

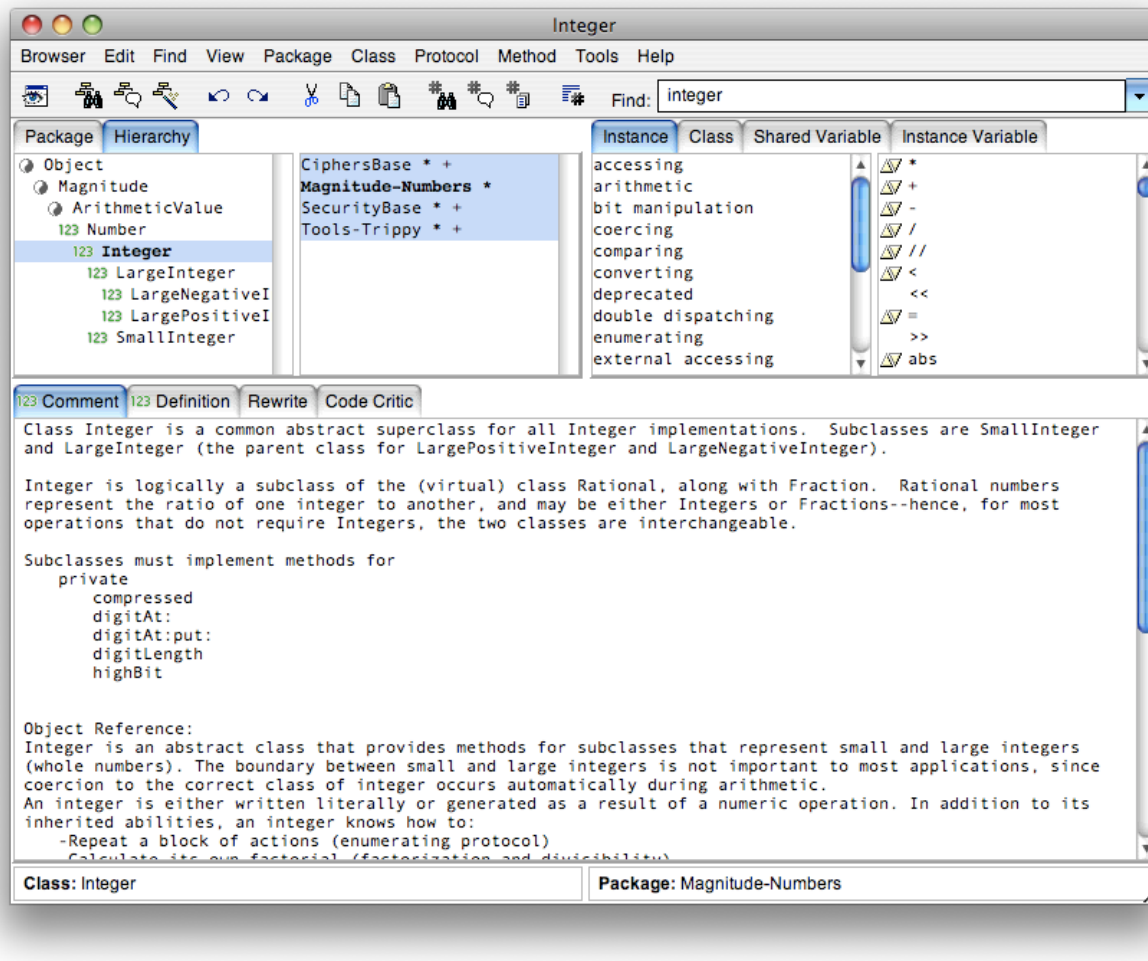
smalltalk 勉強会

2009.2.4

第三章

p.50, 3.1
is a kind of, self, super のことを。

Integer をシステムブラウザで探してみると以下ようになる。



右側に注目。

<
<<
=
>>

の上矢印があれば、それは上に何かがある、ということか？

下矢印があれば、下に何かある、と。

矢印がない、ということはここでだけ定義されている、ということ。

何があるんだ？聞き落とした。辛い。

self はそれが定義されている自分自身に対して、ではなくて、それを呼び出したインスタンスに対して投げる、ということになる。

super はそれが定義されているところの superclass を探しに行く。つまり super を呼び出したはずのインスタンスから見て super などところを探しに行くわけではない。

この意味的なギャップは非常に妙だが、単にそういう機能が欲しかっただけ、のように僕には見える。

super が上位継承を実現するためにあるのは容易に想像が付く。

...前処理 ...

super process.

...後処理

subClassResponsibilities これより下のクラスのどこかで定義しなさい、という意味
shouldNotImplement これより下ではこれをインプリメントしてはいけない、という意味

実際にはエラーダイアログが出るだけ、らしい。
文法、というかシステム

なお self は
self message の部分に
self classname.message (つまり自分自身から見た message 名のメッセージではなく、クラス名を指定して、その message を取ってこい、というようにできる)
ただこれは vm がそのようになっているだけで、コンパイラがそれを断るので実際には動かない。バイトコードにはそういう機能はないが、vm 中のメッセージルックアップ経路を探すコードにそういったものが残っている。)
探索経路を固定指定してやるもんだわなあ。
これだと Java がやっている、class.class.class.method と同じようなことができる。

3.2 コメント

ダブルクォートでくくる。" sample "
Smalltalk ではプログラムの断片にひたすら " " が
Smalltalk のコメントは一人称で書くものらしい。つまり「私は**をします」と書く、ということらしい。そのオブジェクトになりきって書く、ということか。
三人称をつかって書いてはいけない、らしい。慣習として。

よく、メソッドの先頭にコメントとして、こうやって動かすと良いよ、というもの(実行テンプレート)が書かれている。
ピリオドを付けるか付けないかで違いがあるらしい。
(JunOpenGL3dObject panda) show.
とかいった感じ。
show の最後のピリオドがあると「文」だと思われ、メッセージ式つまり「式」だとは思われない、ということか。
これは文である、という主張だそうだけど。
ただし workspace でフォーマットすると消されるらしい。
ピリオドを付けるのは青木さんくらいで、他は少ないらしい。

3.3 定数

定数 (literal) はオブジェクトを書き記す直接的な方法が提供されているものである、とある。定数ではあるが、Smalltalk のオブジェクトであり、メッセージに応えることができる。

リテラルにブロックが含まれていないのは浅岡仕様らしい。青木的にはブロックはリテラルに入る。
プログラムはすべてブロックとして書くが、これはリテラルとして解釈される。つまりオブジェクトがそこに出来ている。ブロックというクラスに new を送ることなく出来たのだ。

整数はその基数を指定できる。

16rA

を workspace で inspect it すれば良い

50rA なんかもそのまま動く。

36rZ printStringRadix: 35

などとすると (printIt)、36 進数の Z を 35 基底で出力すると、'10' とか出てくる。

p.56 にあるプログラムは 2 進数から 35 進数までがループで試されるが、実際には 36 以降はぐちゃぐちゃになるという。サポートできない、ってことかなあ。

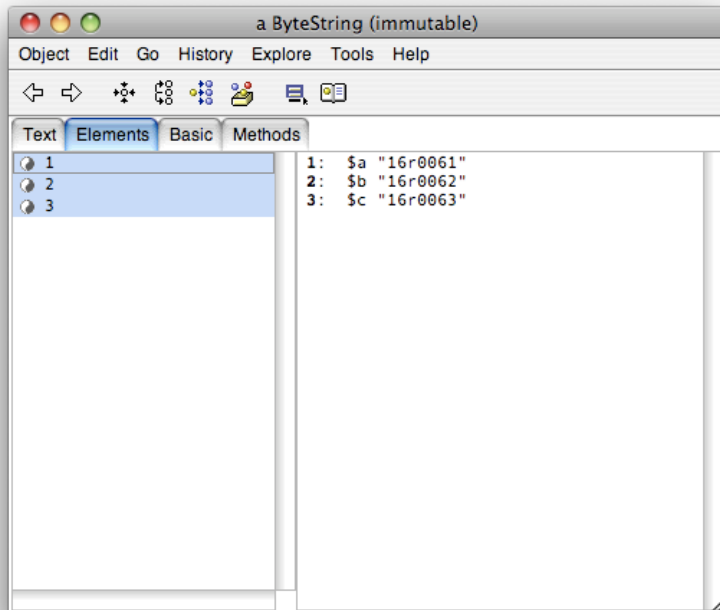
試してみい、と。

単精度、というのは IEEE でいうところの単精度。つまり 6,7 桁有効数字。

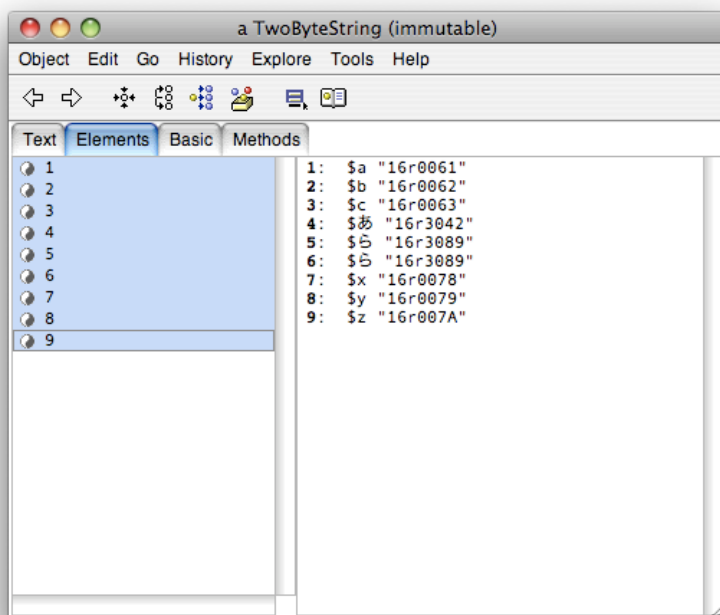
固定小数点形式について、123.45s9 とか書いて、inspectIt とすると、内部状態が出てくる。そこには分子・分母の表現形式になっていることがわかる。
これが出来るんだったら全部そうすりゃいいじゃん、と思うんだけどまあ遅いんだな。
青木さん、分数形式をハードウェア (シリコン) でサポートすればいいのに、と言う。そりゃそうだ。。が。。RISC バイブラインの時間列を乱す数値演算を話をやるかと言われるとイヤだろうなあ、、と僕は思う。。

文字が \$a とか \$ 記号で表記されるのがなかなか面白い。\$x が C などと言う 'x' なんだな。。
Smalltalk は 'abc' で文字列となる。

'abc' printIt とすると以下のような感じ。



内部 Unicode なので、そのまま二バイト文字列で出てくる。
'abcあらxyz' でやると、以下のような感じ。普通に二バイト文字列。



ただし英字だけだと（ウィンドウのタイトルを見よ）ByteString、漢字があると TwoByteString となる。つまり属性については違っている。が、これは恐らく内部 Unicode で無いときへの互換性のためだろうなあ。

シンボルは単語のあたみに # を付けてやっている。#Smalltalker というような感じ。

シンボルはコンパイラシンボルだと思えばいい、ようだ。コンパイラのメモリの中にだけ Smtaltalk というバイト列が登場する。コンパイルが終わったら残らないはずだ。が、原初はそうだったが、いまは String との相互変換などという恐ろしいことが出来るようで、つまり実行時のメモリとしてできている。

恐るべき事に今は Symbol は String のサブクラスとなっている。(@_@!) うへえ気色悪い。

#Smalltalker == #Samltalker を printIt すると true となり、
'Smalltalker' == 'Samltalker' を printIt すると false となる。

（ #Smalltalk hash を printIt すると hash code というかつまりオブジェクトインデックスが出てくるので、それで確認できる、、、と思ったがうま

くいかず。
昔は asOop)

== はオブジェクトの同一性を試すものなので、つまり #Smalltalker というシンボルはシステム中にたった一つだけ作ったものだ、ということ。

値の同一性、同値性、というらしい。を試すのは = なので、これでやると共に = となる。

ところで元々はシンボルはメッセージハンドラのために出来た模様。つまりメッセージはこれで書く、ということか。

==== Appendix に出ている話へと。

Smalltalk の処理系はコンパイラ、インタプリタ、VM などがハイブリッドになって構成されている。
ソースコードはコンパイラがパーサ（字句解析、構文解析）して構文木+名前表にして意味解析をして解析木を作り、そこから最適化などをしてコード生成してバイトコードを得る。

Smalltalk の VM はスタックマシンなのだ。スタック+レジスタ（一時変数名指し）で動くモデル。

が、今回吐いたものは最適化してない、ねえ。きっと。

===== 第四回

-- バイト配列から pp.67-

```
#[1 2 3 4 5 34]
```

など。

-- ブロック・クロージャ

[] で囲むと、それ以内をコンパイルしてください、ということになる。

「ブロック・クロージャ」と呼ぶ。

つまりブロック・クロージャで書いて誰かに渡すことができるということか。

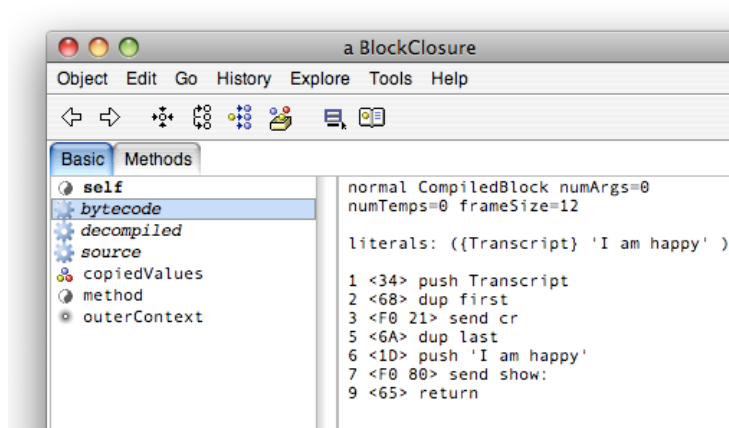
その結果を inspect it すると、内部の method が出てくる。そこに bytes という構造が含まれていることがわかるが、それがバイトコードそのもの。

なおそこに bytecode という構造もあるが、これは斜字で出ており、これはつまりバイトコードを逆アセンブルしたもの。

その下にはデコンパイルした結果が出ている。これらの斜体文字で出てくるのは実体はない模様。

```
| aBoolean |
aBoolean := JunDialog confirm: 'are you happy?' .
aBoolean
  ifTrue:
    [Transcript
      cr;
      show: 'I am happy'].
```

として、ブロッククロージャを inspect it すれば、



てな具合。（あれ、ここには bytes がないぞ）

-- 配列

```
#(100 200 300)
```

こんな感じで何でも書ける。inspect it すると構造が分かって面白い。

```
#(100 101 #(200 300)) とか #(1 2 3 'Smalltalk') とかりテラルであれば何でもはいる。
```

```
| anArray |
anArray := Array new: 2.
anArray at: 1 put: #(100 200).
anArray at: 2 put: #('Smalltalk' 1 2 3 ).
anArray inspect.
```

てなプログラムを動かしたり。
ううむ。最初に new: 2 などとしないといけないのか。可変長じゃないんだ。
動的に大きくするのはちょっと工夫が要るらしい。後述。

```
| anArray |
anArray := Array with: 128.
```

てな感じで書くと、

```
| anArray |
anArray := Array new: 1.
anArray at: 1 put: 128.
```

と等価なのよ。二つ要素を放り込みたければ、なんと、

```
| anArray |
anArray := Array with: 128 with: 129.
```

こんな書き方をする。ところがこの with with でつなげられるのは 4 つまで。
VM がなんせ引数四つまでを非常に速く send するような構造になっているらしい。
あらかた引数は四つまでだろう、という予測で作られているらしい。

五つ以上やるのであれば、やはり new: 5 などとして、at: で指定しながら処理することになる。

動的に大きくする方法としては、copywith: がある。

```
| anArray |
anArray := Array with: 128 with: 129.
```

などとして作った後に、anArray copywith: 130. などとする。が、これは配列を大きくしているわけではなく、元の配列を copy（複製）している。
ただし、これと一緒に、つまり with というわけ。

```
| aStream |
aStream := WriteStream with: (Array with: 129 with: 130).
aStream nextPut: 100.
```

うう、うまくできない。何か違うらしい。まあしかしどっちにしても配列を大きくするようなことはない。内部的には、だけど。

#(100 200 300) てな感じでリテラルとして書いたものは immutable object になるので、いじれない。
例えば初期値として 100 200 300 の三つ組みを渡して、後でそれらをカウントアップするようなプログラムを書こうとすると、初期値を
#(100 200 300) copy などとして（mutable な配列として）与えれば良い。

— 束縛参照

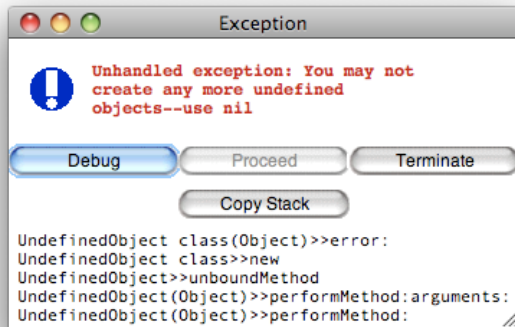
```
{Jun}
```

参照しているオブジェクトを指しているわけではない。
値を参照しているわけではなく、その binding（束縛、というがくくり付け関係）を参照するのだ。と、いうが、まあ深入りしない。（pp.33）
なんのこっちゃらわからん。内部的に使っているらしいが。。

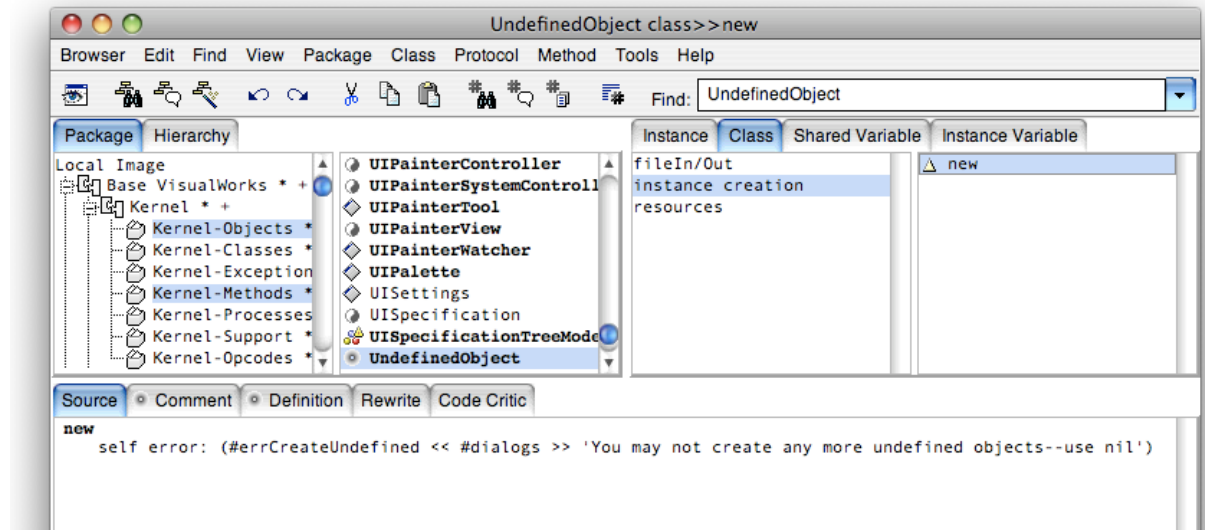
— nil

UndefinedObject クラスのシングルトン（唯一のインスタンス）。未定義要素として使われる。
Array new: 5 とすると、要素にはとらずに nil が五つ入っている。
nil の対義語？は Object new にあたる。nil はこの Object new（anObject）の中に唯一含まれないもの、ということになる。

UndefinedObject new を do it とすると、以下のダイアログが出る。なんと「use nil」とまで出てくる！

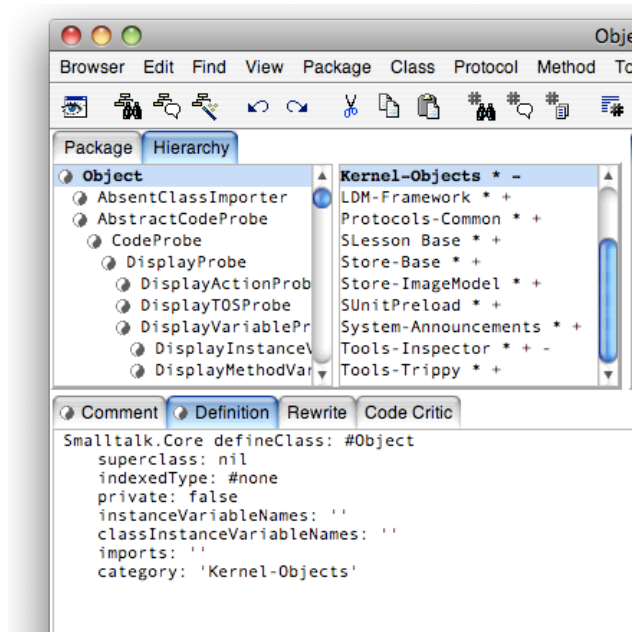


へんだねえ。
UndefinedObject superclass
Object
となる。で、
Object new
an Object
となるわけだから、結局受けるはずなんちゃん。
えいやあとクラスブラウザで見ると、



というわけで、あらまあ。エラーメッセージがまともにハードコードされている。えへへ。
つまりシングルトン、というのは設計上の用語で、言語仕様の呼び名ではないわけだ。が、Smalltalk は極言すると言語仕様は VM 以上であればどこまでがシステム環境で、どこまでが言語仕様か、という境界がないように僕には見えている。その意味ではこの「まるでユーザが自分でコードして作った」ように見えるシングルトンは、まあなかなか良いではないか。

が、妙な話で、Smalltalk では全てのオブジェクトが nil から産まれたことになっている。試しに Object クラスの Definition を見ると、superclass はなんと nil になっている。ところが nil は UndefinedObject の instance だというからもうわけがわからないっす。。。 (深入りしないように)



—— true

true は真を表す、True クラスのシングルトンなのだ。（では True に new を送ってみるか、くらいが良い態度だねえ。）

```
| aBoolean |
aBoolean := JunDialog confirm: 'are you happy?'.
aBoolean
  ifTrue:
    [Transcript
      cr;
      show: 'I am happy'].
```

てな感じ。お。[] でくっついてある。ブロッククロージャだ。こうやって使うのだ。

false も同様。

Boolean クラスの and: メソッドは ifTrue: メソッドと同じ。両者は同義なのだ、と。へえ。まあそうだねえ。
or: は ifFalse: で実装できるわなあ。順に評価しようとするよね。

—— 偽変数

リテラル: nil, true, false
変数: self, super, thisContext
というような区分けになっているが、これら全部をひっくるめてシュード変数、と言うか、な、と。
まあでもイマドキは pseudo variable とかは言わないらしい。

ブロック（ブロッククロージャ）を青木さんはリテラルだ、と思うらしい。まあコンパイルオブジェクトだからリテラル（immutable object だわ）。

—— tips

self class と書いて do it すると、クラスが知れる。（UndefinedObject と出る）

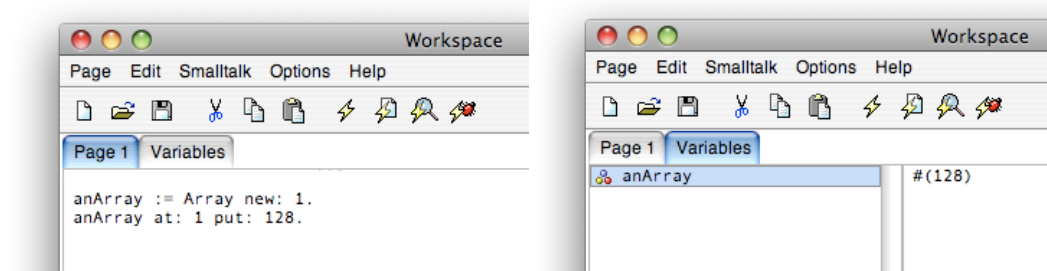
----- 変数（ようやっとだ！あと 20 分しかないよ！）

変数については有効範囲（スコープ）と寿命（ライフタイム）が幾らかある。
慣習的に Camel Case で名付けるが、aVariable というように小文字で始める場合は狭い領域で使うもの（インスタンス名、メソッド名、一時変数名）、Variable のように大文字で始める場合は広域で使うもの（クラス名、帯域変数名）、というように慣習がある。

— 一時変数 (temporary variable)

縦棒 || でくくる。そのプログラム単位の中でだけ有効になる。ライフタイムはプログラム実行時だけ。

なお workspace で一時変数を定義しないで動かした場合、勝手にその変数を一時変数のつもりで workspace 無いに登録してしまう。（勝手に）



てな感じで、知らないうちに「Variables」タブなんてもののなかに勝手に入ってる！
恐ろしい。

第四回 Smalltalk における数論（というか数の処理）

bind 束縛について、名前と値を結びつける、ただし実際には名前へのポインタと値へのポインタをくくる状態。

分数は Fraction という型（クラス）である。と。

サンプルにある、

```
aSelector := #+.
```

の + は # つまりシンボルの + となっている。そりゃなんだ？

```
aValue := x perform: aSelector with: y.
```

perform: は万能関数、とか言う奴で、100 perform: #+ with: 123 は 100 に + を送りつけろ、123 をくっつけてな、という意味。
perform with でマル覚えするのがよい？

switch case 分に相当する処理が掛ける、というのが。は一。しかしこれは関数を引数にしているようなものだなあ。

重送信（なぜ型を気にしないで済むのだろう）

MethodCollector new allImplementorsOf: #+.

を Inspect it すると、+ が実装されているクラスが出てくる。20 個くらい？
が、Jun が多いので、つまり VisualWorks では十数カ所にしかない。

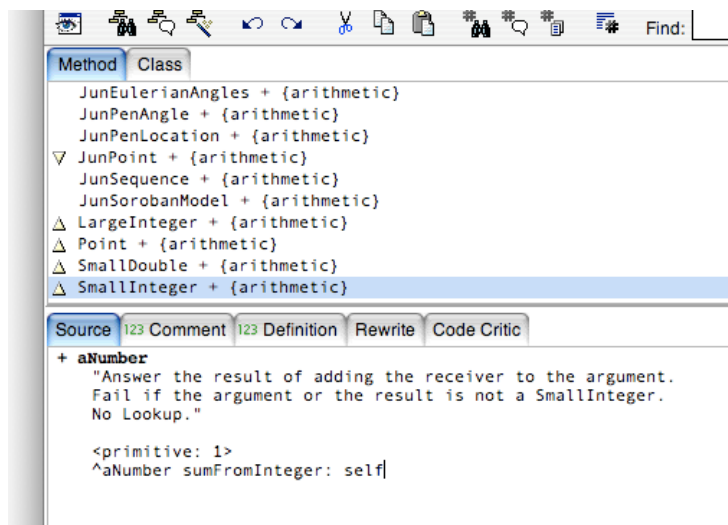
```
| messageSelector methodDefinitions |
messageSelector := #+.
```

```
methodDefinitions := MethodCollector new allImplementorsOf: messageSelector.
```

```
Refactory.Browser.RefactoringBrowser
  openListBrowserOn: methodDefinitions
  label: 'Implementors of ' , messageSelector printString
  initialSelection: messageSelector.
```

```
^methodDefinitions
```

を Dolt して、その結果のなかから SmallInteger を見ると、げげげ、<primitive: 1> などというものがある！



SmallInteger は 32bit 使う。内部は sign bit (1) + 29 bit data + tag (2bit) でなっている。つまり値としては 30bit 整数。

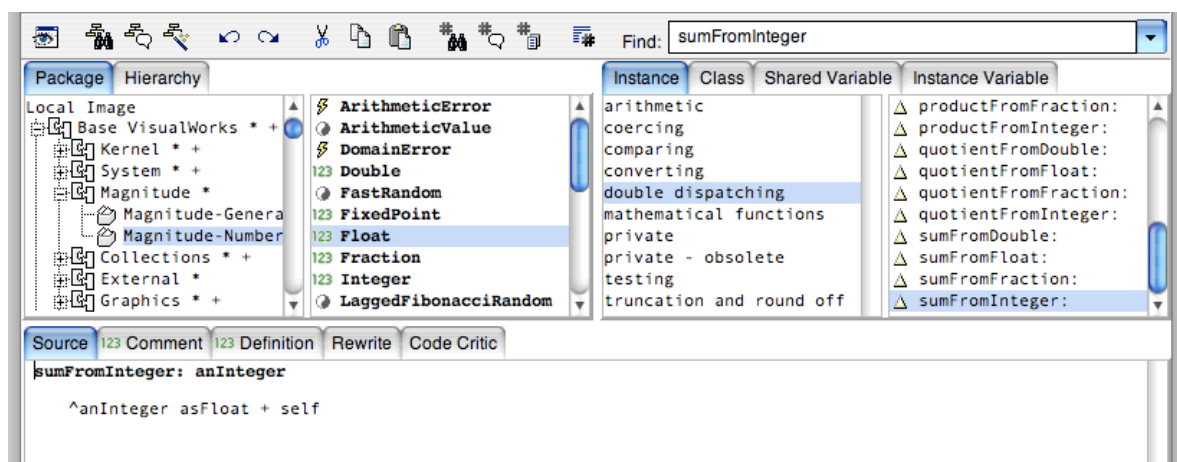
この tag にビットが何か立っているとそれは整数なので、それを見ると足し算として実行する。
ポインタが来る場合は 32bit システムなので、アドレス最後の 2bit は 00 でしかない。
(つまり 4Byte 境界でアライメントされるから)

で、上の <primitive: 1> は SmallInteger の加算に相当する。
この次の行、^aNumber sumFromInteger: self は、上の <primitive: 1> に失敗したときだけ実行する。(ドコにもそう書いてないがそういう仕様なのだ！)
で、SmallInteger でなければ <primitive: 1> は失敗し、その結果、元の値 (のオブジェクト) に対して sumFromInteger: を送る。引数として self をつけて。(つまり 100 + 123.5 を実行すると、(primitive: 1 は失敗して) 123.5 に対して、自分自身 (100) を添えて、+ メッセージを投げ返すことになる。
それでたとえば Float の + 処理実装部分を探し出すことが出来るはずだ、ということになる。

面白い仕掛けだなあ。

(同じ型どうしの演算は全部 primitive になっているらしい。ほんま？ そんな事せんでもええんちゃうの？ 単に型チェックしてからやるより高速だからやっただけでしょうに。。。)

今度はそれを追いかける。確かに Float で sumFromInteger メソッドが実装されているだろうか？
System Browser で sumFromInteger: メソッドを搜してみる。Float クラスのものが以下の通り。



おっと。引数だった anInteger に asFloat を投げている。これは単純な型変換で 100 asFloat で 100.0 にできる。
これで Float 型に直して、+ self つまり自分自身 (この場合は 123.5 だった) を足す。

つまり 100 + 123.5 は 123.5 + 100 に変化し、100 を 100.0 に合わせて (型変換して)、Float どうしの加算によって処理が行われる。

この場合は型は合っているはずなので、これ以上は要らない。

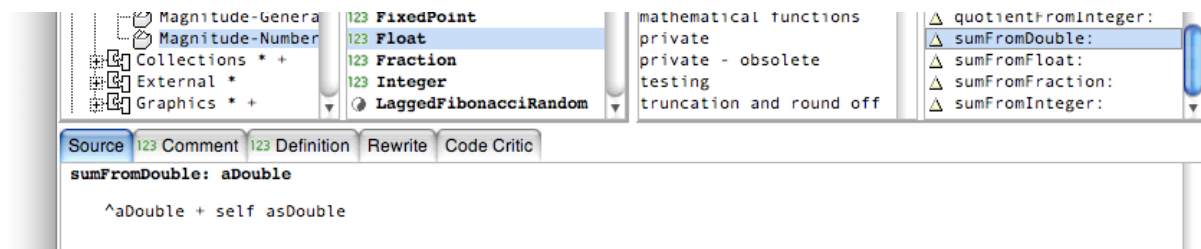
====

以下に精度を表現する generality というものを示す。これによって値の高い方に合わせて型変換して計算する、という仕組み、なんだけど。

あれれ？ これはいつ必要になるんだっけ？ さっきのパターンだと呼び返した時点で型合わせの対象がわかってしまうので、
と、思ったら、なんと、要らない。つまり上に書いた経路であれば、generality は殆どの場合不要である。

上に示したように Float が Integer に対して asFloat したりしている、これは Float は Integer を足すには Float 化すべきだ、と知っている (ハードコードされている) ことを意味する。よく調べてみると Float の sumFromDouble: は asDouble の位置が逆転している！
つまり二引数 (二項演算子) のどっちをどの型にすればいいか知ってるんだよ。

例を出す。Float クラスでは、sumFromInteger: の場合、asFloat が引数 (記述では前方) に対してあるのに、sumFromDouble: の場合、asDouble: が self (記述では後方) に対して掛かっている。



つまり generality なんても出てこないじゃない。
と思うと、なんとこうしたハードコードは (generality を使わないで演算させる) 加速のためらしい。

generality はこれらのハードコードに合わなくなった場合に機能するとのこと。

んで、以下、その generality の説明。

精度の高い型に合わせるのは generarity メッセージを使う。

```
100.10 generality.
```

```
80
```

```
3.3 generality.
```

```
80
```

```
100 generality.
```

```
20
```

これで値のより大きい方に型変換する。相手に合わせて型変換するのは coerce: メソッド。

```
6.7 coerce: 123
```

は、Float (6.7) に対して SmallInteger (123) を型変換する。

単に変換するだけで演算はしない。させようすると結局 6.7 + (6.7 coerce: 123) と書くしかない。

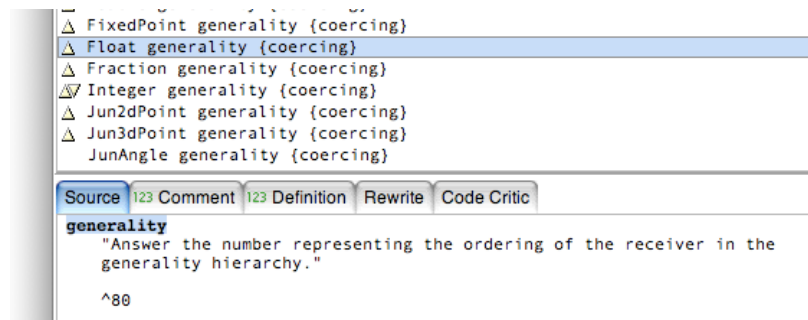
メモできなかったけれども、generality の相互の値によって、

```
aNumber1 generality aNumber2 generality [ ifGreater
    aNumber1 retry: aSelector with: aNumber2.
] else [
    aNumber2 retry: aSelector with: aNumber1.
]
```

のようなコードがあった。どっちをどっちに合わせるか（どっちにどっちを retry で投げつけるか）がそこで制御される。

以下は各型ごとに generality の値(80とか40とか)がハードコードされているのが分かる。

```
| messageSelector methodDefinitions |
messageSelector := #generality.
methodDefinitions := MethodCollector new allImplementorsOf: messageSelector.
Refactory.Browser.RefactoringBrowser
    openListBrowserOn: methodDefinitions
    label: 'Implementors of ', messageSelector printString
    initialSelection: messageSelector.
^methodDefinitions
```



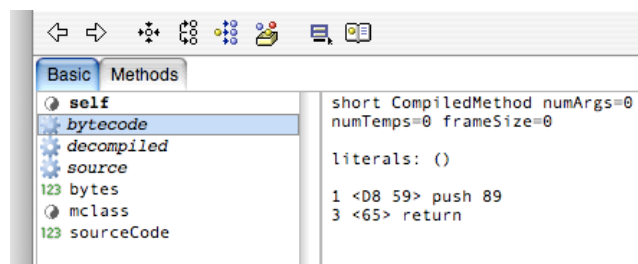
うへえ、80 モロちゃん。(^^!)

===

なんだか SmallDouble てな妙なものに話が突っ込んでいる。

SmallDouble compiledMethodAt: #generality.

でコンパイルした結果が得られる。以下。



うへえ。generality メソッドのコンパイルコードは単に 89 を返すだけなのですね。^80 (return 80) だもんねえ。

=====

ところで generality は高い方にばかり合わせるとは限らない。例えば 1/2 と 1/2 を足したら 1 になるが、これは Fraction と Fraction を加算したら SmallInteger になったパターンで、つまりこれは generality が小さくなっている。（演算の結果）
つまり generality を機械的に判定して演算すれば終わりというわけではない、かも、ね。
（演算の結果、常に generality を縮約できるかどうかチェックすればいいんだけど、それはいやだよねえ。そういう縮約は rational というメッセージで実装されているらしい。ration = 有理数、のような感じ。）

=====

同じ型どうしの演算は全部 primitive になっているらしい。ほんま？そんな事せんでもええんちゃうの？
まあバイトコードが余ったからスピードのためなんだろうねえ。

=====

2009/5/13

・私的変数 private variable p.75
小文字で始める。インスタンス変数とクラス変数と両方ともある。らしい。

私的変数はクラス（あるいはインスタンス）に付随して作られる。
つまりクラスを宣言したときに付随的に宣言する。
クラスは一般に aNameSpace に defineClass: #NameOfClass を投げて作る。（定義する、と言うのが正解？）
具体的には (p.76)
| aClass |
aClass := Smalltalk defineClass: #TransientBeing
 superclass: #{Core.Object}
 instanceVariableNames: 'firstName familyName birthday'
 classInstanceVariableNames: 'numberOfTransientBeings'

といった感じ。
・ Smalltalk というネームスペース、でやってみた。
・ ここでは TransientBeing という名前で作ってみた、というかシステム辞書に登録してね、と言ってみた、ということ。
 # つきなでのリテラルとして入る。
・ instanceVariableNames に文字列としてインスタンス変数名を入れている。これはいつシステム辞書に（ポインタ（参照）として）入るんだ？

これを、aClass inspect. して中身を見ると、そのインスタンス変数が見える、と思う、けどようわからん。
なおライフタイムはインスタンスの寿命と同じ。
（だってクラス（インスタンス）に付随して作られるからね。）

superclass の #{Core.Object} は束縛参照という。Core も一つの名前空間である。
Smalltalk という名前空間の中にあるらしい。p.78 に上下関係つきの図がある。一番上が Root。
束縛参照というのは実際には共有変数そのもの、らしい。共有変数の指定の方法で束縛参照を実現する、のだな。

・ 共有変数 p.77
名前空間の中で閉じて使える変数、らしいが、、、、というかこれは名前空間のアクセス手段そのものだ。
変数といっているが値の入れ物じゃない。（名前の入れ物だ）

例えば以前 Text というオブジェクト（クラス？）は Core ネームスペースの Text しかなかったが、
Text は今では XML ネームスペースにも存在する。
ただ Text と workspace で doit すると、システムは Core か XML かどっちの？と聞いてくる。

もう少しそれらしい例。
Text fromString: 'test'
を print it すると、やはり Core / XML を尋ねてきて、それを選択すると以下のように直したうえで結果が出る。
Core.Text fromString: 'test'
このドット表記はどうも Smalltalk らしくないが、まあしょうがない、、、、

この共有変数は p.81 にあるように、
Smalltalk defineSharedVariable: #TransientVariable

 initializer: 'nil'.

として定義する。これはネームスペースに対して defineClass とやるのと対等な立場になっており、つまり共有変数というのはクラスと同じ立場で（同じようなもので）存在させようとしている。
使い道としても、何かの大域変数（値の入れ物）として使うようなことはなく（やってもいいけど）、あるとしたらオブジェクトの名前をネームスペースを限定しながら指定する（同定する）ための手段（表記法）として使われる。#{ } の中で書く、つまりシステム辞書に対して指定するような状況にある。
（だから上の私的変数の supercalss #{Core.Object} のような記述で登場する。）

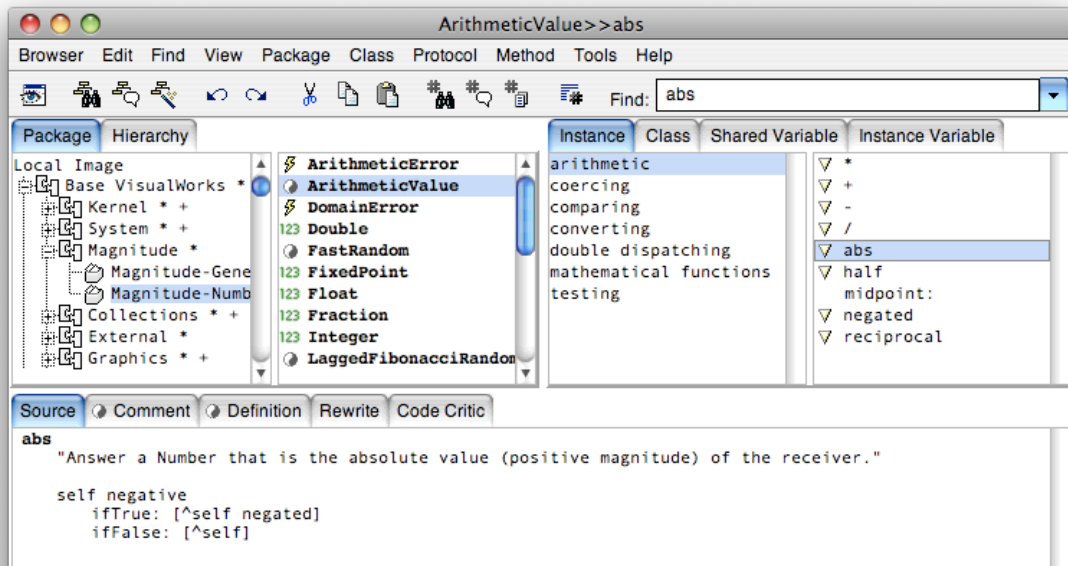
ただ、単に Core.Object と書いても良い。（さきほどの Core.Text の例を見よ）
#{Core.Object} と書くと、これをコンパイルしたときはなく実行したとき？に何か、、、、あれ？？まあいいや。パス。

ところでクラスも名前空間だから、上で述べたようにせずクラスに対して define することもできる。p.82
これは共有変数をクラスのスコープに閉じこめたもので、これはしかし値の入れ物のように使う。
つまりクラスからもインスタンスからもアクセスできる変数となる。
(私変数はインスタンスメソッドからしかアクセスできない。クラスメソッドからはできない。)

・特殊変数 (シュード変数)

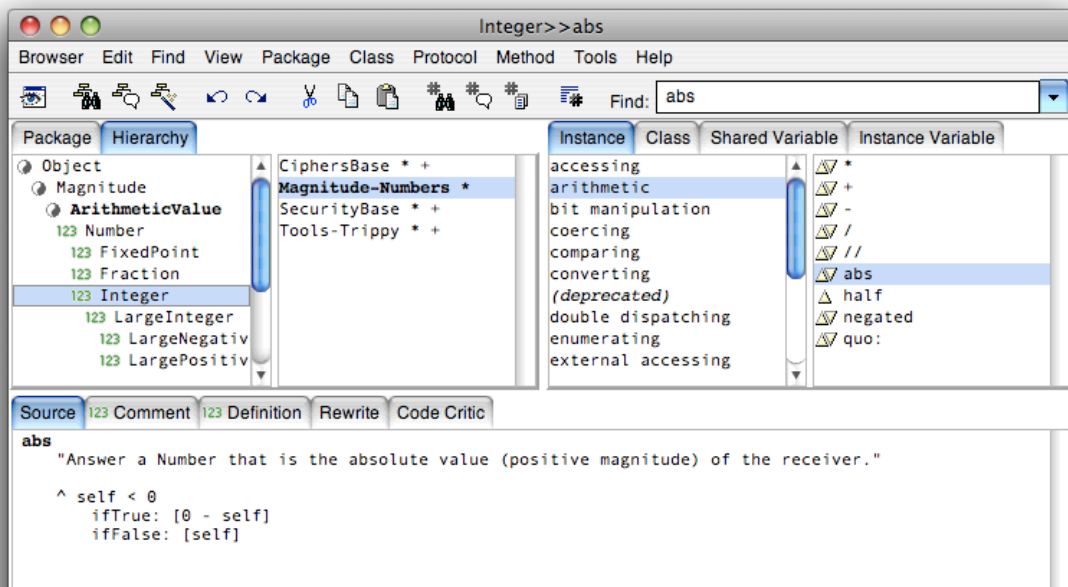
変数の名前が予め決まっており、後から変えられない。self, super などがある。

3 abs の例：



を見ると、自分自身に negative を投げて、if で分岐。

なおこの状態で、パッケージでなく Hierarchy タブを選択して、他のクラスでの abs の実装を見て回ることができる。



super
はまあいいとして。

thisContext

は多分僕には関係ない pseudo 変数だと思う。。。システムプログラミングする人に必須かもしれないが。
このオブジェクトをどうやって呼び出したかを調べることができる特殊変数。トレーサー用？
(誰かがオブジェクト (自分) にメッセージを送ってきたとしたら、それを誰が送ってきたかを知ることが出来る。)

・メッセージ (ようやくだ！)

workspace で printIt やってるときは、sender は workspace になる。つまり nil だって。(workspace の value は nil だけ?)
普通は (オブジェクトの) プログラムの中にコードが書かれているので、その場合はそのオブジェクトが sender である。

sender についての意味、典型的な例を挙げることが出来なかったのが面白い。
1 + 2 については、つまり 1 がレシーバで + がメッセージセクタ、2 が引数となる。
しかしその文脈ではどこにもセクターが登場しない。
workspace で実行するときは workspace そのもので、その実体は nil だ。
だから nil が sender になる、というが、そりゃおかしい。
例えばどこかのインスタンスのメソッドに 1 + 2 と書かれた行があれば、そのインスタンスが sender となる。

===== 6/3

p.88 メッセージから

・単項メッセージ

メッセージとは実際にはメッセージセクタ+引数のことを指す。
ということはこの単項メッセージは結局メッセージセクタそのものであるはずだ。
これ、一万とか何万とあるそうです。

3 asPoint

3 @ 3

この表示はかなりおかしい。x, y に (3, 3) が入ったオブジェクトとなるはずなのだが、それを直接的に表示せず、評価すればそのようになるはずの integer の 3 に対して @ を投げるようなことをしている。結果を結果として出さず、違う言い方に言い換えているようだ。
その理由は何だ? というような質問をしたら、青木さんから asPoint はそのように printString をやっているだけだ、とのことであった。

3 asPoint printString

'3 @ 3'

これを内部表現に即して出そうとすると、たとえば storeString などがあるらしい。

3 asPoint storeString

'(Core.Point x: 3 y: 3)'

ああ、これこれ。これがまあ妥当な表現なわけだね。

それにしてもメモリの中にそのように格納されている、ということを直接的に表現すると言う意味ではこちらの方がより近いなあ。

・二項メッセージ

+, @ などがそれにあたらしい。数十種しかないそうで、ある意味特殊らしい。

なお何かを実装する際に引数が一つだったら二項メッセージでもキーワード・メッセージでもどちらでも実装できる。
しかしほぼ間違いなくキーワード・メッセージでやる。二項メッセージは二文字までしか対応させられないらしく、それについてはやはり作るにしても数が限られてしまう。だから汎用的なものでじっくり作らないといけない、という慣習上の問題らしい。
それ以外の何もシステム上の制約はないらしい。

・キーワードメッセージ

引数は 255 まで (バイトコードに収まる数ということかな?)。

引数が 4 つまでだとそのままブッシュ、それ以上は array を構成して渡すらしいのでちょっと遅いらしい。

Point x:10 y: 20

てな感じで書くのだが、このとき「x: 10 y: 20」がメッセージ、となる。このときメッセージセクタはなんと x: y: ではなく xy: になる。

実際には #x:y: と思えばよい。ははは。インタプリタ向けだ! 中が透けると。

#(10 20 30 40) at: 3

30

#(10 20 30 40 50 60) copyFrom: 2 to: 4

#(20 30 40)

というような感じ。なおこれは

#(10 20 30 40 50 60)

perform: #copyFrom:to:

with: 2

with: 4

として実行される。

#(10 20 30 40 50 60)

perform: #copyFrom:to:

withArguments: #(2 4)

でもいい。

なお with: は上のケースでは三つまでしかない (つまり合計四つまで) 。

それ以上ある場合は withArguments で array にして渡せ、と言っている。

・演算順序

3 + 4 * 5 は「左からやる」のでうまくいかない。メッセージはすべて左から順に評価するのだ。
(sugaring は嫌いだったんだな Kay は!)

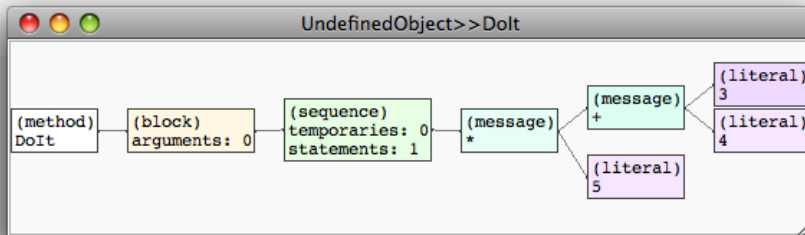
いやなら $3 + (4 * 5)$ とやりなさい、と。

・デバッガ

このあたりのことを調べたければ Do it しないで Debug it すればいい。とは、言うが、これ難しい。まあ当たり前だ。どこまで step in して、どこまでで step over すればいいのかさっぱりわからん。

・処理順序を構文木で見る

(JunParseTree code: '3 + 4 * 5') show
とすると、以下のような表示が行われる。



まあこういう構文木になるわけえ。

===== p.468 へとジャンプ！！！！

JunSystemUtility printInheritancesOf: MessageSend

とやると、クラス MessageSend へのインヘリタンスが出てくる。

```
Object
| Message
|| MessageSend
||| LinkMessage
```

JunSystemUtility printInheritancesOf: Integer
てなことをやってみるともう少し分かる？

Message allInstVarNames

OrderedCollection ('selector' 'args') なんだこの args は ! arguments と書け、と青木さん曰く。

MessageSend allInstVarNames

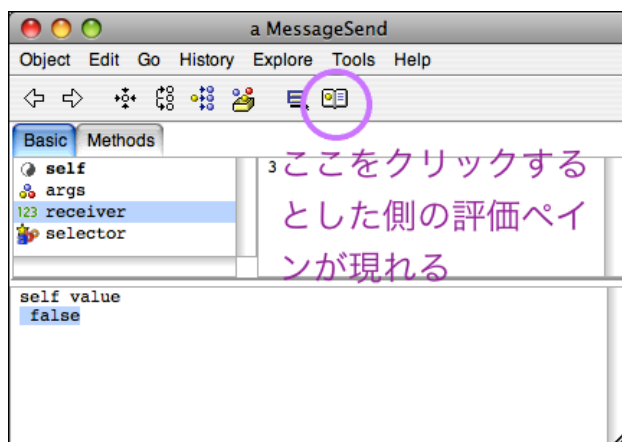
OrderedCollection ('selector' 'args' 'receiver')

以下はレシーバが 3、セクタが even: のメッセージ送信そのものをインスタンスにせよ、というように言った例。

MessageSend receiver: 3 selector: #even

a MessageSend with receiver: 3, selector: #even and arguments: #()

これを printit しないで Inspect it してみると、内部構造が見える。その中で（一番下のペイン）で self value してやると、false（3 は even じゃないよ）となる。

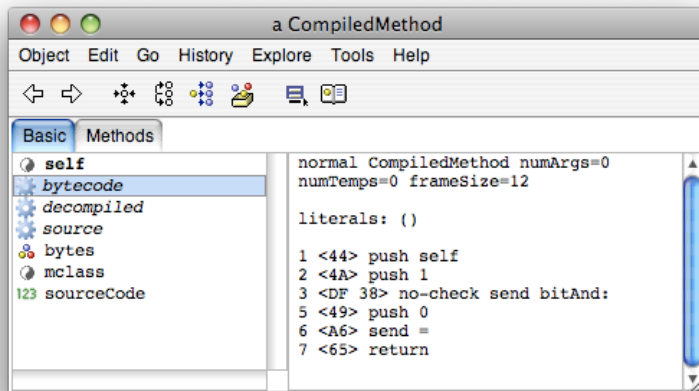


となる。

以下は、3 が even に答えられたことを前提に、その even がコンパイルメソッドに対してどうなるかを調べる。

3 class compiledMethodAt: #even

を、inspect it すると、これで 3 のクラス、つまり smallInteger クラスに対して、compiledMethodAt: が even に対して割り当てられたメソッド（ただしバイトコード）が出てくる。



3 class compiledMethodAt: #to:by:

などとしても実際には何も出てこない。つまりこれはこのクラス（smallInteger）には実装されていなかった。

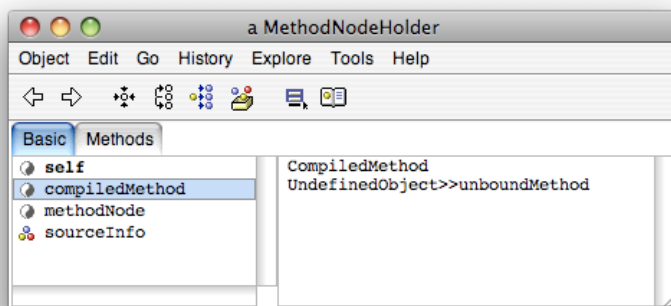
3 class superclass superclass compiledMethodAt: #to:by:

などとして上までたぐっていくとまあ出てきたり。

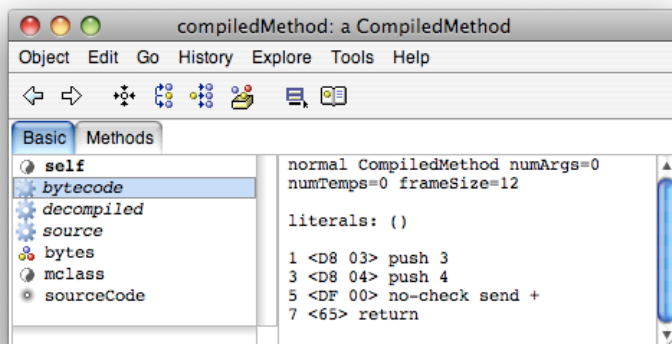
p.474 の Compiler new あたりが面白い。

Compiler new parseNoPattern: '3 + 4' in: nil class notifying: nil.

とかやって、コンパイラにコンパイルさせて、それを Inspect it すると、



となるが、この compiledMethod 部分をダブルクリックすると、



というように見える。

ところで Compiler new parseNoPattern: '3 + 4' in: nil class notifying: nil. はかなり面白い表記で、in: nil class が一つのものになっている点に注意。つまり nil class で nil が存在しているクラス、を、in: の引数として渡しているのである。(@_@!)

===== p.94 へパ-----っく！

・文末（ピリオド）

は、まああってもなくてもいいんだけど、文と文のセパレータとしては必要になるので、そこ注意。

プログラムの最後にどのように終わるのか（どのように戻り値を返すのか）で、`yourself` が出てきて話がちょっとおかしくなった。（つまり戻り値と `yourself` の意味の違いにひっかかってしまった）

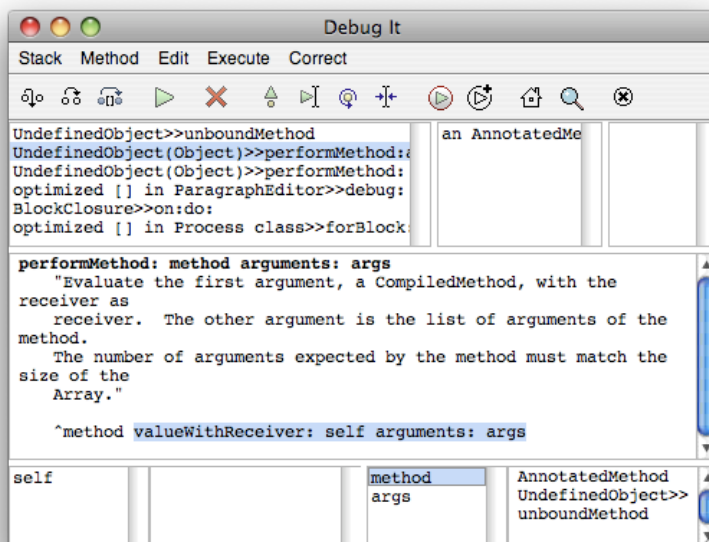
`aBody` で終わると、
`aBody yourself` と、
`^aBody` とはバイトコードが異なる。
ちょっと後で較べてみよう。

Compiler new parseNoPattern: '[a] a := 3 + 4. ' in: nil class notifying: nil.
などとしてバイトコードを見る。

あるいは、単に

```
| anInteger |  
anInteger := 3 + 4.
```

などとして Debug it し、以下の Debugger ウィンドウで少し上のレベル（デバッガでの表示並びでは下方向）にいて、`method` をダブルクリックし、



湧いて来る下のウィンドウの bytecode を見てみる、でもいい。

最初に試すパターン、

```
| anInteger |  
anInteger := 3 + 4.
```

（つまり戻り値などについて何も言わない）だと、bytecode は、

```
1 <D8 03> push 3  
3 <D8 04> push 4  
5 <DF 00> no-check send +  
7 <4C> store local 0; pop  
8 <10> push local 0  
9 <65> return
```

ただし、decompiled を見ると、

```
| t1 |  
^t1 := 3 + 4
```

になっていて、勝手に `^` がついている。（下の方の★に書くけど、これは無駄をしている）

次に

```
| anInteger |  
anInteger := 3 + 4.  
anInteger.
```

（つまり戻り値として、というより単に何かを置いてみた、だけ）なら、

```
1 <D8 03> push 3  
3 <D8 04> push 4  
5 <DF 00> no-check send +  
7 <4C> store local 0; pop  
8 <10> push local 0
```


9 <65> return

の、

```
| t1 |  
^t1 := 3 + 4
```

で、なんとまったく同じ。

(これは自動的に推定された return すべきものがたまたま明示的に置いたものと同一だけだった、という話でもあるが)

次。

```
| anInteger |  
anInteger := 3 + 4.  
anInteger yourself.
```

(つまり yourself と行って、お前自身を置くんぞと明記してみた) なら、

```
1 <D8 03> push 3  
3 <D8 04> push 4  
5 <DF 00> no-check send +  
7 <4C> store local 0; pop  
8 <10> push local 0  
9 <F0 40> send yourself  
11 <65> return
```

で、

```
| t1 |  
^(t1 := 3 + 4) yourself
```

となっていて、ともに yourself をメッセージ送信している。なおこの yourself の実体は Object クラスにある、らしい。

さて ^ で戻してみるとどうなるか。

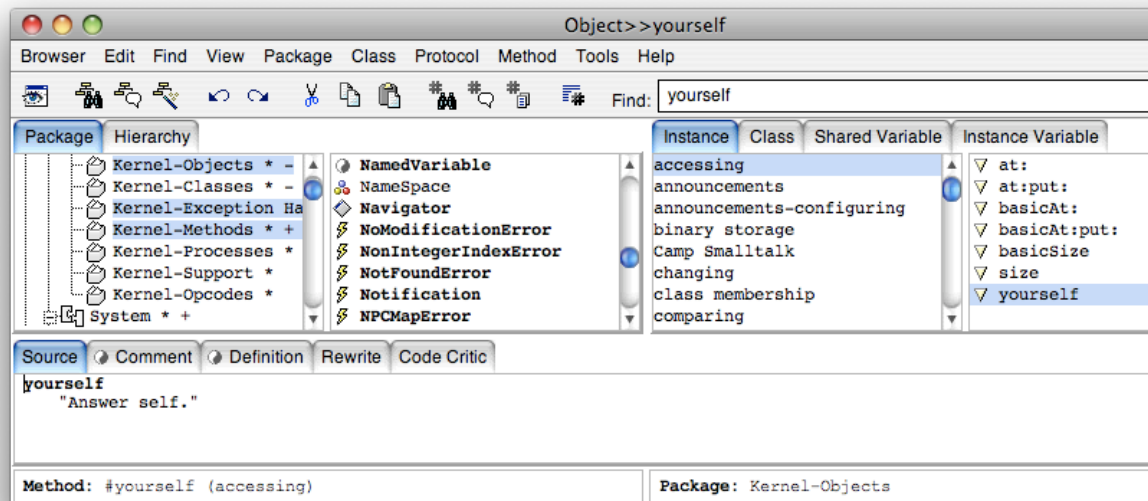
```
| anInteger |  
anInteger := 3 + 4.  
^anInteger.  
1 <D8 03> push 3  
3 <D8 04> push 4  
5 <DF 00> no-check send +  
7 <4C> store local 0; pop  
8 <10> push local 0  
9 <65> return
```

予想どおり、普通にやる。

が、一連のコードを見るときかなり面白い。つまり ^ で戻すときは、それまで残っていたであろう演算結果 (local 0) を、わざわざスタック最上部を取り除いて、そこに push して return している。(無駄ちゃん! 下の★★参照)

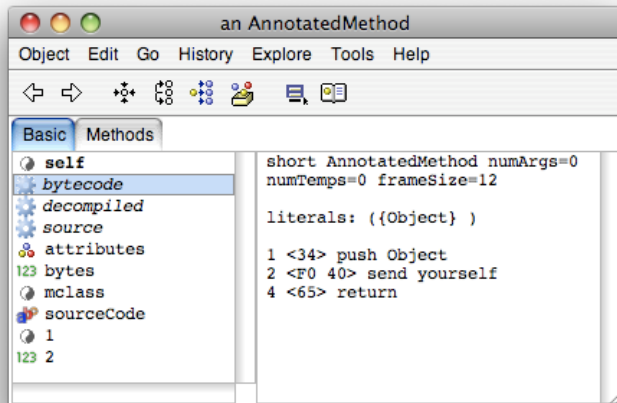
ところで以下が Core.Object の yourself method 実装。

濱崎さんが教えてくれたとおりに、「何もしていない」。そう、return すらししていない。(コメントでは ^self せよと書いてあるけどコードはない!)



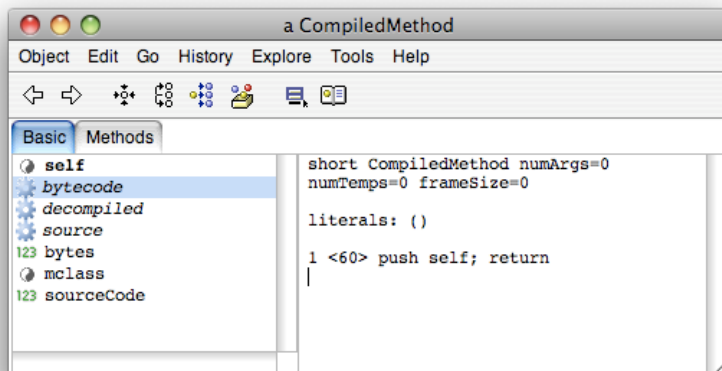
つまりこれは yourself が呼ばれたときには最初に自分自身が push されているであろうから、それを return するよう仕向けているのだ。
^self と書くのも嫌だ、ということか。(最適化されない環境では一旦 pop されて、local 0 か何か自分自身を再 push するのが嫌なのか)

ために Object yourself とだけ書いて Debug it してバイトコードを見ると、あれあれ、



うーん、実際に動くところは分からないなあ。（ところで yourself がバイトコード上ではラベル40とハードコードされてるよ！（^_!））

いやいやそこで止まることはなし。先の compiledMethodAt: があるじゃないか。
Object compiledMethodAt: #yourself.
とやって、Inspect it してみると、



である。つまり push self と return は実はバイトコード上では 1 バイトの命令として存在していた。ははは。
なお decompiled では
yourself
^self
となっている。ソースは勿論
yourself
"Answer self."
なのだけれどね！

ただ、上の例は、戻り値の返し方、あるいは yourself の用法としてはかなり悪い。
恐らく次のカスケードを用いた例が何か作れると思う（のでやってみるか）。

・カスケード（セミコロン）

Transcript などで顕著だけれど、対象となるオブジェクトが同一である、ということを略記するだけのもの。
なおカスケードしないで書くと、それだけ正直に push pop を繰り返してしまうので（非常に意味なく盲目に！！）これを省略する（そりゃー pop してまたすぐ push するのは意味が無いじゃないの！）
でもまあ最適化の対象なので、最近はちゃんとバイトコード最適化するみたいよ。

例えば以下の二行については、前者は値が（つまり Inspect It した結果が）110 なのに対して、後者は OrderedCollection なインスタンス (100 110) である。

OrderedCollection new add: 100; add: 110.

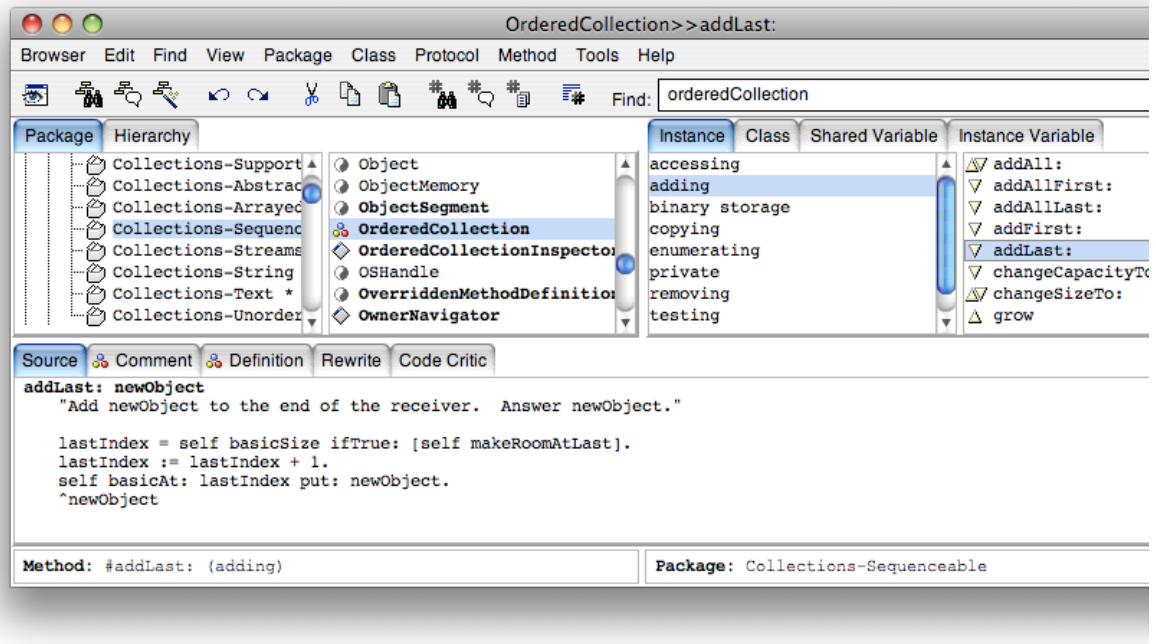
OrderedCollection new add: 100; add: 110; yourself.

つまり最後にやった演算である add: 110 の結果は 110 なのだ。

これはまあ仕様上の問題なんだけれど、これを yourself で回避することが出来る。

ちゃんと書くと、前者は
| aCollection |

```
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
であり、その結果はもちろん ^(self add: 110) になり、これは 110 である。
つまり OrderedCollection の add: は最後に self ではなく引数である 110 を ^ する。
ほんまかいなとシステムブラウザで見てみると、
```



というわけで、みごとに ^newObject になっている。(^self ではなく)
ところが後者は以下のように記述されたことを意味する。

```
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
aCollection yourself.
```

これは Core.Object にある yourself によって解釈され、つまり ^self となる。だからまあ言うてしまうと

```
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
^aCollection.
```

と書いたときと変わらん。が、こういう一時変数を用意しないとそれができない場合でも、カスケードと yourself を利用すれば略記が可能ということかな。

(なお ^aCollection と書かず単に aCollection と書いたときも、内部的にはただ ^self とされるので、同じ結果が得られる。)

以下ちょっとバイトコードを見て確認。

まず何も書かないケース。このプログラム全体の値は 110 になることに注意。(aCollection が戻るわけではない)

```
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
```

バイトコードはこれ。

```
1 <34> push OrderedCollection
2 <BC> send new
3 <4C> store local 0; pop
4 <10> push local 0
5 <D8 64> push 100
7 <B8> send add:
8 <56> pop; push local 0
9 <D8 6E> push 110
11 <B8> send add:
12 <65> return
```

逆コンパイルはこちら。

```
| t1 |
(t1 := OrderedCollection new) add: 100.
```

```
    ^t1 add: 110
```

となる。

次、aCollection を明に積む、というかこうしないとこのプログラムは 100, 110 を詰めた配列が戻らない。

```
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
aCollection.
```

バイトコード、

```
1 <34> push OrderedCollection
2 <BC> send new
3 <4C> store local 0; pop
4 <10> push local 0
5 <D8 64> push 100
7 <B8> send add:
8 <56> pop; push local 0
9 <D8 6E> push 110
11 <B8> send add:
12 <56> pop; push local 0
13 <65> return
```

で、

```
| t1 |
(t1 := OrderedCollection new) add: 100.
t1 add: 110.
^t1
```

となる。ここから分かるように、aCollection と書いても、^aCollection と書いても結果は同じなのだ。

(実際やってみても同じ)

で、以下のように今度は yourself を書く。

```
| aCollection |
aCollection := OrderedCollection new.
aCollection add: 100.
aCollection add: 110.
aCollection yourself.
```

バイトコード、

```
1 <34> push OrderedCollection
2 <BC> send new
3 <4C> store local 0; pop
4 <10> push local 0
5 <D8 64> push 100
7 <B8> send add:
8 <56> pop; push local 0
9 <D8 6E> push 110
11 <B8> send add:
12 <56> pop; push local 0
13 <F0 40> send yourself
15 <65> return
```

となり、

```
| t1 |
(t1 := OrderedCollection new) add: 100.
t1 add: 110.
^t1 yourself
```

と、はあ、まあそうか。(^ で yourself を送った結果を返す、か)

というわけで、この例では「何も書かない(自然に最後の結果だけが返る)」「明にスタックに置いて返るように書く(=^ がついたのと同じことをする)」「yourself をつけて置いて返るように書く」のがそれぞれ異なる挙動となるものだった。

が、それにしてもこの例の yourself はほとんど意味がないことがわかる。実際には send yourself は return self になって戻るだけだから、単に ^t1 と書いた方が余程速い。(VM はこれを最適化の対象と見なしたいのではないかなと思えるほどだ)

逆に何も書かない場合のバイトコードはちょっと無駄をしているように見える(上に出した ★★ のところ)

```
| anInteger |
anInteger := 3 + 4.
(つまり戻り値などについて何も言わない) だと、bytecode は、
1 <D8 03> push 3
3 <D8 04> push 4
5 <DF 00> no-check send +
7 <4C> store local 0; pop
8 <10> push local 0
9 <65> return
```

となるが、このバイトコード、おそらく最後の 7, 8, 9 行目は、単に

```
7 <65> return
```

で良いはず。store local 0 が何か重要なら（どうか？）

```
7 <??> store local 0;
```

```
8 <65> return
```

でええんちゃうかな。（store local 0 が重要というなら、逆に store local 0; return などというバイトコードがあっても不思議でないけど）

つまりこの「最後に残っているものを自然に出す」だけのコードなのに、わざわざ pop して再 push するのは損だわね。何でやる。

（以上★★終わり）

さて戻って、yourself と表記することに便利ことがあるとしたら何か、という例の中身をみる。

```
(OrderedCollection new) add: 100; add: 110
```

```
1 <34> push OrderedCollection
```

```
2 <BC> send new
```

```
3 <68> dup first
```

```
4 <D8 64> push 100
```

```
6 <B8> send add:
```

```
7 <6A> dup last
```

```
8 <D8 6E> push 110
```

```
10 <B8> send add:
```

```
11 <65> return
```

と、

```
(OrderedCollection new) add: 100; add: 110; yourself
```

```
1 <34> push OrderedCollection
```

```
2 <BC> send new
```

```
3 <68> dup first
```

```
4 <D8 64> push 100
```

```
6 <B8> send add:
```

```
7 <69> dup
```

```
8 <D8 6E> push 110
```

```
10 <B8> send add:
```

```
11 <6A> dup last
```

```
12 <F0 40> send yourself
```

```
14 <65> return
```

で、両者結果が違う。（いや、結果となる値も本当に違うし、バイトコードも当然違う）

もしこれを一時変数を用いてやると、

```
| aCollection |
```

```
aCollection := OrderedCollection new.
```

```
aCollection add: 100.
```

```
aCollection add: 110.
```

```
aCollection.
```

バイトコード、

```
1 <34> push OrderedCollection
```

```
2 <BC> send new
```

```
3 <4C> store local 0; pop
```

```
4 <10> push local 0
```

```
5 <D8 64> push 100
```

```
7 <B8> send add:
```

```
8 <56> pop; push local 0
```

```
9 <D8 6E> push 110
```

```
11 <B8> send add:
```

```
12 <56> pop; push local 0
```

```
13 <65> return
```

で、まあ一行ですらりと書けないことと、あと、pop しては push local 0 の繰り返しで dup よりは効率悪いねえ、と。

=====

しかし dup fist, dup, dup last などの意味が分からんようになった（頭の中ですっきりスタックが追えない）。

などを書いて、カスケードがどのように振る舞うか、実際のスタックの動きを想像していると、はて、分からんようになった。

想像では普通にこれ（dup）をやるとスタックに積み残しが出る。

```
(OrderedCollection new) add: 100; add: 110
```

```
1 <34> push OrderedCollection
```

```
2 <BC> send new
```

```
3 <68> dup first
```

```
4 <D8 64> push 100
```

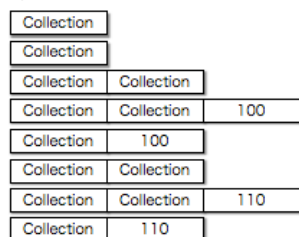
```
6 <B8> send add:
```

```
7 <6A> dup last
```

```
8 <D8 6E> push 110
```

```
10 <B8> send add:
```

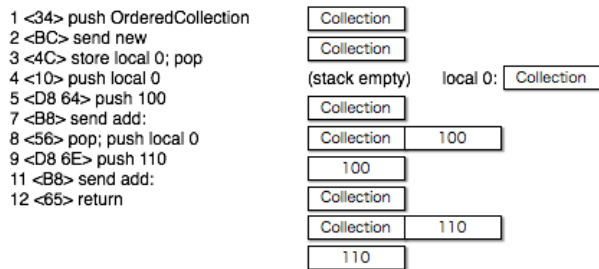
```
11 <65> return
```



あれ？最後に積み残しが出たぞ。dup last の意味が分からん。そもそも 100 を捨てる動作がどこにもない（上の図ではつじつま合わせのために捨てている）

(戻り値は本来 110 になるので最後は合うが、その前に残っているはずの OrderedCollection インスタンスが余計なものとして残る)
以下のように一時変数を使うように書いたプログラムはきれいさっぱりスタックからものが消えるように見えるが。

```
l aCollection l  
aCollection := OrderedCollection new.  
aCollection add: 100.  
aCollection add: 110.
```



なお、

OrderedCollection new add: 100; add: 110; add: 120
の場合は、dup first, dup, dup last が登場する。

```
1 <34> push OrderedCollection  
2 <BC> send new  
3 <68> dup first  
4 <D8 64> push 100  
6 <B8> send add:  
7 <69> dup  
8 <D8 6E> push 110  
10 <B8> send add:  
11 <6A> dup last  
12 <D8 78> push 120  
14 <B8> send add:  
15 <65> return
```

ので、バイトコード一覧を捜す。

<http://www.cincomsmalltalk.com/pdf/TechNotes/vwBytecodes.pdf>

に仕様あり。

OpDupFirst 16r68

Duplicate the top value on the stack, in preparation to send the first message of a cascade.

The duplicated value is the receiver of the cascaded messages.

スタックトップを複製するカスケードの最初の準備でやる、と書かれている。へえ。もっぱらソレ用みたいな感じだな。

複製されるのはカスケードされるメッセージのレシーバだと。

OpDupNext 16r69

Pop the top value off the stack, then duplicate the new top value of the stack.

This is used in preparing to send a message that is part of a cascade, when the message is neither the first nor the last message in the cascade.

In such a case, the value which is to be popped is the return value of the preceding message of the cascade, and the value being duplicated is the receiver of the cascade.

スタックトップをポップして（捨てて！）、新たにスタックトップとなった値を複製、ですと！

やはりカスケードの、それも先頭でも途中でもない時に使われるとある。やっぱ専用か。

このような場合、ポップされたvalueは先のカスケードされたメッセージの結果であるはずで、かつ複製されたvalueは、カスケードのレシーバであるはず、と。もうまったく決めうちというか専用ね。

OpNoDup 16r6A

Pop the top value off the stack.

The popped value is the return value of the previous message send, removed in preparation to send the last message of a cascade.

（この命令 6A はデバッグなどの bytecode 表示では dup last と表示される。そっちのほうが分かりやすいか？）

スタックの値をポップする。（あれ？単なる pop 命令では無いのか？？そういえば見たことがない。）

ポップされた value は直前に送られたメッセージの戻り値であるはずだが、カスケード中の最後のメッセージ送信の準備のために取り除くのだ、と。

あっはっは。字面にダマされたけど、なんのことはない、カスケード専用命令、それもセットでそのように動作するように揃えられてるよ。
これで疑問（上の図でのスタック積み残し）は消えた。

=====脱線終わり。一応納得。

第9回 (2009.7.1) は欠席したので自習。

Block Closure。

・ブロッククロージャはインスタンスなのである、ということ？

```
| aBlock |  
aBlock := [3 + 4].  
^aBlock
```

を print it すると、BlockClosure [] in UndefinedObject>>unboundMethod などとエラーがでる。

デバッグで見るとどうも [3 + 4]のところで出るっぽい。でも最後を

^aBlock value

とすると 7 と値が出る。こりゃいったい何じゃ。教科書 p.99 に value の例があるが、これとはちょっと話が合わない？

そもそも ^ は自分自身に対して value するということじゃないんかいな。違うか、p.98 最後の例は [] 自体を return させる例だけけど、それはエラーになることを示している。これを「valueメッセージを受信するまで実行が猶予されていると考えることができる」とあるが、うーんそれならそれっぽいエラーメッセージを吐けば良いではないか？（というかなぜエラーとする？そのまま [] ごと返せば良いではないか？）

・ value メッセージによる評価

[3 + 4]

は BlockClosure [] in UndefinedObject>>unboundMethod によるエラーとなるが、

[3 + 4] value

は 7 となる。さて、

[a | 3 + a] value: 4

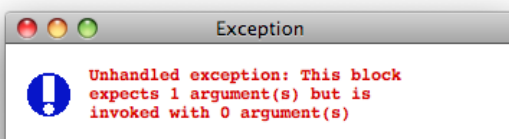
は 7 となる。この value: の後に続く値はブロッククロージャの一時変数（っぽいもの）に引き渡される。

:a がそれ。コロンがこんなところで変数の前に出てくるとは。しかし値の引き渡しの結び付けがない。順番？

ところで、

[a | 3 + a] value

をやると以下のようなエラーとなる。



引数が一つあるはずなのに用意されずに呼び出されたよ！

では、と、

[3 + a] value: 4

などに変なことをやっても無駄。（a という変数がないよ、とかそういうところでまず止まる。）

ともあれ素直に、

[x :y | x * 2 + y] value: 3 value: 4

とやると結果は10 で、やはり順番だ。いまひとつ奇妙だがまあしょうがないか。本当なら

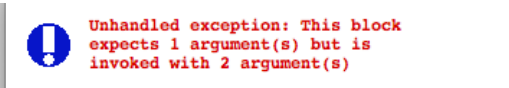
[x :y | x * 2 + y] x: 3 y: 4

と書きたいところだけど、、、

もう一つ面白い例。

[x | x * 2] value: 3 value: 4

つまりvalue: による引き渡し数の方が多いと？



引数が二つあるはずなのに用意されずに呼び出されたよ！

面白い。で、青木さんの説明では次は「間引き実行」として cull: が紹介されている。

どうやらこれは value: に変わるもので、過剰に与えた場合前方のみ使って後はほうっておいてくれるらしい。

[x :y | x * 2 + y] cull: 3 cull: 4

結果は正しく 10 になる。次のものだ と value: ではエラーになったが、

[x | x * 2] cull: 3 cull: 4

cull: では正しく 6 となる。過剰に与えた場合はそれでいいとして、不足があった場合は？というと、やはり直上と同じく「足りない」とコンパイラに怒られる。まあしょうがないか。

・ ブロッククロージャでの複文と return

```
| redBlock greenBlock blueBlock mainBlock |
redBlock := [Transcript show: 'red'; cr].
greenBlock := [Transcript show: 'green'; cr].
blueBlock := [Transcript show: 'blue'; cr].
mainBlock :=
  [Transcript clear.
   redBlock value.
   greenBlock value.
   blueBlock value].
mainBlock value
```

てなこともできる。まあオブジェクトなので。。。で、途中脱出したい場合は普通に return すればよい。

以下、適当に greenBlock で ^nil して実行を途中で止めてみたりして。

```
| redBlock greenBlock blueBlock mainBlock |
redBlock := [Transcript show: 'red'; cr].
greenBlock := [Transcript show: 'green'; cr. ^nil].
blueBlock := [Transcript show: 'blue'; cr].
mainBlock :=
```

```
[Transcript clear.  
redBlock value.  
greenBlock value.  
blueBlock value].  
mainBlock value
```

・制御文 (if文)

むかしは ifNil はなかったらしい。最近 VisualWorks に入り出した。Squeak も取り込んだらしい。
コレがなかった当時は (Boolean...) isNil ifTrue: [.....] とかいった感じで書いたんだろうなあ。
ところで VisualWorks では Control-T で ifTrue: と入力ショートカットされる。Control-F は ifFalse: うひょー。
Control-G は := (get) だ。でも他に面白いものはなかった。てことはそんなによく使うのか ifTrue: は !

```
| anObject |  
Transcript clear.  
(anObject := nil)  
ifNil: [Transcript show: anObject printString].
```

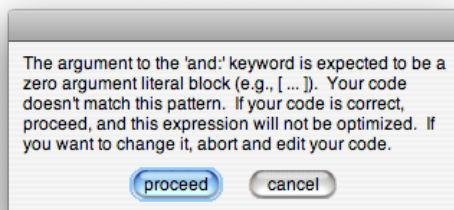
てな感じ。条件節を用意して、それに対してメッセージ ifNil: などを投げる。このときブロックロージャには引数として一つだけ、条件判定値が渡されるようで、例えば

```
| anObject |  
Transcript clear.  
(anObject := 123)  
ifNil: [Transcript nextPutAll: anObject printString]  
ifNotNil: [:it | Transcript nextPutAll: it printString].  
Transcript flush
```

のようにしてそれを扱える。[:it | (ここで it を使う)] という感じ。

ところで Smalltalk には論理和 (and) の構文(?)は無い。ifTrue: と同じものとして扱わせる。

```
( 1 = 1 )  
は true を返すので、  
( 1 = 1 ) and: ( 2 = 2 )  
と書けるのかと思うと、書けないことはないが以下のエラーダイアログが出る。
```



[..]みたいなブロックが来るのが普通ちゃう？という警告

なおこれでも proceed すると動作する。つまり一般的な記法は以下になる。

```
( 1 = 1 ) and: [ 2 = 2 ]  
これが気持ち悪ければ  
( 1 = 1 ) and: [ ( 2 = 2 ) ]  
とすれば良いが、それにしても妙だと思ったらなんのことはない、これは ifTrue: と同じだ。  
( 1 = 1 ) ifTrue: [ ( 2 = 2 ) ]  
まあ確かに and はそういうもんだわなあ。同じ理屈で or: は ifFalse: そのものである。  
( 1 = 1 ) or: [ 2 = 2 ]  
( 1 = 1 ) ifFalse: [ ( 2 = 2 ) ]
```

・繰り返し

```
Transcript clear.  
5 timesRepeat:  
[Transcript show: 'test'; cr].  
Transcript flush
```

など。anInteger timesRepeat: であることに注意。元の定義を見ると、

```
timesRepeat: aBlock  
| count |  
count := 1.  
[count <= self  
whileTrue:  
[aBlock value.  
count := count + 1].  
^nil
```


となっている。whileTrue 自身もブロックを使っているが、これ、どこかでループせずに全体をうまく構築したのは偉いなあ。（後述★）
ところで恐るべきコトに timesRepeat の戻り値は常に nil である。（上記のごとくハードコードされている）

ところでところで Smalltalk イディオムには「Browser browseAllImplementorsOf: #timesRepeat:」のようなものがある、とある。
<http://www.cc.kyoto-su.ac.jp/~atsushi/Documents/SmalltalkIdioms/chapter4/index-j.html>
が、VisualWorks の場合は下記のようなになる。これは便利だなあ。（メソッドの定義をシステム中から探し出す）
MethodCollector new browseAllImplementorsOf: #timesRepeat:

次、インターバル（間隔、なのだが、まあ数列、あるいは collection だと思えばよい。青木本は「間隔オブジェクト」となっている）。

```
Transcript clear.  
(1 to: 10 by: 2) do:  
    [each |  
     Transcript  
       nextPutAll: each printString;  
       cr].  
Transcript flush
```

こちらの to: by: あたりの定義は Number クラスにあって内容は以下の通り。

```
to: stop by: step do: aBlock  
    (Interval from: self to: stop by: step) do: aBlock  
Interval クラスの from:to:by: を見ると、  
from: startInteger to: stopInteger by: stepInteger  
"Answer a new instance of Interval, starting at startInteger, ending and  
  stopInteger, and with an interval increment of stepInteger."  
^self basicNew  
  setFrom: startInteger  
  to: stopInteger  
  by: stepInteger
```

となっている。basicNew って何だ？コメントを見るとこれでインスタンス生成するらしいが、

setFrom:to:by: は Interval クラスの private method になっており、そこには

```
setFrom: startInteger to: stopInteger by: stepInteger  
    "Initialize the instance variables."  
    start := startInteger.  
    stop := stopInteger.  
    step := stepInteger
```

とある。インスタンス変数の設定をするだけだ。これに対して do: [...] が掛かる格好なので、Interval クラスの do: の定義を見ると、

```
do: aBlock  
    "Evaluate aBlock with each of the receiver's elements as the argument."  
    "We avoid accumulating a delta to minimize round-off error."  
    | n end |  
    "We don't use Number>>to:do: here because that would create a circularity."  
    n := 0.  
    end := self size - 1.  
    [n <= end]  
        whileTrue:  
            [aBlock value: start + (step * n).  
             n := n + 1]
```

となっている。aBlock に対して value: でカウンタ変数が与えられているのがわかる。

この do: はいろいろ使えるようなので、例えば

```
anInterval do: [each | "繰り返し処理"]
```

だけでなく

```
aCollection do: [each | "繰り返し処理"]
```

などでも使われている。

```
aCollection do: [each | "繰り返し処理"] separatedBy: ["繰り返しの間の処理"]
```

というものもある。

```
#{111 222 333 444 555}
```

```
do: [each | Transcript nextPutAll: each printString]
```

```
separatedBy: [Transcript show: '-'].
```

とかいった感じ。（出力は 111-222-333-444-555 になる）

面白いなあ。これの面白いところは「間の処理」に限定されているので、このプログラム例では最初や最後に余計な - が付かないようになっているが、
そのために特に何も配慮しなくていいところが良い、というものらしい。（最初の改行をなくしたいとか、最後の空白を外したいとか、そういうこ
と。）

単純繰り返し while はこういう感じになる。

```
| anInteger |  
anInteger := 1.  
[anInteger < 10] whileTrue: [
```

```

Transcript show: anInteger printString; cr.
anInteger := anInteger + 1
]

```

で、この whileTrue: を追いかけてみる。（先述★）

定義は Kernel>>BlockClosure のインスタンスメソッドとしてあり、内容は

```
whileTrue: aBlock
```

```
"Evaluate the argument, aBlock, as long as the value of the receiver is true."
```

```
^self value
```

```
ifTrue:
```

```
    [aBlock value.          まず渡されてきた実行部を一回実行
```

```
    [self value] whileTrue: [aBlock value]]    その後 self （レシーバなのでつまり条件節 (anInteger < 10) に相当）を一回評価して、も
```

```
う一回 whileTrue: で再帰呼び出し。
```

というわけでループは内部的には再帰呼び出し風になっている。では ifTrue: は？と思うとこれが Boolean のインスタンスメソッドであるが、その内容は subclassResponsibility となっている。Boolean は true / false なので、中身を見ると ifTrue: がある。

```
ifTrue: alternativeBlock
```

```
^alternativeBlock value
```

で終わり。ははは。もちろん true に実装された ifFalse: は、

```
ifFalse: alternativeBlock
```

```
^nil
```

で、以上おわり。

・ inject 注入処理

いろいろあるので追いかけるときがないが、

```
| aValue |
```

```
Transcript clear.
```

```
aValue := (1 to: 10) inject: 0 into: [:total :each | total + each].
```

```
Transcript
```

```
    nextPutAll: aValue printString;
```

```
    cr;
```

```
    flush
```

ちなみに結果は 55。これでなぜ total を用いて aValue に繰り込み処理が行われるかはかなり謎。そういう仕様なのだ（一つ前の結果を第一引数に残して再投入する）と思うのがよい。これを inject を使わずに書くとなんぶんこうなる。

```
| aValue |
```

```
Transcript clear.
```

```
aValue := 0.
```

```
(1 to: 10) do: [:each | aValue := aValue + each].
```

```
Transcript
```

```
    nextPutAll: aValue printString;
```

```
    cr;
```

```
    flush
```

何故コレを inject と言うか、というのは分からなくはないが、もう意味はなく覚えるしかない、と青木説。

・ fork

例外や fork もあるが、今回はパス。。。でもちょっとだけ。

fork は [.....] fork のようにして使う。つまり Block Closure を置いて、それに fork メッセージを送る。定義は Block Closure のインスタンスメソッドにある。

```
fork
```

```
"Create and schedule a process running the code in the receiver. Answer the new process."
```

```
| forkedProcess |
```

```
forkedProcess := Process forBlock: self priority: Processor activePriority.
```

```
forkedProcess resume.
```

```
^forkedProcess
```

つまり resume とかいろいろあるが、Process クラスに forBlock: を投げることで実現する。これはクラスメソッドで、インスタンス生成する。定義はこれ。

```
forBlock: aBlock priority: anInteger
```

```
"Answer an instance of me that has suspended aContext at priority anInteger."
```

```
| newProcess |
```

```
newProcess := self new.          おー珍しく(久々に見ただけ?) new してるぜー
```

```
newProcess suspendedContext:
```

```
    [aBlock
```

```
        on: Process terminateSignal
```

```
        do: [:ex | ex return].
```

```
        Processor activeProcess finalTerminate] newContext.    なんか Processor でのがある。suspendedContext として作ってから
```

```
newContext するってことかな？
```

```
    newProcess priority: anInteger.
```

```
    ^newProcess
```

で、動き出してから priority を設定する。へんなの。設定してから newContext すればいい（実行すればいい）のに。違うのか。

ま、とりあえず知りたかったこと（どうやってforkなんてシステム内部の話と接点を持つのだ）はなんとなく分かったような。

Processor クラス、というのはどうも存在せず、Kernel の Shared Variable として存在する Processor がそれらしい。

```
Smalltalk.Kernel defineSharedVariable: #Processor
  private: false
  constant: false
  category: 'Kernel-Processes'
  initializer: nil
```

という感じ。まあこのあたりを使って VM と対話しながら突っ込むのかな。（当たり前だけれどプロセステーブルなどは Smalltalk イメージの中にあるはず。つまり VM に突っ込むところだけが問題なのだけれど、そこはこうした shared variable 経由で対話ということか。それはありそう。）