles doivent faire face à des situations exceptionnelles. Ces situations exceptionnelles peuvent être d'ordre matériel (problème d'écriture sur un disque...) ou logiciel (division par zéro, fichier inexistant...). Pour cela les langages de programmation offrent la possibilité de spécifier des exceptions et comment y répondre. Smalltalk offre un mécanisme d'exception au demeurant simple mais très puissant. Il est ainsi possible non seulement de déclarer, lever et capturer des exceptions comme dans la plupart des langages mais aussi de republier de nouvelles exceptions à la place de l'exception originale et dans certains cas de relancer l'exécution d'un programme comme si l'exception originale ne s'était pas produite. Dans cet article nous montrons ces différents aspects ainsi que des messages importants tels que ensure: qui permet de garantir l'exécution d'une séquence d'instructions. En Smalltalk, plusieurs frameworks de gestion d'exception, souvent incompatibles entre eux, existèrent avant le standard ANSI de 1996. Squeak suit ce standard et c'est cela que nous présentons dans cet article. Dans un second article nous montrons les aspects avancés. Comme convention, chaque nom de méthode et de message est précédé par un #.

Garantie d'exécution

Prenons comme exemple la gestion des fichiers. Il s'agit d'une utilisation complexe de ressources pouvant engendrer des bugs difficile à déceler. Un exemple typique est le fait d'avoir des fichiers ouverts que l'on ne peut plus atteindre. Ceci est dangereux car le nombre de fichiers ouverts est souvent limités par la plate-forme d'exécution et il est alors possible qu'une application se détériore juste car elle ne peut plus ouvrir de fichiers.

Le message ensure: permet de spécifier qu'une suite d'actions doit impérativement être suivie d'une autre suite d'actions. Et ceci même si la première produit des comportements anormaux. Le receveur du message doit être un bloc (délimité par des [], un bloc est une méthode anonyme ou lambda expression). Regardons un exemple. Imaginons que nous voulions créer un fichier image pharo.gif à partir d'une capture écran définie par l'utilisateur:

```
| writer |
[writer := GIFReadWriter on: (FileStream newFileNamed: 'pharo.gif').
writer nextPutImage: (Form fromUser)] ensure: [writer ifNotNil: [writer close]]
```

La méthode nextPutImage: de la classe GIFReadWriter convertit une Form (instance de la classe Form qui représente un morceau d'écran) en une image au format gif. Cette méthode écrit dans un flot qui ici est un fichier. La méthode nextPutImage: ne ferme pas la stream le flot dans laquelle elle écrit, par contre, nous devons nous assurer que si un problème survient ou non lors de l'écriture dans le fichier, nous fermons le fichier. Le message ensure: envoyé au bloc assure cela (pour les programmeurs Java parmis nous, ensure: est l'équivalent du try{} finally{} en Java). Il faut noter que dans le cas où le newFileNamed: ne se déroule pas comme prévu, nous basculons dans le ensure: et la variable writer reste non initialisée (elle référence nil). Avant de faire un close nous devons donc nous assurer qu'elle n'est pas à nil.

Voici un autre exemple montrant l'utilisation de ensure: pour garantir que le curseur de la souris n'est changé que pendant l'exécution d'une séquence d'instructions et reprend bien sa forme originale une fois celle-ci terminée.

```
Sensor>>>showWhile: aBlock
        "While evaluating the argument, aBlock, make the receiver be the cursor
        shape."
        | oldcursor |
        oldcursor := Sensor currentCursor.
```

```
        self show.
        ^aBlock ensure: [oldcursor show]
```

Notez que le message ensure:, qui est envoyé à un bloc, rend comme résultat l'évaluation de aBlock. Notez qu'ensure: assure que le bloc passé en argument est exécuté indépendamment le code protégé se comporte de manière normale ou pas. Dans tous les cas de figures, le résultat du ensure: est le résultat de aBlock (et non pas de show!).

Le cas de ifCurtailed:
Le message ifCurtailed: est peu connu mais il est le pendant de ensure:. Il s'utilise de la même façon que le message ensure:. Il s'applique à un bloc comme receveur et un bloc comme argument. ifCurtailed: est utilisé pour des actions de nettoyage. L'argument (le second bloc) est exécuté si l'exécution du receveur produit une terminaison anormale de la méthode qui le contient. Cette fin anormale de l'exécution peut être due à une exception menant à l'abandon de l'exécution ou à un retour non local à l'intérieur du receveur du ifCurtailed:.

Voici des exemples, imaginons la méthode foo suivante

```
foo
        [^ 10 ] ifCurtailed: [Transcript show: 'Nettoye'].
        Transcript show: 'Jamais affiché'.
        ^ 33
```

L'expression ^ 10 à l'intérieur du bloc va amener à abandonner le reste de la méthode et à retourner la valeur 10. Les messages sur les deux dernieres lignes ne sont jamais exécutés. Par contre le bloc argument est exécuté et affiche le message. Ainsi si vous exécutez les expressions suivantes, vous

[^ 10] ifCurtailed: [Transcript show: 'blah'; cr] affiche blah dans le transcript.
[10] ifCurtailed: [Transcript show: 'blah'; cr] n'affiche rien.
[1 / 0] ifCurtailed: [Transcript show: 'blah'; cr] affiche le texte si on abandonne l'exécution en pressant sur le button abandon du débogueur.

Le message #ifCurtailed: pourrait être implanté en utilisant le message #ensure: comme suit comme indiqué par P. Bonzini le développeur de GNU Smalltalk.

```
ifCurtailed: curtailBlock
    | result curtailed |
    curtailed := true.
    [ result := self value. curtailed := false ] ensure: [
       curtailed ifTrue: [ curtailBlock value ] ].
    ^result
```

De manière similaire, #ensure: pourrait être implanté de la manière suivante en utilisant #ifCurtailed:

```
ensure: ensureBlock
    | result |
    result := self ifCurtailed: ensureBlock.
        "si nous arrivons ici alors execution n'a pas ete
        abandonne dont le ensureblock n'a pas ete execute "
    ensureBlock value.
    ^result
```

Nous venons de voir comment nous protéger de situation anormale sans pour autant polluer le flot de contrôle. Maintenant nous allons nous intéresser à comment exprimer une telle situation en utilisant les exceptions.

Les Exceptions: Principe

Sans mécanisme d'exception, la partie invoquant une certaine fonctionnalité doit s'assurer que le résultat rendu est le bon. Ainsi les programmes écrits avec des langages n'offrant pas de mécanisme d'exception se trouvent souvent remplis de tests que l'appelant se doit d'invoquer. Il s'agit des fameux codes d'erreurs. Les codes d'erreurs rendent la logique des applications complexe et souvent tarabiscotée. Il devient alors plus difficile d'identifier la logique principale de celle prenant en compte les erreurs. Et bien sûr le code devient fragile et délicat à maintenir. L'utilisation d'exceptions libère le code des tests d'erreurs.

L'idée d'un mécanisme d'exception est de clairement séparer le code principal de la gestion des cas d'erreurs mais surtout d'éliminer la nécessité de passer explicitement le contrôle du signaleur de l'erreur (le code qui détecte l'erreur) au code qui traite l'erreur (handler en anglais). Regardons de manière schématique la même fonctionnalité exprimée sans et avec exception. Voici par exemple, une méthode d'une vieille version de squeak.

```
ResourceManager class>>cacheResource: urlString inArchive: archiveName
        …
        fd := Project squeakletDirectory.
        file := fd oldFileNamed: self resourceCacheName
        file isOk
                ifTrue: [...]
                ifFalse: [ fd forceNewFileNamed: self resourceCacheName]
        …
```

Bien sûr la section testant les codes d'erreurs est souvent complexe à écrire. Voici la même fonctionnalité en utilisant une exception.

```
ResourceManager class>>cacheResource: urlString inArchive: archiveName
        …
        fd := Project squeakletDirectory.
        file := [fd oldFileNamed: self resourceCacheName]
                        on: FileDoesNotExistException
                        do:[:ex| fd forceNewFileNamed: self resourceCacheName].
        …
```

Étudions cet exemple : la fonctionnalité qui crée et écrit dans le fichier ne se soucie pas de savoir si le fichier est bien créé. C'est le bloc de récupération d'erreurs (qui « traite le problème ») qui en est responsable. Il y a une séparation claire. De même le passage du flot de contrôle est implicite. Le message #on:do: peut être expliqué de la manière suivante : évalue la séquence des instructions contenues dans le bloc receveur et en cas de problème laisse le second bloc argument analyser la situation et éventuellement la résoudre.

Un point de vocabulaire. La littérature donne des noms à ces deux blocs : d'une part signaleur ou *bloc protégé* pour la partie du programme qui détecte et signale une erreur et d'autre par *bloc de récupération* (handler en anglais) la partie qui traite le problème si celui-ci arrive et offre une solution. Ici le premier bloc est le bloc protégé et le bloc de récupération (handler)s est le bloc suivant le #do:.

Il est important de remarquer que le message #on:do: définit en quelque sorte la durée de validité du bloc de récupération. Ce bloc ne prendra effet que pour l'exécution du premier bloc. Une fois celui-ci exécuté normalement, le bloc de récupération n'aura plus de raison d'être. De plus un bloc de récupération est spécialisé par rapport à un type d'erreur. Ici le bloc de récupération ne sait traiter que les erreurs de type FileDoesNotExistException (ou des erreurs plus spécifiques comme nous le verrons bientôt).

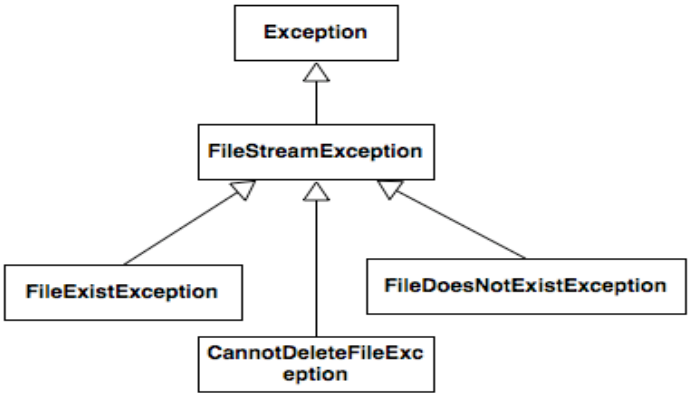
La norme définit le message on: exception do: handlerAction de la manière suivante : exécute le receveur

dans le contexte dans la portée du bloc de récupération d'exception. Le receveur est évalué de telle façon que si durant son évaluation, une exception est levée, alors handlerAction est évalué. Le résultat de l'évaluation du receveur (un bloc) est retournée.

Exception en Smalltalk

Les exceptions sont organisées en hiérarchies de classes: ainsi conceptuellement on peut considérer que FileDoesNotExistException, FileExistsException et CannotDeleteFileException sont toutes trois des exceptions spéciales du genre FileStreamException. Ainsi un bloc de récupération associé à une exception peut traiter une erreur plus générale.

En Smalltalk, les exceptions sont des objets comme tous les autres aspects du systèmes. Les exceptions sont instances de classes d'exceptions qui sont organisées en hiérarchies de spécialisation. Ainsi FileStreamException est la superclasse de FileDoesNotExistException, FileExistsException et CannotDeleteFileException. FileStreamException étant une classe à part entière elle peut ajouter des informations qui sont importantes pour caractériser l'erreur qu'elle représente. Ainsi FileStreamException définit la variable d'instance fileName qui permet de préciser le nom du fichier créant problème. La racine de la hiérarchie des exceptions est la classe Exception.



Les deux messages clefs sont on:do: et signal. Comme on l'a vu précédemment, un message de la forme on: anException do: aBlock évalue aBlock si le bloc receveur du on:do: lève une exception. Les messages signal/signal: permettent de lever une exception.

Squeak contient un ensemble de tests d'excellente facture qui a été spécifié par les développeurs de Squeak pour s'assurer de l'adhérence de Squeak au regard de la norme. Nous utilisons le cas échéant certains de ces tests car ils illustrent de manière compacte le comportement que nous voulons montrer.

Signaler une Erreur

Pour signaler une exception, il suffit de créer une instance de l'exception et de lui envoyer le message signal ou signal: avec un texte descriptif. La classe Exception définit la méthode signal qui permet de créer et signaler une exception. Voici deux façons équivalente de lever l'exception ZeroDivide. La séconde crée une instance et lui envoie le message signal.

```
ZeroDivide new signal    ou    ZeroDivide signal
```

On peut se demander pourquoi on crée une instance juste pour signaler une erreur et pourquoi ne pas plutôt utiliser la classe pour faire cela. Créer une instance est important car cette instance représente la situation exceptionnelle que nous voulons décrire, nous allons pouvoir associer un état spécifique à cette instance qui caractérisera le problème. Ainsi il est possible d'avoir plusieurs exceptions d'une même classe représentant des erreurs différentes. Lorsque une erreur est signalée, le mécanisme d'exception va rechercher dans la pile d'exécution si il y a un bloc de récupération d'erreur (handler) associé à ce type d'erreur ou plus général et auquel cas l'exécuter en lui passant l'instance d'exception créée. En effet, le bloc handler est un bloc ayant un argument qui représente l'exception ayant signalée l'erreur. Nous verrons plus loin comment cette instance peut être utilisée.

Ainsi il est possible de préciser lors du signalement d'une erreur des informations spécifiques comme l'illustre

le code suivant pris de la classe StandardFileStream. Ici lorsque l'exception est signalée, le code précise le nom du fichier qui n'existe pas en utilisant le message #fileName:.

```
StandardFileStream>>oldFileNamed: fileName
    "Open an existing file with the given name for reading and writing. If the name has no directory part, then the file will be created in the default directory. If the file already exists, its prior contents may be modified or replaced, but the file will not be truncated on close."
    | fullName |
    fullName := self fullName: fileName.
    ^(self isAFileNamed: fullName)
        ifTrue: [self new open: fullName forWrite: true]
        ifFalse: ["File does not exist..."
            (FileDoesNotExistException new fileName: fullName) signal]
```

Ensuite le code suivant pourra tirer avantage du nom du fichier qui a conduit à lever l'erreur. L'argument ex du bloc traitant l'exception [:ex| ...] est l'instance d'exception FileDoesNotExistException ou une de ses sous-classes qui a été signalée. Ici le traitement consiste simplement à notifier que le fichier n'existe pas en utilisant les informations données par l'instance d'exception.

```
    | result |
    result := [(StandardFileStream oldFileNamed: 'error42.log') contentsOfEntireFile]
        on: FileDoesNotExistException
        do: [:ex | ex fileName , 'not available'].
    Transcript show: result; cr
```

Par défaut, une exception possède une description utilisée par les outils de développement afin de reporter des erreurs plus compréhensibles. Ainsi toute instance d'exception sait répondre au message #description. De plus, le message de description peut être précisé soit en utilisant le message #messageText: ou en signalant l'exception à l'aide du message #signal: et en lui fournissant la description.

Le mécanisme du fameux message #doesNotUnderstand: qui fait apparaître un débogueur est basé sur la définition d'une exception MessageNotUnderstood sous-classe directe d'Error. Ce signal intervient pour tout message non compris par un objet. Le code de la méthode #doesNotUnderstand: définie sur la classe Object illustre le fait que l'on peut associer des informations spécifiques à une instance d'une classe d'exception. Ici, le receveur du message, ainsi le message envoyé sont stockés dans l'instance de MessageNotUnderstood auquel le message signal est envoyé. Le mécanisme qui conduit à l'ouverture d'un débogueur Smalltalk sera expliqué lors de la présentation du fonctionnement du message halt car il s'agit du même mécanisme.

```
Object>>doesNotUnderstand: aMessage
    "Handle the fact that there was an attempt to send the given message to the receiver but the receiver does not understand this message (typically sent from the machine when a message is sent to the receiver and no method is defined for that selector)."
    MessageNotUnderstood new
        message: aMessage;
        receiver: self;
        signal.
    ^ aMessage sentTo: self.
```

Identification et exécution des handlers

Maintenant, nous décrivons comment les bloc de récupérations associés à une séquence et une erreur sont sélectionnés pour réaliser la récupération lorsqu'une exception est levée. Pour cela nous devons expliquer comment l'exécution d'un programme est représentée par le système.

A tout moment, la pile d'exécution d'un programme est représentée par ce que l'on appelle une liste de contextes d'activation. Un contexte de l'activation d'une méthode représente les données nécessaires à l'identification de la méthode et son exécution (le receveur, les arguments d'appel, variables locales), elle contient également une référence au contexte de la méthode appelante. En Squeak les classes MethodContext et BlockContext représentent ces informations. Les contextes forment ainsi une chaine d'activations représentant la pile d'exécution. Par exemple, quand l'exception FileDoesNotExistsException est lancée, la pile d'exécution contient les contextes suivants en tentant d'ouvrir un bookmorph sur un fichier inexistant.

Il est à noter que les contextes d'exécution sont aussi des objects en Smalltalk. Suivant les implantations, ces objets peuvent être substitués par la pile d'exécution de la machine virtuelle afin d'eviter de manipuler des objets en permanence. En Squeak la machine virtuelle ne fait pas cette substitution mais manipule des objets qui représentent la pile.

L'exécution du message aBlock on: ExceptionClass do: anHandler attache au contexte d'exécution créé à la fois le type d'exception représenté par ExceptionClass et le bloc de récupération associé (anHandler). Ce sont ces informations qui sont utilisées pour identifier le bloc adéquat quand une exception est lancée. Par exemple, à tout moment la pile d'exécution contient un handler.

Si aucun handler n'est présent sur la pile, la méthode #defaultAction est invoquée soit par ContextPart>>handleSignal: soit par UndefinedObject>>handleSignal:. Au tout début de la pile d'execution, la seconde méthode. La méthode est définie de la façon suivante:

UndefinedObject>>handleSignal: exception

"When no more handler (on:do:) context left in sender chain this gets called. Return from signal with default action."

^ exception resumeUnchecked: exception defaultAction

Et #handleSignal: est invoquée par Exception>>signal.

Voici les étapes conceptuelles par lesquelles le système passe pour identifier et exécuter le bloc de récupération à exécuter suite à une exception E.

Etape 1. Trouver tous les contextes de la pile d'invocation qui ont établi des handlers. Nous obtenons une liste ordonnées de contextes.

Etape 2. Elimine de cette liste tous les contextes dont le type d'exception n'est pas compatible avec E. C'est ici que l'on voit qu'une exception plus générale peut être utilisée pour identifier un bloc de récupération. Cette liste contient alors tous les bloc de récupération qui peuvent récupérer l'exception E. Ils sont aussi triés par ordre de distance plus proche par rapport au contexte ayant levé l'exception.

Etape 3. Cette étape consiste à exécuter alors le premier handler et de garder la liste des contextes restant

afin de pouvoir leur passer le contrôle si bloc de récupération le décide.

Une situation importante est aussi de traiter le cas où la liste des blocs de récupération est vide. Dans ce cas, l'exception levée (E) est elle-même sollicitée: on envoie le message #defaultAction à l'exception. Par défaut sur la classe Error, cette méthode conduit à l'ouverture d'un débogueur de la même façon que la méthode halt dont nous expliquons le mécanisme plus loin.

Notez que dans la pratique, la machine virtuelle ne navigue pas toute la pile d'exécution pour récupérer la liste des contextes mais utilise la chaine d'activations et la remonte si nécessaire pour trouver le premier contexte et son handler, et l'exécuter.

L'exemple suivant montre que si la même exception est levée lors du bloc de récupération elle n'est pas capturée et mène à une exception non gérée (unhandledException).

```
signalFromHandlerActionTest
        [self doSomething.
        MyTestError signal.
        self doSomethingElse]
                on: MyTestError
                do: [self doYetAnotherThing.
                    MyTestError signal]
```

Supposons que les messages affiche une chaine dans le Transcript, nous obtenons alors 'doSomething / do YetAnotherThing' suivi d'un débogueur. Ce comportement est normal puisque la seconde exception est levée hors du bloc protégé par le message #on:do:. Pour qu'elle est une chance d'être récupérée il faudrait qu'un autre message #on:do: définisse un block erreur associé pour l'ensemble de l'expression comme suit :

```
signalFromHandlerActionTestWithSecondOnDo
        [[self doSomething.
        MyTestError signal.
        self doSomethingElse]
                on: MyTestError
                do: [self doYetAnotherThing.
                    MyTestError signal]]
                        on: MyTestError
                        do: [ self doFinal]
```

Cette fois-ci nous obtenons: 'doSomething / doYetAnotherThing / doFinal' sans débogueur. Nous obtenons la même situation si nous contrôlons plus finement le bloc levant la seconde exception.

```
signalFromHandlerActionTestWithSecondOnDo
        [self doSomething.
        MyTestError signal.
        self doSomethingElse]
                on: MyTestError
                do: [ [self doYetAnotherThing.
                        MyTestError signal]
                                on: MyTestError
                                do: [ self doFinal]]
```

Possibilités pour la récupération d'erreur

Regardons maintenant les possibilités qui s'offrent lors de la récupération d'erreur. En Smalltalk nous pouvons, abandonner l'exécution, rendre une valeur, ressayer, signaler une autre exception, passer la gestion de l'exception au bloc de récupération suivant ou relancer l'application comme si de rien était. Nous

abordons les premières possibilités maintenant et les plus avancées comme la reprise du flôt de programme dans un second article.

Abandonner l'exécution.

Une des premières possibilités est simplement d'abandonner l'exécution du bloc protégé. Imaginons que nous ayons une méthode pour afficher des menus et que la première action lève une erreur. Il est possible de simplement abandonner l'exécution du bloc protégé en par exemple présentant un message d'erreur.

```
menu
	[ self firstActionSignalingAnError.
	self secondActionReturning42 ]
		on: Error
		do: [:ex | UIManager default inform: 'An error has occurred: ', ex messageText ].
```

Rendre une valeur (le message return: uneValeur)

Évaluer un bloc retourne la valeur de la dernière instruction contenue dans ce block. Et ce, que ce bloc soit protégé ou pas. Cependant, il y a des situations où le résultat doit être retourné par le bloc de récupération. Le message #return: uneValeur envoyé à une exception a pour effet de retourner uneValeur comme valeur du bloc protégé. Une variation est le message return qui rend la valeur nil.

Voici un exemple qui rend quand le fichier demandé n'existe pas rend la valeur nil

```
stream := [FileStream readOnlyFileNamed: url pathForFile]
		on: FileDoesNotExistException do:[:ex| ex return: nil].
```

Notez que la spécification n'est pas claire par rapport à la différence entre do: [:ex | 100] et do: [:ex | ex return: 100] pour retourner une valeur. Nous suggérons d'utiliser le message #return:, bien que ces deux expressions soient équivalentes en Squeak.

```
% a garder peut etre pour le bouquin
%timeStamp
%	"Answer a TimeStamp that corresponds to my (text) stamp"
%	| tokens date time |
%	tokens := self stamp findTokens: Character separators.
%	^ tokens size > 2
%		ifTrue: [[date := Date fromString: (tokens at: tokens size - 1).
%			time := Time fromString: tokens last.
%			TimeStamp date: date time: time]
%				on: Error
%				do: [:ex | ex return: (TimeStamp fromSeconds: 0)]]
%		ifFalse: [TimeStamp fromSeconds: 0]
```

Notez que le message return/return: est un message avec une sémantique un peu spéciale car bien qu'il permette de rendre une valeur, à la suite de son exécution, le flot de contrôle ne revient pas au point de programme suivant ce message.

Réessayer avec retry and retryUsing:.

Parfois on peut vouloir simplement refaire la même action avec des paramètres différents ou une action

différente. Cela est possible en utilisant les messages #retry et #retryUsing:. Notez qu'il faut faire attention que lors du nouvel essai, le contexte qui a levé l'erreur originale ne soit pas reproduit car cela pourrait aboutir à une boucle infinie.

L'exemple suivant prit du framework de test des exceptions, montre un exemple d'utilisation de #retry. L'exécution affiche les résultats suivants qui montrent le flot suivi: 'doSomethingString / doYetAnotherThingString / doSomethingString / doSomethingElseString'. Il montre clairement que le bloc protégé est exécuté de nouveau depuis le début mais dans un environnement modifié.

```
ExceptionTester>>simpleRetryTest
	| theMeaningOfLife |
	theMeaningOfLife := nil.
	[self doSomething.
	theMeaningOfLife == nil
		ifTrue: [MyTestError signal]
		ifFalse: [self doSomethingElse]]
			on: MyTestError
			do: [:ex | theMeaningOfLife := 42.
				self doYetAnotherThing.
				ex retry]
```

Le message retryUsing: aNewBloc qu'en t'a lui permet de spécifier une séquence d'actions nouvelles qui seront exécutées à la place du bloc protégé en cas de nouvel essai. L'exemple suivant affiche : 'doSomethingString / doYetAnotherThingString'

```
ExceptionTester>>simpleRetryUsingTest
	[self doSomething.
	MyTestError signal.
	self doSomethingElse]
		on: MyTestError
		do:  [:ex | ex retryUsing: [self doYetAnotherThing]]
```

En reprenant un des exemples utilisés au début, nous pouvons ainsi proposer d'interroger l'utilisateur lorsqu'un fichier n'est pas trouvé.

```
	| result |
	result := [(StandardFileStream oldFileNamed: 'error42.log') contentsOfEntireFile]
				on: FileDoesNotExistException
				do: [:ex |
					| newName |
					newName := UIManager default request:  'Problem reading file. Another name?'.
			ex retryUsing: [(StandardFileStream oldFileNamed: newName) contents]
				].
```

Notez que l'utilisation de #retryUsing: dans cet exemple n'est pas optimal car il duplique la logique du bloc protégé. Nous vous laissons comme exercice la réécriture de l'exemple en paramètrisant le code du bloc protégé et en utilisant #retry. Notez encore une fois la particularité des messages #retry and #retryUsing:, le flot de contrôle ne revient pas après l'exécution de tels messages contrairement à un message normal.

Dans notre prochain article, nous vous montrerons les autres possibilités comme le fait de pouvoir relancer le programme comme si l'erreur n'avait pas existée ou de passer la main aux autres blocs de récupération.

A propos de halt.

En Smalltalk il est toujours intéressant de regarder comment le système lui-même utilise les exceptions. Le message #halt permet de stopper l'exécution d'un programme. Il agit comme un breakpoint. Par defaut, le message #halt, mène à l'ouverture d'un débogueur. Ce comportement est basé sur la levée d'exceptions. Regardons la définition de la méthode #halt définie sur la classe Object.

```
Object>>halt
        "This is the typical message to use for inserting breakpoints during
        debugging. It behaves like halt:, but does not call on halt: in order to
        avoid putting this message on the stack. Halt is especially useful when
        the breakpoint message is an arbitrary one."
        Halt signal
```

L'exception Halt est une sous-classe directe de la classe Exception. L'exception Halt se definit comme pouvant être redémarrée (resumable) en définissant la méthode isResumable pour rendre true. Nous verrons la puissance des exceptions resumables dans l'article suivant. L'exception Halt redéfinit la méthode #defaultAction qui spécifie l'action à exécuter lorsque personne ne traite l'erreur (c-a-d ne définit de bloc de récupération pour une exception de ce type).

```
Halt>>>defaultAction
        "No one has handled this error, but now give them a chance to decide how to debug it. If none
handle this either then open debugger (see UnhandedError-defaultAction)"
        UnhandledError signalForException: self
```

Ici la méthode #defaultAction léve une nouvelle exception UnhandledError en spécifiant l'erreur qui n'a pas été récupérée. Vous pouvez regarder la définition de la méthode #signalForException: anError. L'exception UnhandledError définit comme action par défaut l'ouverture d'un débogueur comme suit.

```
UnhandledError>>defaultAction
        "The current computation is terminated. The cause of the error should be logged or reported to the
user. If the program is operating in an interactive debugging environment the computation should be
suspended and the debugger activated."
        ^ToolSet debugError: exception.
```

ce qui après quelques messages conduit à l'ouverture d'un débogueur comme suit.

```
StandardToolSet>>debug: aProcess context: aContext label: aString contents: contents fullView: aBool
    ^Debugger openOn: aProcess context: aContext label: aString contents: contents fullView: aBool
```

Lorsqu'un message est envoyé à un objet qui ne sait y répondre, la machine virtuelle ne trouvant pas de méthodes adéquates à exécuter dans la classe du receveur ou superclasses envoie le message #doesNotUnderstand: au receveur. Si ce message n'est pas redéfinit, il utilise le même mécanisme d'exception que le Halt qui mène à l'ouverture d'un débogueur. Cette utilisation d'exception a comme avantage de pouvoir modifier ce comportement dans un environnement déployé pour par exemple copier la pile et l'envoyer par mail en cas de problème.

Conclusion

Cet article offre une introduction aux exceptions en Smalltalk et tout particulièrement en Squeak. L'objectif de cet article est double : illustrer l'intérêt d'utiliser un mécanisme d'exception pour se protéger des défaillances et présenter les particularités du langage Smalltalk. En effet, très peu de langages de programmation offrent des outils aussi avancés pour manipuler les exceptions. Par exemple, les langages dits « mainstreams » ne peuvent pas placer des handlers dans des expressions, mais uniquement comme statement. Par exemple,

l'équivalent de x := [Error signal] on: Error do: [:ex| 5] n'a pas d'équivalent. De plus, retry n'a pas d'équivalent non plus.

Dans un prochain article nous présenterons les constructions plus excitantes et des utilisations des exceptions à des fin de notifications.

Liens

- [BLUEB] Smalltalk-80: The Language and its Implementation : http://stephane.ducasse.free.fr/FreeBooks/BlueBook/
- Le site officiel : http://www.squeak.org/
- Pharo project: http://www.pharo-project.org/
- Le Wiki de la communauté française : http://community.ofset.org/wiki/Squeak
- Le groupe des utilisateurs européens de Smalltalk (European Smalltalk User Group). L'adhésion est gratuite : http://www.esug.org/
- Des livres gratuits en ligne sur Smalltalk et Squeak : http://stephane.ducasse.free.fr/FreeBooks.html
- Squeak par l'Exemple, http://www.squeakbyexample.org/fr/