Des exceptions exceptionnelles en Smalltalk

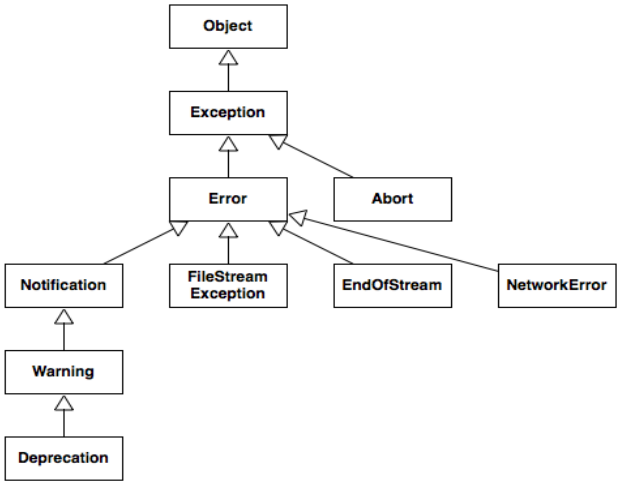
S.Ducasse – stephane.ducasse@free.fr
A. Bergel – alexandre@bergel.eu

Dans l'article précédent nous avons montré comment Smalltalk permet de lever, capturer des exceptions. Nous avons montré comment une exception peut être relancée. Dans cet article nous aborderons les aspects les plus puissants de la gestion des exceptions en Smalltalk et tout particulièrement la reprise du flot d'exécution au point de signalement d'une exception. Nous montrerons comment l'exception de Deprecation fonctionne et ce qu'est une reprise d'exception.

Error et Notification

Tout d'abord, regardons les différentes sortes d'exceptions proposées en Smalltalk. Ici il n'y a pas de différence notable avec des langages tels que C# ou Java. Différentes sortes d'exceptions peuvent être levées: on distingue deux principales catégories, les erreurs indiquant que le programme ne peut plus s'exécuter normalement et les notifications qui indiquent qu'un événement se passe mais que si aucune action n'est effectuée le programme continue sa marche normalement. Les notifications sont souvent spécialisées en alertes (classe Warning) qui sont utilisées pour notifier des comportements anormaux mais non fatales.

Typiquement un message non compris (message not understood) lève l'exception MessageNotUnderstood qui est une sous-classe de la classe Error. Les notifications sont utilisées pour implanter des retours d'informations dans les interfaces graphiques. En Squeak, ProgressNotification sert à faire avancer les barres de progression d'une tâche. ProgressNotification est une sous-classe de Notification.



En Smalltalk, une exception possède une propriété supplémentaire : non seulement on peut lui définir des états et méthodes supplémentaires mais une exception peut déclarer que le flot du programme qui lève l'exception peut reprendre (resume en anglais) suite à la levée de cette exception. Le flot normal de l'application continue alors. Bien sûr, il n'y a pas de magie et le bloc de récupération doit avoir changer l'état du système pour que cette reprise fasse du sens.

La propriété d'être reprise est orthogonale à la notion d'erreur. Une erreur n'est pas forcément non-resumable. Par exemple, l'exception MessageNotUnderstood qui est une exception sous-classe d'Error mais elle est resumable. La classe MessageNotUnderstood définit la méthode isResumable comme suit pour déclarer que la reprise du flot de contrôle est possible.

MessageNotUnderstood>>isResumable

```
^true
```

Maintenant regardons comment un bloc de récupération peut demander la reprise du cours normal de l'application. Encore une fois comme tout est objet en Smalltalk, il s'agit d'un message de la classe Exception: le message #resume:.

Continuer l'exécution : le message resume:

Nous avions dit qu'un des traitements possibles consiste à continuer l'exécution du bloc protégé juste aprés le point où l'exception est levée. Ainsi il est possible pour le bloc de récupération d'effectuer un traitement puis de repartir dans le flot d'exécution du bloc protégé. Cette fonctionnalité mise en oeuvre à l'aide du message resume:, offre de grandes possibilités de gestion d'erreurs et une grande souplesse. Regardons tout d'abord un test disponible dans l'ExceptionTester.

```
| it |
[self doSomething.
it := MyResumableTestError signal.
it = 3 ifTrue: [self doSomethingElse].
it := MyResumableTestError signal.
it = 3 ifTrue: [self doSomethingElse] ]
        on: MyResumableTestError
        do: [ :ex |
                self doYetAnotherThing.
                ex resume: 3 ]
```

Ce test affiche les résultats suivants: 'doSomething / doYetAnotherThing / doSomethingElse / doYetAnotherThing / doSomethingElse'. Il illustre que le message #resume: anObject retourne anObject comme valeur du message qui a signalé l'erreur. Ici le message MyResumableTestError signal conduit d'une part à l'exécution du bloc de récupération qui exécute doYetAnotherThing et le message #resume: 3 rend la valeur 3 comme résultat de l'execution de l'expression MyResumableTestError signal. Le point clef à comprendre est que #resume: permet de continuer l'exécution du programme au point exact où l'exception a été levée et d'y associer une valeur.

Le message #resume est aussi disponible mais ne permet pas de rendre une valeur. Le test suivant affiche 'doSomething / doSomethingElse / doYetAnotherThing' et illustre bien la reprise du flot dans le bloc protégé.

```
[self doSomething.
MyResumableTestError signal.
self doSomethingElse.
MyResumableTestError signal.
self doYetAnotherThing]
        on: MyResumableTestError
        do: [ :ex | ex resume].
```

L'exemple suivant montre comment Installer (la classe qui installe les packages) installe un package sachant que celui-ci peut lever des warnings.

```
Installer>>installQuietly: packageNameCollectionOrDetectBlock
        self package: packageNameCollectionOrDetectBlock.
        [ self install ] on: Warning do: [ :ex | ex resume: true ].
```

Une seconde utilisation typique de #resume est lorsqu'une interaction avec l'utilisateur est nécessaire, comme dans l'exemple suivant:

```
MyApplication>>readOptionsFrom: aStream
  | option |
```

```
    [aStream atEnd] whileFalse:
      [option := self parseOptionString. "nil if invalid"
      option isNil
        ifTrue: [InvalidOption signal]
        ifFalse: [self addOption: option]].
    aStream close.

MyApplication>>readConfiguration
    | stream |
    stream := 'options' asFilename readStream.
    [self readOptionsFrom: stream]
      on: InvalidOption
      do: [:ex |
        (UIManager default confirm: 'Invalid option line. Continue loading?')
          ifTrue: [ex resume]
          ifFalse: [ex return]].
    stream close.
```

Si une option invalide est fournie dans #readOptionsFrom:, une exception InvalidOption est signalée. Si le bloc de récupération dans #readConfiguration décide de faire un #resume, le message #signal dans #readOptionsFrom: retourne et l'exécution se poursuit.

Naturellement, il n'est pas toujours judicieux d'utiliser #resume dans toutes les situations. Par exemple, il n'y a pas de manière raisonnable à poursuivre l'exécution après une exception FileNotFound. C'est pourquoi chaque exception spécifie si elle peut être reprise ou pas (méthode #isResumable). Dans Error, #isResumable renvoie false, donc par défault les erreurs ne peut être reprises. Une variante de #resume est #resume:, qui permet au message signal de retourner une valeur.

Deprecation: Warning Par l'exemple

Nous montrons maintenant les étapes de définition d'une exception avec l'implantation du pattern Deprecation. Le pattern de deprecation permet d'indiquer qu'une fonctionnalité ne doit plus être utilisée et sera enlevée dans une version suivante. En Squeak, on exprime ce pattern de la manière suivante: on utilise le message #deprecatedExplanation: comme suit.

```
foo
        self deprecated: 'The method #foo was not good. Use Bar>>newFoo instead.'
        ^ 'foo'
```

Lorsque la méthode foo est exécutée, si la préférence qui demande de notifier les méthodes dépréciées est activée, alors une fenêtre de notification est affichée et le développeur peut éventuellement demander à continuer l'exécution et ainsi passer outre la dépréciation.

Nous montrons comment le pattern Deprecation a été ajouté dans Squeak. Pour définir sa propre exception, il suffit de créer une sous-classe d'une des classes d'exception. Ici, Deprecation est une sous-classe de Warning. Nous devons définir les méthodes : #isResumable, #description et #defaultAction. De plus comme les méthodes héritées nous satisfont et nous n'avons plus rien à faire. Ici la méthode #defaultAction de la classe Warning ouvre un débogueur quand l'exception n'est pas capturée et la méthode #isResumable de la superclass Notification rend vrai pour indiquer que le flot peut être repris: ce qui est clairement le cas avec un message de dépréciation.

```
Warning>>defaultAction
        "The user should be notified of the occurrence of an exceptional occurrence and given an option of
```

```
        continuing or aborting the computation. The description of the occurrence should include any text
        specified as the argument of the #signal: message."
        ToolSet
                debugContext: thisContext
                label: 'Warning'
                contents: self messageText, 'Select Proceed to continue, or close this window to cancel the
operation.' withCRs.
        self resume.


Notification>>isResumable
        "Answer true. Notification exceptions by default are specified to be resumable."
        ^true
```

Pass, resignalAs:, outer

Pour illustrer les dernières fonctionnalités, nous vous proposons de définir un nouveau message et d'en changer étape par étape la sémantique pour comprendre le message #pass et #resignalAs:.

Le message #perform: en Smalltalk permet d'envoyer un message dont le nom de la méthode à invoquer est passé en paramètre. Il existe plusieurs variantes suivant le nombre d'arguments du message. Ainsi 1 perform: #+ withArgument: 2 est équivalent au message 1 + 2. De la même manière Beeper beep est équivalent à Beeper perform: #beep.

La méthode #perform: est très utile pour créer des interfaces ou du comportement qui n'est pas défini de manière statique mais qui peut être créé à la volée.

Imaginons que nous voulons définir une méthode qui permet d'envoyer plusieurs messages unaires à un même objet. Une définition immédiate est la suivante. Nous parcourons la liste des selecteurs et les exécutons un par un à l'aide du message #perform:.

```
Object>>performAll: selectorCollection
        selectorCollection do: [:each | self perform: each]
```

Cependant, que faisons nous si l'un des sélecteurs de la collection n'est pas compris par le receveur ? Nous pouvons par exemple capturer ce type d'exception. Nous pouvons redéfinir la méthode comme suit:

```
Object>>performAll: selectorCollection
        selectorCollection do:  [:each |
                                [self perform: each]
                                        on: MessageNotUnderstood
                                        do: [:ex | ex return]]
```

Le problème de cette définition est qu'elle capture bien les erreurs liées au fait qu'un des sélecteurs de la liste peut être incompris par le receveur mais aussi les erreurs liées à l'exécution d'une méthode. Ce n'est guère satisfaisant, car cela peut cacher des bogues difficiles à identifier. Nous voulons donc distinguer les erreurs qui sont produites par le sélecteur dont nous avons demandé l'exécution. Nous allons donc faire appel à la méthode #pass qui permet de relancer la recherche de bloc de récupération.

Nous modifions notre définition pour que l'exception soit retourne (dans le cas ou il s'agit d'un sélecteur inconnu), soit quand l'erreur est due à une méthode invoquée de laisser l'erreur se produire. Nous demandons à l'exception de retourner ou sinon de passer la récupération de l'erreur au prochain handler. Pour cela nous utilisons le message #pass. Cette méthode définie sur la classe Exception ne traite pas l'exception et passe la récupération au prochain bloc de récupération (handler) – ce qui peut éventuellement amener à une exception non-traitée. On voit ici clairement l'utilisation de la chaîne de bloc de récupération mentionnée dans le premier article. Le méthode Exception>>pass est elle aussi une méthode spéciale tout comme #retry:, #return: et #resume:. l'exécution de la méthode #pass ne revient pas au point du flot de contrôle qui les a invoqués.

Notez ici que notre implantation vérifie aussi que seuls les messages envoyés au receveur initial sont traités.

```
Object>>performAll: selectorCollection
    selectorCollection do: [:each |
            [self perform: each]
                on: MessageNotUnderstood
                do: [:ex | (ex receiver == self and: [ex message selector == each])
                        ifTrue: [ex return]
                        ifFalse: [ex pass]]]
```

Supposons maintenant que nous ne voulons plus ignorer les messages non-compris par le receveur mais que nous voulions lever une erreur spécifique que nous définissons (InvalidAction). Pour cela nous voulons pouvoir relever une exception *à la place* de l'exception originale.

Signaler la nouvelle exception (comme présentée dans le code suivant) depuis le bloc de récupération ne fonctionne pas totalement car la nouvelle exception est levée à un endroit différent dans la pile d'exécution que le message original. Cela peut mener à une situation différente lors du calcul des blocs de récupération applicable.

```
[:ex | (ex receiver == self and: [ex message selector == each])
            ifTrue: [InvalidAction signal]
            ifFalse: [ex pass]]
```

La solution consiste à utiliser la méthode #resignalAs: comme suit:

```
[:ex |  (ex receiver == self and: [ex message selector == each])
            ifTrue: [ex resignalAs: InvalidAction]
            ifFalse: [ex pass]]
```

La méthode #resignalAs: permet de signaler une exception alternative à la place de l'exception receveur de ce message.

#outer est très similaire à #pass. Envoyer le message #outer à une exception fait basculer le flot de contrôle au prochain handler présent dans la pile. La différence se tient dans l'effet d'un #resume ou #resume: qui peut s'en suivre. Résumer une exception levée par #outer poursuit l'exécution là où #outer a été invoqué. Par exemple, considérons cet exemple tiré de la classe ExceptionTester:

```
[[self doSomething.
MyTestNotification signal.
self doSomethingExceptional]
        on: MyTestNotification
        do: [:ex | ex outer. self doSomethingElse]]
                on: MyTestNotification
                do: [:ex | self doYetAnotherThing. ex resume]
```

Les méthodes invoquées sont # doSomething, #doYetAnotherThing, et #doSomethingElse. L'utilisation de #outer fait passer le flot de contrôle dans le prochain handler de la pile. Le resume renvoie le flot de contrôle au niveau du #outer. Si #pass avait été utilisé à la place de #outer, le resume aurait renvoyé là où le signal a été levé. Dans ce cas, la 3ème méthode qui aurait été invoquée est #doSomethingExceptional.

Ensemble d'exceptions

Jusqu'à présent, nous avons vu qu'un bloc de récupération d'exception est associé à un type d'exception bien défini. Un bloc va être activé que si une exception du même type auquel il est associé est signalée – soit l'exception est une instance de la classe spécifiée, soit elle est une sous-classe. Cependant, il y a des situations où on aimerait avoir un bloc de récupération pour un ensemble d'exceptions non-liée entre elle par une relation de type. Un ensemble d'exception peut être utilisé comme 1er argument dans le #on:do:. Cela permet d'activer le handler pour plus d'une exception. Par exemple:

```
[...]
on: MessageNotUnderstood, FileNotFound, ZeroDivide
do: […]
```

Le bloc de récupération […] sera activé pour toute exception signalée et comme étant une sous-classe ou non des classes MessageNotUnderstood, FileNotFound, et ZeroDivide.

Finalement comment tout cela est implanté ?

Vous n'avez pas à lire la suite et vous n'avez pas à comprendre ceci pour utiliser les exceptions de manière pratique. Par contre, si comme nous vous vous posiez des questions, nous allons ouvrir le couvercle avec vous. Comme vous le verrez ce n'est pas compliqué et cela ne vaut pas la peine de ne pas savoir. Ce que nous allons faire est par contre un exemple superbe de l'accès à l'information en Smalltalk.

Regardons d'abord la définition de la méthode on:do:. On voit deux points: il s'agit d'une primitive (199) qui est un point d'appel de code C de la machine virtuelle et la définition à proprement parlée de la méthode: le résultat de l'exécution du receveur (le bloc protégé) est rendu, et le booléen handlerActive est mis a vrai.

BlockContext>>on: exception do: handlerAction

```
"Evaluate the receiver in the scope of an exception handler."
| handlerActive |
<primitive: 199>
handlerActive := true.
^self value
```

On pourrait croire que la primitive 199 est complexe et écrite en C. Mais ce n'est pas le cas. La primitive 199 est une primitive qui produit systématiquement une erreur. En Smalltalk, cela veut dire que le code qui suit la primitive est lors exécuté. Ici il est donc systématiquement exécuté. ici la méthode #on:do: est exécutée comme si elle n'était pas une primitive. L'exécution de la primitive sert juste à marquer le contexte d'exécution (instance de BlockContext): Ce contexte d'exécution est marqué avec le numéro de la primitive (ici 199). Le type d'exception (Error, MessageNotUnderstood...) ainsi que le block de récupération sont stockés dans ce contexte d'activation (à l'indice 1 et 2 respectivement). Pour vous en convaincre, exécutez le code suivant. Vous obtenez un tableau de deux éléments avec l'exception et le bloc de récupération.

```
| exception handler |
[exception := thisContext sender at: 1.
 handler := thisContext sender at: 2.
          1 / 0]
         on: Error
         do: [:ex| ].
{ exception. handler }
```

La machine virtuelle trouve donc les contextes d'activation qui représentent un appel à on:do: simplement en examinant les numéros des contextes d'activation. En gros la machine virtuelle parcourt la pile à la recherche d'une activation de méthode compilés de numéro 199. Le code de la machine virtuelle est une optimisation du code Smalltalk suivant.

On voit avec la méthode #findNextHandlerContextStarting que la recherche du bloc de récupération cherche depuis le contexte activé le premier contexte représentant l'exécution d'un message #on:do: et retourne nil si elle arrive en haut de la pile.

MethodContext>>findNextHandlerContextStarting

```
"Return the next handler marked context, returning nil if there
is none. Search starts with self and proceeds up to nil."
| ctx |
```

```
<primitive: 197>
ctx := self.
        [ctx isHandlerContext ifTrue:[^ctx].
        (ctx := ctx sender) == nil ] whileFalse.
^nil
```

MethodContext>>isHandlerContext

```
"is this context for method that is marked?"
^method primitive = 199
```

Les contextes de méthodes (également connu sous le nom d'activation frame ou frame tout simplement) sont des objets manipulables comme n'importe quel objets. Ils forment une liste chaînée, qui peut être tronquée à n'importe quel moment. Il s'agit du principe de base des exceptions. En théorie, il n'est pas nécessaire à la machine virtuelle de fournir des primitives dédiées aux exceptions. Il s'agit là d'un bien bel exemple de la puissance Smalltalk.

Conclusion

Ce second article sur les exceptions en Smalltalk offre un large panel d'outils complètement inconnus des langages de programmation plus répandus tel que Java et C++. Bien que seulement une partie de la norme ANSI soit présentée, nous avons passé en revue les mécanismes les plus utilisés de manipulation du flot de contrôle en utilisant les exceptions.

Au dela de la connaissance « technique », cet article montre la richesse et l'incroyable expressivité des langages de programmation dit dynamiquement typés dont Smalltalk en fait partie.

Liens

- Le site officiel : http://www.squeak.org/
- Pharo project: http://www.pharo-project.org/
- Des livres gratuits en ligne sur Smalltalk et Squeak : http://stephane.ducasse.free.fr/FreeBooks.html
- Squeak par l'Exemple, un livre open-source sur Squeak http://www.squeakbyexample.org/fr/
- 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/