Reflexió en Smalltalk

- Pròleg
  - Les Mestaclasses en 7 parts
  - Classes Indexades i Variables d'Instància
  - Variables de class-instància
  - Variables de class
- Reflexió
  - Introspecció
    - Inspecció d'objectes
    - Consultar el codi
    - Accedit els contexts d'execució
  - Intercessió
    - Sobreescriure #doesNotUnderstand
    - Classes Anònimes
    - Method Wrappers
- Continuacions

Pròleg

Les *Metaclasses* en 7 parts

1. Tot objecte és instància d'una classe

2. Tota classe hereta eventualment d'Object

Tot és un objecte. La classe de cada objecte hereta d'*Object*

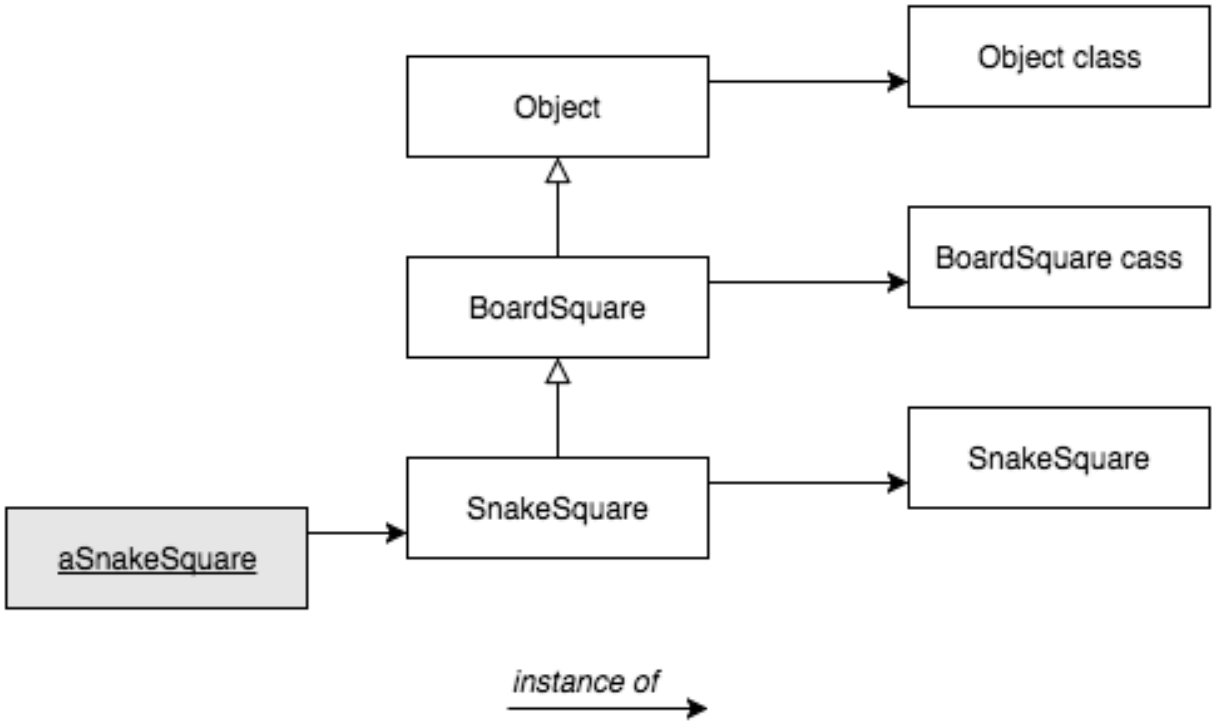
Quan un objecte rep un missatge, el mètode es busca al diccionari de mètodes de la seva classe, i, si cal, a les seves superclasses, fins arribar a *Object*.

Object representa el comportament comú a tots els objectes (com la gestió d'errors, per exemple). Totes les classes haurien d'heretar d'*Object*.

3. Tota classe és instància d'una metaclasse

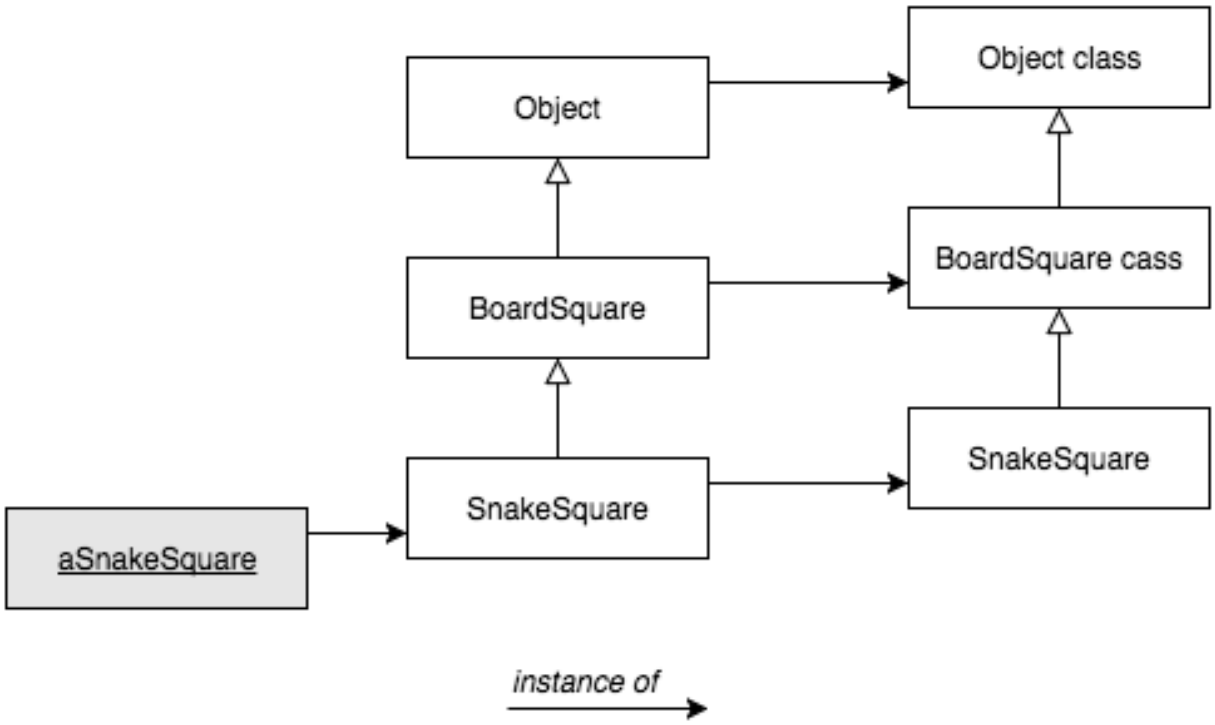
Com a Smalltalk tot és un objecte, les **classes també son objectes**. Cada classe X és l'única instància de la seva *metaclasse* anomenada X *class*.

Les *metaclasses* es crean quan es crea un classes de forma implícita. Les *metaclasses* no es comparteixen, cada classe és **instància única** de la seva *metaclasse*.

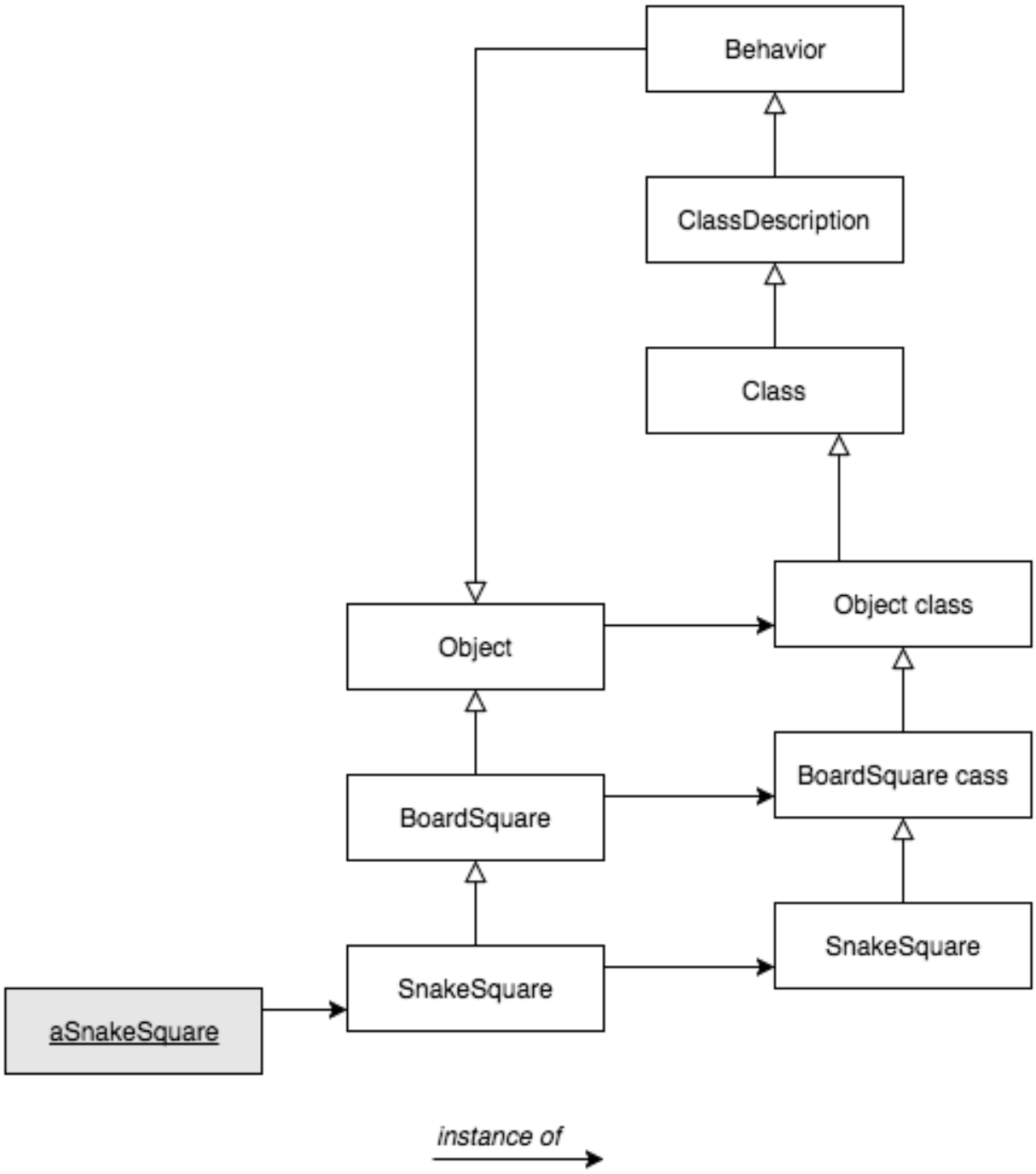


Per accedir a la *metaclasse* d'una classe s'ha d'activar el *class side* al Pharo.

4. La jerarquia de metaclasses és equivalent a la jerarquia de classes



5. Tota metaclasse hereta de Class i Behavior



Behavior

És el mínim estat necessari pels objectes que tenen instancies. Té l'interfície bàsica pel compilador.

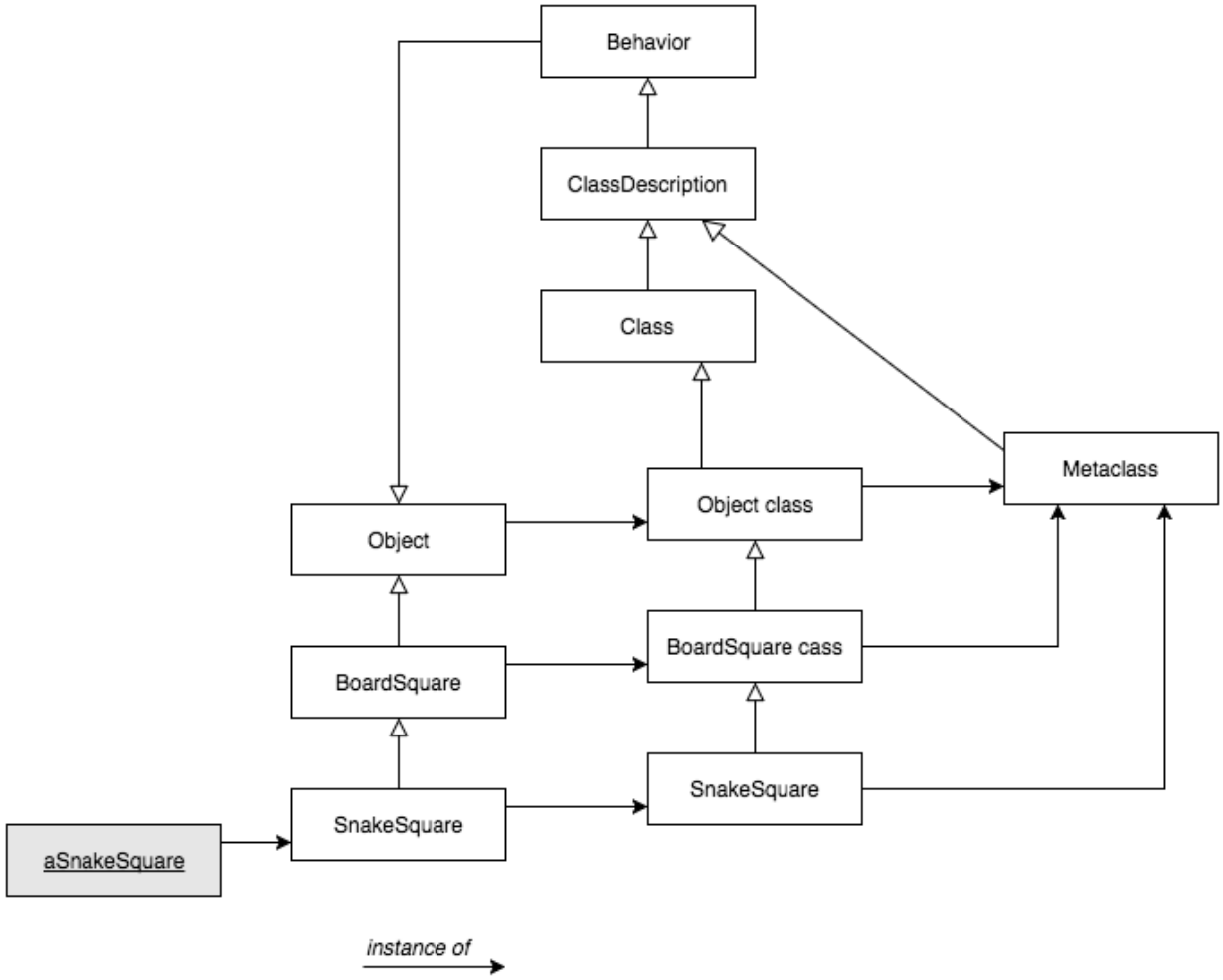
ClassDescription

Afageix algunes utilitats a *Behavior*. És una classe abstracte, les utilitats que proporciona estan pensades per *Class* i *Metaclass*.

Class

Representa el comportament comú de totes les classes (com, compilació, emmagatzematge de mètodes, variables d'instancia, etc).

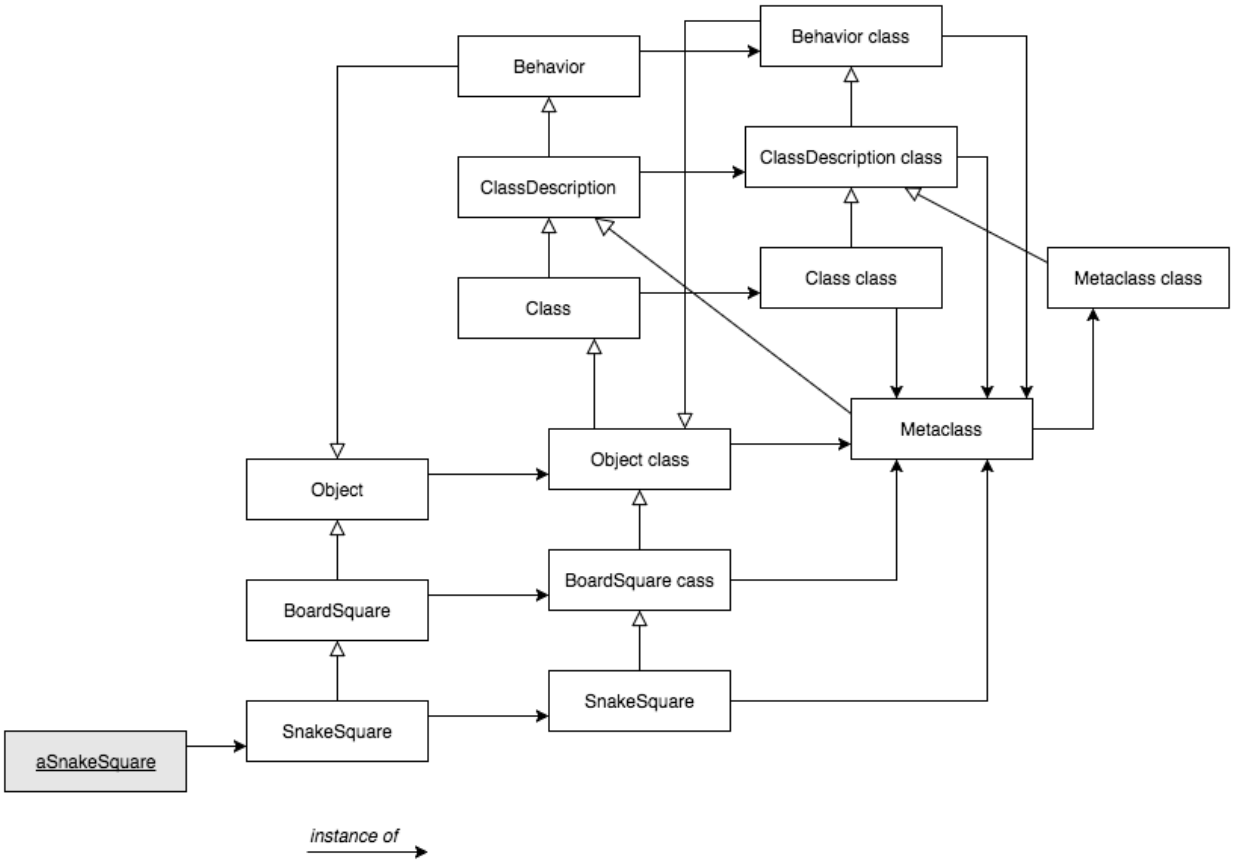
6. Tota metaclass és instància de Metaclass



Metaclass

Representa el comportament comú de totes les *metaclasses*

7. La metaclasse de Metaclass és instància de Metaclass




```
testHierarchy
    "The class hierarchy"
    self assert: SnakeSquare superclass = BoardSquare.
    self assert: BoardSquare superclass = Object.
    self assert: Object superclass superclass = nil.
    "The parallel metaclass hierarchy"
    self assert: SnakeSquare class name = 'SnakeSquare class'.
    self assert: SnakeSquare class superclass = BoardSquare class.
    self assert: BoardSquare class superclass = Object class.
    self assert: Object class superclass superclass = Class.
    self assert: Class superclass = ClassDescription.
    self assert: ClassDescription superclass = Behavior.
    self assert: Behavior superclass = Object.
    "The Metaclass hierarchy"
    self assert: SnakeSquare class class = Metaclass.
    self assert: BoardSquare class class = Metaclass.
    self assert: Object class class = Metaclass.
    self assert: Class class class = Metaclass.
    self assert: ClassDescription class class = Metaclass.
    self assert: Behavior class class = Metaclass.
    self assert: Metaclass superclass = ClassDescription.
    "The fixpoint"
    self assert: Metaclass class class = Metaclass
```

Classes Indexades i Variables d'Instància

Tenim dues maneres de representar objectes *Variables d'Instància* per utilitzar-los, amb nom o indexades

- Amb **nom** `name` de `GamePlayer.class`
- **Indexada** `#(Jack Jill) at: 1` seria "Jack".

Des del punt de vista més a baix nivell seria:

- Objectes amb referències a altres objectes (*pointer*)
- Objectes amb arrays de bytes (*word*, *long*)

Fem la diferència per raons d'eficiència: emmagatzemar arrays de *bytes* (com les strings de **C**) és més eficient que emmagatzemar un array de referències, cada una d'elles apuntant a un sol *byte* i ocupant una *word*

Una **variable indexada** s'afegeix implícitament a la llista de **variables d'instància**.

- Només hi ha una variable indexada (d'instància) per classe
- Accés amb `#at:` i amb `#at:put:`

Les subclasses d'una classe indexable han de ser també indexades

IndexedObject

Declaració de la classe:

```
Object variableSubclass: #IndexedObject
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'ClassesIndexades'
```

Exemple d'us:

```
(IndexedObject new: 2)
    at: 1 put: 'Fèlix';
    at: 2 put: 'Arribas';
    at: 1. " Print it => 'Fèlix' "
```

Implementació

Variables de classe-instància

Les classes són objectes, instàncies de la seva metaclasse, així que poden tenir variables d'instància.

Exemple: El patró Singleton

Volem que la classe sigui singleton (codi).

```
Object subclass: #Singleton
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'Patterns'
```

I al class side (metaclasse):

```
Singleton class
    instanceVariableNames: 'uniqueInstance'
```

Ara toca controlar la creació de lobjecte *Singleton* i l'access a *uniqueInstance*:

```
new
    "You cannot create a new singleton object"
    self error: 'Use uniqueInstance to get the unique instance of this object'

uniqueInstance
    "get the unique instance of this class"
    uniqueInstance isNil
       ifTrue: [ uniqueInstance := self basicNew initialize ].
    ^ uniqueInstance
```

Variables de classe

Serveixen per compartir informació entre instàncies d'una classe. Son variables compartides i direcament accessibles per totes les instàncies de la classe i la subclasse. Comença amb una lletra majúscula.

Reflexió

Introspecció

Fent introspecció a Smalltalk podem arribar a inspecionar objectes, consultar el seu codi i també accedir els contests d'execució.

Inspecció d'objectes

Una classe té un format, una superclasse i un diccionari de mètodes. Com hem vist abans cada classe té una *metaclasse* que es crea implicitament quan es crea la classe. On es crea aquesta metaclasse i la classe com a instància? Ho podem trobar a `SlotClassBuilder >> #buildNewClass`. En aquesta funció es crea un metaclasse nova `metaclass := Metaclass new.` i s'instancia creant una nova classe `newClass := metaclass new.`.

```
buildNewClass
    | metaclass newClass |
    metaclass := Metaclass new.
    metaclass
        superclass: self superMetaclass
        withLayoutType: FixedLayout
        slots: classSlots.

    newClass := metaclass new.
    newClass setName: name.

    newClass
        superclass: superclass
        withLayoutType: self layoutClass
        slots: slots.

    newClass declare: sharedVariablesString.
    newClass sharing: sharedPoolsString.

    installer classAdded: newClass inCategory: category.

    installer installTraitComposition: traitComposition on: newClass.
    installer installTraitComposition: classTraitComposition on: metaclass.
```

```
    ^ newClass
```

Cal distingir entre *metaobjectes* i *metaclasses*. Amb el nom ja ens ho podem imaginar. Una *metaclasse* és la classe de les classes. Una classe les instàncies de la qual són classes. En canvi un *metaobjexte* és un objecte que descriu o manipula altres objectes, per exemple:

- **Estructura:** `Behavior`, `ClassDescription`, `Class`, `Metaclass`, `ClassBuilder`
- **Semántica:** `Compiler`, `Decompiler`, `IRBuilder`
- **Comportament:** `CompiledMethod`, `BlockContext`, `Message`, `Exception`
- **Control de l'estat:** `BlockContext`, `Process`, `ProcessorScheduler`
- **Recursos:** `WeakArray`
- **Noms:** `SystemDictionary`
- **Llibreries:** `MethodDictionary`, `ClassOrganizer`

Aquests *metaobjectes* tene les seves *metaoperacions*. Les metaoperacions són les que ofereixen informació (o *metainformació*) dels objectes.

Utilitzem la *metaoperació* `instVarNamed:` per accedir a la variable d'un objecte pel nom i fem servir `put:` per canviar el seu valor.

```
| punt |
punt := 10@2.
punt class. "Point"
punt x. "10"
punt instVarNamed: 'x'. "10"
punt x: 4. "Point doesNotUnderstand x:" "No podem modificar la variable x de punt d'aquesta manera"
punt instVarNamed: 'x' put: 4.
punt "(4@2)"
```

També podem accedir a al *metainformació* (`Object >> #class`, `Object >> #identityHash`) I canviar-la (`Object >> #primitiveChangeClassTo:`, `ProtoObject >> #become:`, `Object >> #becomeForward:`).

primitiveChangeClassTo

Canvia la classe de l'objecte receptor del missatge per la classe del objecte argument. Les dues classes tenen la mateixa estructura a les seves instancies. Per això quan creem el metode `thisIsATest` *browser* ho entén pero un nou *Browser* no.

```
testPrimitiveChangeClassTo
    | behavior browser |
    behavior := Behavior new.
    behavior superclass: Browser.
    behavior setFormat: Browser format.
    browser := Browser new.

    browser primitiveChangeClassTo: behavior new.
    behavior compile: 'thisIsATest ^ 2'.

    self assert: browser thisIsATest = 2.
    self should: [ Browser new thisIsATest ] raise: MessageNotUnderstood.
```

become

Intercanvia totes les referències d'un objecte a l'altre i vice-versa. `punt1` passa a ser `punt3` i `punt3` passa a ser `punt1`.

```
testBecome
    | punt1 punt2 punt3 |
    punt1 := 0@0.
```

```
    punt2 := punt1.
    punt3 := 100@100.
    punt1 become: punt3.
    self assert: punt1 = (100@100).
    self assert: punt1 == punt2.
    self assert: punt3 = (0@0).
```

becomeForward

Intercanvia totes les referències d'un objecte a l'altre. `punt1` passa a ser `punt3`. `punt3` no canvia.

```
testBecomeForward
    | punt1 punt2 punt3 |
    punt1 := 0@0.
    punt2 := punt1.
    punt3 := 100@100.
    punt1 becomeForward: punt3.
    self assert: punt1 = (100@100).
    self assert: punt1 == punt2.
    self assert: punt2 == punt3.
```

Consultar el codi

A Pharo podem veure el codi de totes les classes i métodes gràcies al System Navigation, però ademés podem accedir a informació "interessant" sobre aquest codi com per exemple les subclasses (`subclasses`, també les subclasses de les subclasses amb `allSubclasses`), les linies de codi (`linesOfCode`), superclasses (`allSuperclasses`), etc.

Com hem vist abans, totes les classes són subclasse de *Behavior*. Behavior té un diccionari de metodes `MethodDictionari`, un diccionari de `CompiledMethod`. Podem accedir a aquests metodes accedint per nom a al diccionari.

```
5 factorial.
5 perform: #factiorial.
```

Accedir els contexts d'execució

La pila d'execució por ser reificada i maniupala. `thisContext` és una pseudo-variable que ens dóna accés a la pila.

Creem `Integer >> #factorial2` per veure el funcionament de la pila.

```
factorial2
    "Answer the factorial of the receiver."

    self = 0 ifTrue: [ thisContext explore. self halt. ^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial2].
    self error: 'Not valid for negative integers'
```

`thisContext explore` ens obrirà el context actual i `self halt` aturarà l'execució. Com la functió factorial és recursiva, quan ara fem, per exemple, `5 factorial` veurem totes les crides recursives.

> El metode `factorial2` està implementat al paquet *Reflexió*. Així que només cal executar al Workspace `5 factorial2`.

Com tot és un objecte modelitzem la pila d'execució amb objectes, concretament amb la classe `MethodContext`. Aquesta classe gestiona l'espai associat a l'execució d'un `CompiledMethod` (PC, el mètode en si, *sender* i *receiver*). El *sender* és el previ MethodContext.

Al aturar el context amb el mètode `halt` hem d'anar amb cuidado. No podem posar `halt` en mètodes que s'utilitzen sovint ja que es començarà a aturar tot. Podem crear el metode `haltIf`, que s'atura només si el mètode ha estat invocat des d'algun altre amb un selector determina:

```
haltIf: aSelector
    | context |
    context := thisContext.
    [ context sender isNil ]
        whileFalse: [
            context :=  context sender.
            (context selector = aSelector)
                ifTrue: [ Halt signal ]
        ].
```

```
foo
    self haltIf: #fighters.
    ^ 'foo'
```

```
fighters
    ^ (self foo), 'fighters'
```

En el mètode `#foo`, diu que faci `halt` si `#foo` és cridada desde `#fighters`. El mètode `#figthers` crida a `#foo`, al fer `#foo` `#fighters`, salta.

```
HaltDemo new foo. " 'foo' "
HaltDemo new fighters. " fa Halt"
```

Halt

```
HaltDemo        haltIf:
HaltDemo        foo
HaltDemo        fighters
```

BlockWithExit

```
Object subclass: #BlockWithExit
    instanceVariableNames: 'block exitBlock'
    classVariableNames: ''
    category: 'Reflexio'
```

BlockWithExit class

```
with: aBlock
    ^ self new with: aBlock
```

BlockWithExit

```
with: aBlock
    block := aBlock
```

```
value
    exitBlock := [^ nil].
    ^ block value.
```

```
exit
    exitBlock value
```

BlockClosure

```
withExit
    ^ BlockWithExit with: self
```

Exemple

```
| theLoop coll |
Transcript open.
coll := OrderedCollection new.
1000 timesRepeat: [ coll add: 1000 atRandom ].
theLoop := [
    coll do: [ :each |
        Transcript show: each asString; cr.
        (each < 100)
            ifTrue: [theLoop exit]
    ]
] withExit.
theLoop value.
```

BlockWithExit és una variant de **BlockClosure** que permet sortir de la closure cridant exit. En aquest codi es crea una llista de 1000 elements amb un nombre aleatori entre 0 i 1000. En el bloc de `theLoop` s'itera per tota la llista (`coll do: [ :each |`), es mostra el valor (`Transcript show: each asString; cr.`) i si el valor és menor de 100 (`(each < 100) ifTrue:`) es surt del bloc (`[theLoop exit]`).

Com el bloc que es crea és el per defecte de Smalltalk (`BlockClosure`) afegim el metode `#withExit` que crea un `BlockWithExit`.

Intercessió

Sobreescriure `#doesNotUnderstand:`

Cal crear un objecte mínim. Embolica un objecte normal (*wrap*), no enten quasi res i redefineix `#doesNotUnerstand:`. És superclasse de `nil` o `ProtoObject` per no tenir la implementació normal de `#doesNotUnerstand:`. Finalment utilitxa el metode `#become:` per substituir i controlar l'objecte a controlar.

```
ProtoObject subclass: #LoggingProxy
    instanceVariableNames: 'subject invocationCount'
    classVariableNames: ''
    category: 'Reflexio'
```

La idea es col · locar aquest objecte entre el missatge i l'objecte receptor (`receiver`).

```
initialize
    invocationCount := 0.
    subject := self.
```

La variable d'instància `subject` serà on enviarem el missatge quan l'objecte no l'entengui:

```
doesNotUnderstand: aMessage
    Transcript show: 'performing ', aMessage printString; cr.
    invocationCount := invocationCount + 1.
    ^ aMessage sendTo: subject
```

Exemple

```
testDelegation
    | point proxy |
    point := 1@2.
    proxy := LoggingProxy new.
    proxy become: point.

    self assert: point class = LoggingProxy.
    self assert: proxy class = Point.
```

```
    self assert: point invocationCount = 0.

    self assert: point + (3@4) = (4@6).
    self assert: point invocationCount = 1.
```

Quan `point` es transforma en el proxy només sap fer `#doesNotUnderstand`. Ja no és `(1@2)`, el proxy passa ser-ho. Quan se li envia el missatge `#+` a `point`, ja no l'entén i executa `#doesNotUnderstand`. A `#doesNotUnderstant`, point escriu pel `Transcrip`, incrementa el `invocationCount` i finalment envia el missatge al `subject`.

La variable `subject` s'ha inicialitzat amb `self`, és a dir, `proxy`. `proxy` s'ha transformat en el punt `point`, ho podem veure en el primer `#assert:`. Aixi que envia el missatge a `proxy`, que ara és el punt `(1@2)` i si que l'enten.

Getters "on demand"

És pot sobreescriure el mètode `#doesNotUnderstand` per generar codi dinàmicament. Un cop la classe reb un missatge que no entén comprova si alguna de les seves variables d'instància té el nom del missatge, és a dir, s'està demanant pero com no te *getter* dóna error. Si la variable existeix compila `#nom ^ #nom` i l'executa. Si no existeix continua amb l'execució normal de `#doesNotUnderstand`.

```
doesNotUnderstand: aMessage
    | messageName |
    messageName := aMessage selector asString.
    (self class instVarNames includes: messageName)
        ifTrue: [self class compile: messageName , String cr , ' ^ ' , messageName.
            ^ aMessage sendTo: self].
    super doesNotUnderstand: aMessage
```

Classes anònimes

Consisteix en crear un instancia de `Behavior`, definir els mètodes i posar-la entre la instància i la classe. Una classe anònima permet un control sel · lectiu, no dóna problemes amb el `self`, és eficient i dóna transparencia a l'usuari.

```
| casseAnonima set |
casseAnonima := Behavior new.
casseAnonima superclass: Set;
    setFormat: Set format.

casseAnonima compile:
    'add: anObject
        Transcript show: ''adding '', anObject printString; cr.
        ^ super add: anObject'.

set := Set new.
set add: 1.

set primitiveChangeClassTo: casseAnonima new.
set add: 2.
```

El primer `#add` és normal, el segón és el compilat en el codi i mostra "*adding 2*" pel Transcript.

Per que aquest codi funcioni cal crear aquests dos nous mètodes a la classe `TBehavior`

```
basicLocalSelectors
```

```
    ^nil

basicLocalSelectors: aSetOrNil
    self subclassResponsibility
```

Method Wrappers

La idea és poder executar codi abans i despres de que s'executi el mètode que s'invoca. Es substitueix el mètode per un objecte que implementi `#run:with:in:`.

```
Object subclass: #LoggingMethodWrapper
    instanceVariableNames: 'method reference invocationCount'
    classVariableNames: ''
    category: 'Reflexio'

initializeOn: aCompiledMethod
    method := aCompiledMethod.
    reference := aCompiledMethod methodReference.
    invocationCount := 0

run: aSelector with: anArray in: aReceiver
    invocationCount := invocationCount + 1.
    ^ aReceiver withArgs: anArray executeMethod: method
```

Apart d'això tenim el mètode `#install` (i `#uninstall`, que és molt similar) que fa el *wrap* del mètode.

```
install
    reference actualClass methodDictionary at: reference methodSymbol put: self
```

Exemple d'execució:

```
logger := LoggingMethodWrapper on: Integer>>#factorial.
logger invocationCount. "0"
5 factorial.
logger invocationCount. "0"
logger install.
[ 5 factorial ] ensure: [logger uninstall].
logger invocationCount. "6"
10 factorial.
logger invocationCount. "6"
```

Al fer *wrap* d'un mètode totes les instàncies queden controlades, només s'intercepten els missatges conegurs (es pot controlar només un sol mètode) i no cal compilar per instal · lar.

Continuacions

A Pharo 3.0 tenim la classe `Continuation` que serveix per guardar la pila d'execució en un moment donat.

```
Object subclass: #Continuation
    instanceVariableNames: 'values'
    classVariableNames: ''
    category: 'Kernel-Methods'
```

Per instanciar aquesta classe cal fer servir `#initialitzeFromContex: aContext`. Aquesta funció guarda a la variable d'instància `values` la pila associada al context que es passa com a argument.

```
initializeFromContext: aContext
    | valueStream context |
    valueStream := WriteStream on: (Array new: 20).
```

```
    context := aContext.
    [context notNil] whileTrue:
        [valueStream nextPut: context.
        1 to: context class instSize do: [:i | valueStream nextPut: (context instVarAt: i)].
        1 to: context size do: [:i | valueStream nextPut: (context at: i)].
        context := context sender].
    values := valueStream contents
```

Quan cridem a inicialitzar una nova continuacio amb un context, a partir del context donat (`context := aContext.`) itera per la seva pila d'execució fin al final (`[context notNil] whileTrue: [ ... context := context sender].`) i afageig el context (`valueStream nextPut: context.`) i les variables d'instància (`1 to: context class instSize do: [:i | valueStream nextPut: (context instVarAt: i)].`) i de classe (`1 to: context size do: [:i | valueStream nextPut: (context at: i)].`).

La classe `Continuation` té un mètode anomenat `#value: anObject` que, donat un objecte, recupera el context que teniem guardat, el converteix en el context actual i retorna l'objecte `anObject` per poder continuar l'execució del context tot just restaurat.

```
| lletra |
lletra := Continuation new initializeFromContext: thisContext.
(lletra = $f)
    ifTrue:  [
        Transcript show: 'és la lletra f' ; cr
    ]
    ifFalse: [
        Transcript show: lletra ; cr.
        lletra value: $f
    ].
Transcript show: lletra ; cr.
```

Si executem aquest codi el que sortirà pel Transcrip és:

```
a Continuation
és la lletra f
f
```

El que fa aquest codi és crear una continuació amb el context actual i asignar-li a `lletra`. Després comprova si `lletra` és la lletra f, obviament no ho és perque acabem de dir que és una continuació aixi que executa el bloc `ifFalse`. Dintre d'aquell bloc mostra pel Transcript *a Continuation* i li dona el valor `$f` a la continuació `lletra`.

Allà és quan hi ha el canvi de context. Com he explicat abans el mètode `#value: anObject` recupera el context que teniem guardat, el converteix en l'actual i retorna `anObject`. Així doncs tornem a la linia on li assignavem `Continuation new ...` a `lletra` i retornem `$f`. Al convertir `thisContext` en el context actual l'executó segueix a partir d'alla. Entrarà al bloc `ifTrue` i mostrarà *és la lletra f* i *f*.

Evaluar continuacions: `#value: anObject`

```
value: anObject
    self terminate: thisContext.
    self restoreValues.
    thisContext swapSender: values first.
    ^ anObject
```

Aquest mètode, primer de tot elimina el context actual, per tant deixem d'estar en el moment en que s'ha cridat `#value:`. Acte seguit recupera la pila del moment en que s'ha inicialitzat la continuació. Finalment diu que el context actual és el primer de la pila cargada i retorna el valor que se li ha passat al mètode.

Call current continuation: `#callcc: aBlock`

```
callcc: aBlock
    ^ self currentDo: aBlock
```

La idea d'aquest mètode és capturar el context actual en una instància i passar-li una continuació d'aquest context com a paràmetre a `aBlock` quan és evaluat.

WHAT!?... Exacte, que?

`#currentDo: aBlock` evalua el block donat amb el resultat de `self fromContext: thisContext sender`, és a dir, una continuació del context del `sender` (és el `sender` perque si no el context seria aquella mateixa linea del `currentDo` i no és el que volem). `fromContext` crida a `#initializeFromContext:`, i ja sabem que fa.

```
currentDo: aBlock
    ^ aBlock value: (self fromContext: thisContext sender)
```

```
fromContext: aStack
    ^self new initializeFromContext: aStack
```

Recapitulem: al invocar `callcc` amb un bloc B, evaluem el bloc B amb la continuació resultant del context on es crida `callcc` (`thisContext sender`).

Exemples de `#callcc: aBlock`

```
| x |
x := Continuation callcc: [ :cc | cc value: true ].
x "print => true"
```

Al cridar `callcc` el que fem és evaluar el bloc `[ :cc | cc value: true ]` amb una continuació d'aquell context (aquesta continuació es crea dintre de `callcc`, a `currentDo`). Al evaluar el bloc amb la continuació, evaluem la continuació amb `true`, és a dir, x val true.

Podem afegir uns `Transcripts show:` per veure cuin és l'ordre en que s'executen les coses:

```
| x |x := Continuation callcc: [ :cc |  Transcript show: 'Primer'; cr.  cc value: true. Transcript show:
```

```
El resultat d'aquest codi serà:
```

Primer Segon true "'

Un altre exemple per entendre el valor de `cc` i `x`:

```
smalltalk| cont x |x := Continuation callcc: [ :cc | cont := cc. cont value: 1 ].(x = 1)
ifTrue: [       Transcript show: 'x = '.        Transcript show: x; cr.     cont value: 2
] ifFalse: [      Transcript show: 'x = '.        Transcript show: x; cr. ].
```

El resultat d'aquest codi serà:

```
x = 1
x = 2
```

El evaluar el bloc del `callcc` guardem la continuació a la variable `cont` i l'evaluem amb valor *1*. Continuem i mirem que x sigui igual a *1*, obviament ho és perque acabem de donarli aquell valor. Dins del bloc del `ifTrue:` es torna a evaluar la continuació amb valor *2*.

```
mentreCert: aBlock
    "versió de whileTrue: implementada amb callcc:"
    | cont |
    cont := Continuation callcc: [ :cc | cc ].
    self value
```

```
ifTrue:  [ aBlock value.
           cont value: cont ]
ifFalse: [ ^ nil].
```

En aquest codi volem fer un bucle mentre el bloc `self` sigui cert. Així doncs, creem una **continuació que serà evaluada en el bloc `[ :cc | cc ]`**. Després evaluem `self` (la condició del *whileTrue*), evaluem el bloc `aBlock` si és cert o sortim (amb `[ ^ nil ]`) si es fals. En cas de que sigui cert, després d'evaluar el bloc `aBlock`, **evaluem `cont`** amb `cont` com a valor. Això li passa `cont` al bloc `[ :cc | cc ]`. Aquest bloc retorna `cc`, el *value* que li passis. Aixi que `cont` serà **la mateixa continuacó d'abans, que serà evaluada en el bloc `[ :cc | cc ]`**. Tindrà el context d'aquell moment, l'execució continuarà desde aquell punt i tornarà a evaluar-se `self` com en un `whileTrue`.