オブジェクトの広場

［Happy Squeaking!!］

－ オブジェクト指向再入門 －

[第一回] [第二回] [第三回] [第四回] [第五回]

第二回：オブジェクト指向の基礎固め　編（Part 1）

ＩＮＤＥＸ

記事に対するご意見・ご感想をお待ちしています。
umezawa@tyo.otc.ogis-ri.co.jp
気軽にお寄せください。

HOME TOP

オブジェクトの広場

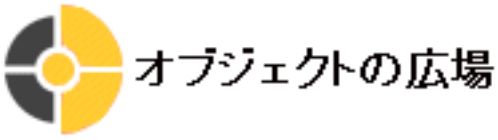
[Happy Squeaking!!]

1．はじめに

古きよき時代と呼ばれるソフトウェアの黎明期には、計算機に行わせる仕事は比較的単純な計算問題にかぎられていました。このため、コンピュータの中には、数字や、単純な計算式が表せれば十分でした。しかし、近年のアプリケーションでは、残念ながらコンピュータに単純な計算のみをさせる要求というのはほとんどありません。銀行口座を作り、それをデータベースに保存したり、インターネット上で、仮想の洞窟に入り複数のユーザで探検ゲームを行ったりなど、非常に複雑な問題解決の仕事をコンピュータにさせることが求められています。こうしたことをコンピュータに行わせる際に、開発者が仕事の内容を、全てコンピュータに合せて数字と式の羅列に変換しなければならないとなれば非常に苦痛でしょう。(とはいえ、古くのアセンブラによるプログラミングはまさにこうした作業でした)。この苦痛はある意味当然のことです。人間は、数字と式の組み合わせで物事を考えているわけではないのですから。

通常、人間は、「もの」を基本的な単位として考えるということをします。具体的なものや抽象的なものも含めて、現実の世界は森羅万象さまざまな「もの」から成り立っていると捉え、特定された「もの」を手がかりにして脳の中で処理が進んでいきます。(「もの」を特定する働きをもつのは「ことば」です。ことばがなければ人間は一切の思考を持つことができません。幼児の言葉の獲得は、思考の獲得の過程でもあるわけです。)
そこで、こうした「もの」を、ソフトウェアの中にも処理の基本単位として表現できれば、複雑なシステムを作成する際にも、考え方が人間に近くなるので対処しやすいのではないか、という考えがでてきました。オブジェクト指向はこのような逆転の発想から生じています。計算機の都合に人間が合せるのではなく、コンピュータを人間の思考に近づけるのです。
「オブジェクト」とは「もの」を指します。もともと米国で生まれた考え方なので横文字になっていますが、無理やり日本語にすれば「もの指向」ということです。

「オブジェクト指向」の考えをシステムに取り入れたわかりやすい例としてGUIがあります。Windows、MacintoshなどのGUIは、現実世界に存在する「もの」をソフトウェア上に比喩(メタファ)として表すということをしています。実際の机に似たものがデスクトップとしてCRTディスプレイ上に表現され、マウスを手のかわりとして、フォルダをつかみ、ごみ箱に捨てたりします。Dosのコマンドプロンプトでrmdirとカチカチ打ち込むのに比べてはるかに人間に優しいシステムになっているということがいえます。
これは使用者から見た感覚ですが、開発者にとってもやはり同じような「もの」を中心とした単位でソフトウェアが構成できればこれにこしたことはありません。こうした野望を実現するためのもっとも近道は、「オブジェクト」という単位を構成要素として扱える開発方法論やプログラミング言語を使うことです。



[Happy Squeaking!!]

2. オブジェクトの定義

オブジェクトをソフトウェアの中に表すといっても、まさか現実世界の「もの」そのものがソフトウェアの中に存在できるわけではありません。あくまで現実世界の「もの」にたメタファとしての「もの」が存在することになります。
つまり、現実世界のもやもやとした複雑怪奇な「もの」を、ソフトウェア上に表現するために、ある程度「ものとはこういうもの」と定義することが必要になります。

一般的にオブジェクト指向の世界ではオブジェクトを以下のようなものとして扱います。

1:識別できる(object id をもつ)
2:属性をもつ (内部にデータを保持できる)
3:操作をもつ (外部からの刺激に対して反応する)

この定義は、大体において現実世界の「もの」に近いといえるでしょう。
一つ欠けている点といえば、現実世界の「もの」は、特に生き物などの場合、「外部からの刺激を受けなくとも自律的に、勝手に動き出す」ということがあります。一般的なオブジェクト指向では通常ここまでは扱いません。後でも出てきますが、オブジェクトは必ず外部からの刺激によって動き出すことになっています。(勝手に動く自律的振る舞いを持つオブジェクトをエージェントと呼び、ソフトウェアの中に表そうという試みもあります。エージェント指向として知られていますが今回の範囲外です。)

1:の識別性は、「これは何々であって、ほかの何々ではない」と区別がつくことを表します。ソフトウェア上にできたAさんの銀行口座は、Bさんの銀行口座とは区別できなければなりません。両方が識別できずに入れ替わってしまったら大変なことです。また、動物育成シミュレーションを作るとして、ポチとシロはやはり区別できねばなりません。銀行口座のように、「口座番号」が振られればお互いの区別は簡単なことです。しかし自分のペットとして飼うことにしたポチとシロにはそうした番号をつけることは通常しません。そこで、こうした番号が振られてないオブジェクトをソフトウェア上で識別するため、オブジェクト指向のシステムでは、オブジェクトIDという番号が、システムによって自動的に振られることになります。(実装によっては単なるアドレスだったりもします)。こうしたIDは開発者は通常意識することはありませんが、識別のために水面下で役立っているのです。

2:では、「オブジェクトはそれぞれ固有のデータを持っている」ということを表しています。銀行口座の例では、Aさんの口座では残高が15,000であり、住所が東京都、Bさんの口座では残高が200,000、住所が大阪府であるということになります。こうしたオブジェクトの持つ属性は、通常は、第三者が勝手に覗き見たり、値を変更したりということはできません。Aさんの残高がいつのまにか第三者に知られて

いたら大変なことです。
オブジェクトが持つデータは、オブジェクトの外からは通常見えないのです。これをオブジェクト指向の用語では「データがカプセル化されている」などと呼んだりします。「情報隠蔽」と呼ばれることもあります。「情報隠蔽」というと何だかいけないことをしているかのようですが、個々のオブジェクトが固有のデータを持ち、外側から勝手に見れないというのは、現実世界に照らしあわせてもごく自然といえるでしょう。

もう一つ、3:についてですが、オブジェクトは、何らかの刺激に対し反応できる能力を持つということです。これを「操作をもつ」という言葉で言い表します。
通常オブジェクトに対し何らかの刺激を送ると、その反応が帰ってきます。例えば、ポチに対し「お手」というと、ポチが反応して手を(足ですか)差し出します。このようにオブジェクトに対して刺激を送ることを「メッセージを送る」と通常いいます。メッセージを送る側が「センダ(送信者)」、受け取る側は「レシーバ(受信者)」です。
送られてきたメッセージに対して、レシーバが反応できるのは、レシーバの側にそのメッセージに対応した反応の仕方が覚え込まれていたからに他なりません。ポチの例ですと、「お手」という刺激に対する一連の動きを教え込まれて知っていたからこそ、前足が差し伸べられたということになります。この、「刺激に対する反応の仕方」は、「メソッド」という風に呼ばれています。

オブジェクト指向でのプログラミングの処理は、あるオブジェクトに対してメッセージ送信を行い、その結果を受けてさらにあるオブジェクトにメッセージを送るといった処理の連続によって成り立っています。
現実世界でも、多くの仕事はこのような「もの」と「もの」同士の間の連鎖反応的なメッセージ送信の繰り返しで行われていると見ることができます。、ソフトウェアの販売という仕事を例にとれば、実際に販売する人や、在庫管理をする人、マーケティングをする人など、様々な役割をもつ人々(オブジェクト)が、互いに連絡をとりあいながら(メッセージ送信)全体の販売処理を進めていっているのです。

Prev. Index Next

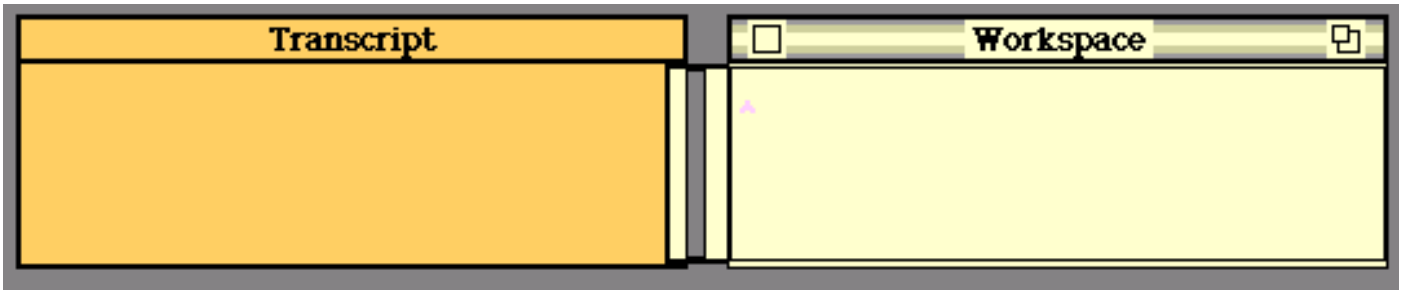
オブジェクトの広場

[Happy Squeaking!!]

3. *Squeak 演習：オブジェクトにメッセージを送る*

それでは、オブジェクトに対して実際にメッセージを送ってみましょう。

Squeakを起動し、ワークスペース、トランスクリプトを開きます。

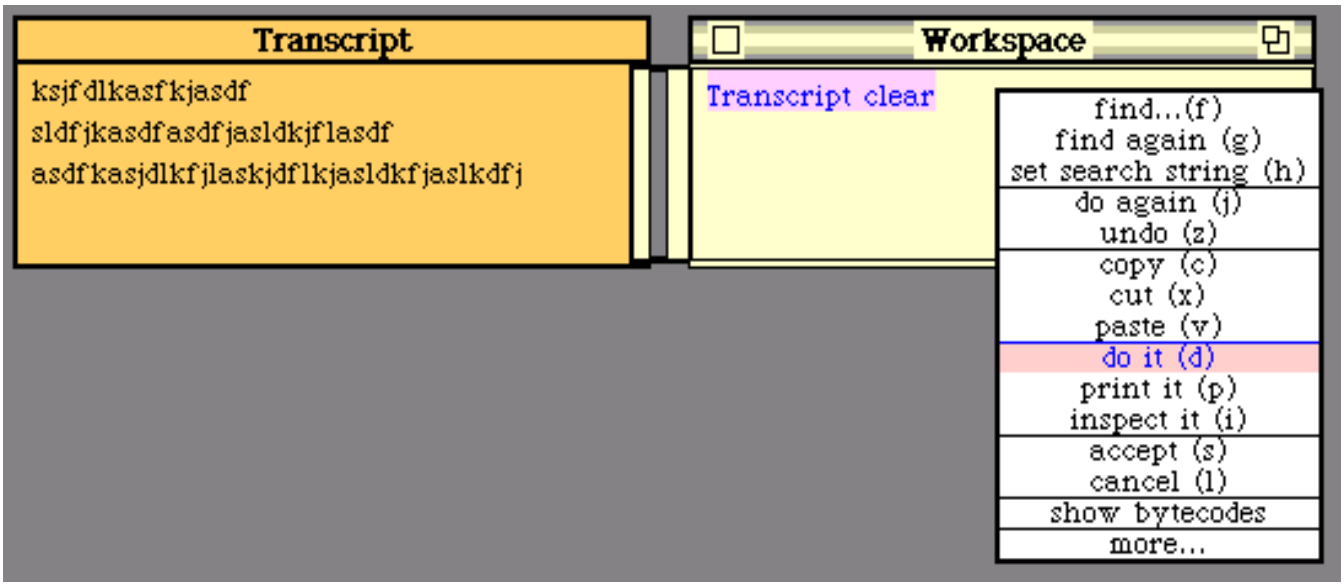


ワークスペースとトランスクリプトを開く

トランスクリプトウィンドウになにやら適当な文字列を書き込んでください。その後ワークスペース上で

```
Transcript clear.
```

と書き、do itしてください。



Transcript に clearメッセージを送る

Transcript の内容がクリアされるはずです。

この場合、Transcriptがレシーバとなるオブジェクト、送られたメッセージはclear、センダはあなた自身ということになります。

> SmallTip: Smalltalkでオブジェクトにメッセージを送る構文は、<オブジェクト>スペース <メッセージ> ピリオドとなる。

ピリオドは、後に実行が続かない限り省略可能です。ここでは、Transcript clearで処理が終わるので本当はなくてもいいのですが、最初ですし、つけておきましょう。

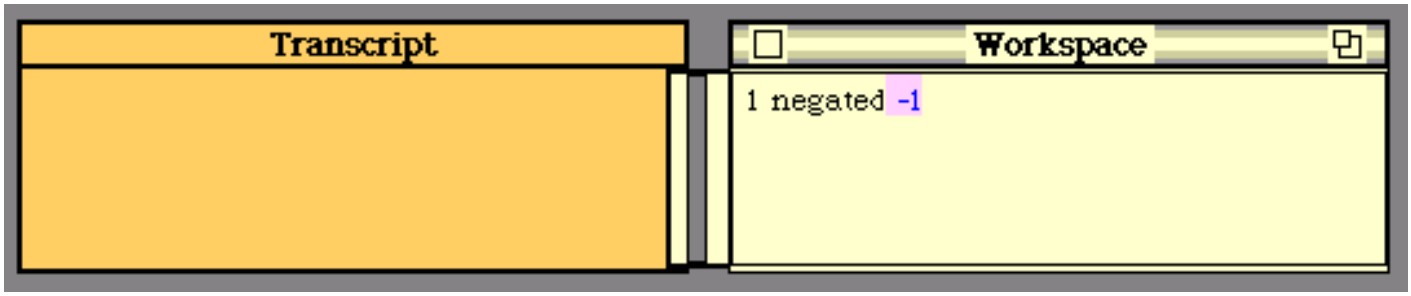
では、いろいろなオブジェクトにメッセージを送ってみましょう。

今までは"do it"で実行を行っていましたが、今後は実行の後の結果(オブジェクトが処理を行った後の反応)が見たいので、"do it"の代わりに"print it"を使ってみてください。Transcript はウィンドウとして目に見えるのですぐに反応の結果がわかりますが、他のオブジェクトではそうはいきません。

ソースの意味は"のコメントで示しています。結果をみるため一文ずつ選んで実行してください。

```
"1 の符号変換"
1 negated.
```

"print it"を実行すると以下のように、すぐとなりにハイライトされて実行の結果が表示されます。



1 negated の実行

> SmallTip: "print it"実行で、メッセージ実行の結果を表示できる。

> SmallTip: Smalltalkのコメントは"(ダブルクォーテション)で囲む。

では同じ要領で、以下を"print it"実行しましょう。

```
"文字列に 長さを聞く"
'abc' size.

"文字列の 先頭を大文字にする"
'xyz' capitalized.

"文字に アスキーコードを聞く"
$A asciiValue.

"配列の最後の要素を聞く"
#( 1 2 3) last.
```

大文字小文字の区別はSmalltalkでは厳格ですので注意してください。

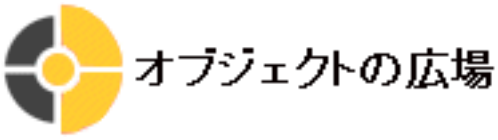
SmallTip: 文字列オブジェクトは' (シングルクォーテーション)で囲む。

SmallTip: 文字オブジェクトは$(ダラー)を先頭につける。

SmallTip: 配列オブジェクトは#() で囲み、要素をスペースで区切る。

Smalltalkでは、数字、文字列、文字といったよく使われる基本的なオブジェクトは、リテラルとして文字列を書くことによって即座に生成できます。
"do it"をしたときにSmalltalkのインタプリタ(正確にはコンパイラ)が文字列を解析して、「先頭が$ならば文字オブジェクトを生成」といった処理を自動的にしてくれるのです。その他もっと複雑なオブジェクトは、後にでてくるようにクラスオブジェクトに対するnewメッセージで生成するのが典型的です。

Prev. Index Next



［Happy Squeaking!!］

3. Squeak 演習：オブジェクトにメッセージを送る

3. 1 連続的なメッセージ送信

次にオブジェクトの連鎖反応的な気分を味わうために連続してメッセージを送ってみましょう。

```
"文字の配列の最後のアスキーコードを聞き、符号を反転する"
#( $a $b $c ) last asciiValue negated.
```

これはどう解釈すればいいのでしょうか?
オブジェクトに対するメッセージ送信が前から順番に行われていくことに注意してください。

まず #($a $b $c)という配列オブジェクトがあり、それに対して last メッセージが飛びます。これの反応として返ってくるのは $c オブジェクトです。
次に$c オブジェクトに対し、 ascillValue メッセージが送られます。これの反応は 99という数字オブジェクトになり、返ってきます。
さらに、99オブジェクトに対して negated メッセージが送られます。この反応は - 99 オブジェクトとして返ることになります。

疑似コード風に書くと以下のようになります。

```
#($a $b $c) last => $c
$c asciiValue => 99
99 negated => -99
```

> SmallTip: メッセージの反応結果のオブジェクトに対して続けてメッセージを送ることができる。

反応結果として返ってくるオブジェクトを変数に代入することもできます。このようにわけて書くと、処理の流れがわかりやすいかもしれません。以下を実行してみることにします。
(これだけ長いとうち間違いをおこしがちです。コピー&ペーストで対処してください)

```
| arr lastElem ascValue answer |
arr := #($a $b $c).
lastElem := arr last.
ascValue := lastElem asciiValue.
answer := ascValue negated.
^answer.
```

今度は一文でなくすべてを選択して"print it "します。

文法的に新しい要素が幾つかでているので順番に解説します。

まず||でこれから使う変数を宣言しています。ワークスペース上で変数を使う場合は必ずこの形での変数宣言が必要になります。これは実行とともに一時的に宣言され、代入され、使われた後は消えてしまうので、一時変数と呼ばれています。

次が代入です。代入は := (コロンとイコール)で表現します。Squeakの場合は←も使用できます（アンダースコアで入力）。←はSmalltalk-80で使われていた歴史的なごりです。ANSIのSmalltalkに従い、本講座では、一貫して:=を使用していきます。コロンとイコールの間にスペースを空けたりしないように続けて書くようにしてください。一方コロン前とイコール後はみやすさのため通常スペースを空けます。

後はメッセージの送信結果を順番に変数に代入し、実行しているだけです。

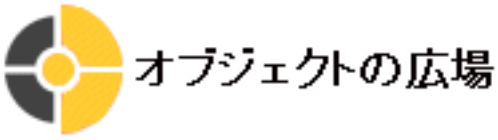
最後の^(キャレット) は、実行結果としてのオブジェクトをメッセージの送り手側に返すために使います。手続き型言語(Cなど)でのreturnに該当します。Squeakの場合は上矢印(↑)で表示されますが同じ意味を表しています。

上の例の場合ではanswerに代入されたオブジェクトを最終結果として送り側に返したいので^answerと書いています。例えば別の値を返すのであればここは、例えば^lastElemなどとします。(その場合answer変数を代入にしか使っていないために警告がでますが、そのまま実行できます)

SmallTip: 一時変数は || で囲む。
SmallTip: 変数の代入は := で行う。
SmallTip: 明示的にあるオブジェクトを返却させるには ^ 記号を使う。

なお、Smalltalkでの代入は、全て参照によるものです。代入によってオブジェクトのコピーが変数の内部にできるわけではありません。その意味では、単にオブジェクトに := によって一時的な名前をつけていると考えることもできます。

Prev. Index Next



［Happy Squeaking!!］

3. Squeak 演習：オブジェクトにメッセージを送る

3. 2 様々な種類のメッセージ

Smalltalkではメッセージの種類を3種類に分割しています。

3. 2. 1 単項メッセージ

オブジェクトにメッセージを送る際に、送り手側で、なんらかのデータを受け取り側に渡してあげたいときがあります。
例えば「Aさんの銀行口座オブジェクトに、新たに3000円振り込みたい」という時に、3000円というデータを銀行口座オブジェクトに渡す必要があります。また、「#($a $b $c)という配列オブジェクトの2番目をとりだしたい」という時には、「2番目」ということを配列オブジェクトに知らせる必要があります。

今まで紹介してきたメッセージにはこうした送り手側のデータが含まれませんでした。

```
'Hello Smalltalk' size. "文字列に大きさを聞く"
```

こうした受け渡すデータ(引数)のない単純なメッセージをSmalltalkでは単項メッセージと呼びます。

3. 2. 2 2項メッセージ

2項メッセージは、受け渡すデータを一つだけ含みます。

!%&*=,/ó ?@ｬ| のどれかの記号を任意にえらんでメッセージ名とします。
これらの記号は、2項メッセージ専用として予約されています。

```
"3オブジェクトに+メッセージを送る、引数は1"
3 + 1.

"文字列オブジェクト'Hello'に連結を行う。引数は'Smalltalk"
'Hello', 'Smalltalk'.
```

通常、数字や文字列などの単純なオブジェクトに対するメッセージ送信で、2項メッセージは使われます。

それでは以下の実行結果はどうなるでしょうか。

```
3 + 5 * 4
```

これは、Smalltalkでは32 になります。
3 オブジェクトに + のメッセージが送られ、引数は5です。
結果として返るオブジェクトは 8、次に * メッセージが 引数 4で送られ、32というオブジェクトが返ってくることになります。

Smalltalkでは演算子といったものを特別扱いしません。あくまでオブジェクトに対してメッセージが順番に送られることになります。

3. 2. 3 キーワードメッセージ

キーワードメッセージには一つ以上のパラメータ(引数)が含まれます。
メッセージ名に: (コロン)が含まれるのが特徴です。
データはそのコロンの後に指定されます。

引数1つの場合
"Transcript にshow: メッセージを送る。引数は'HelloWorld"
Transcript show: 'HelloWorld'

"配列オブジェクト #($a $b $c)の2番目を取り出す"
#($a $b $c) at: 2

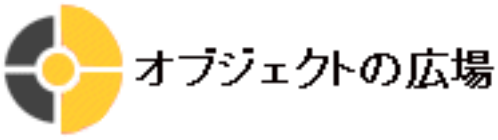
引数2つの場合
"文字列オブジェクト 'HelloSmalltalk' の2番目から5番目を取り出す"
'HelloSmalltalk' copyFrom: 2 to: 5.

引数3つの場合
"文字列オブジェクト 'HelloSmalltalk'の2番目から5番目を'appy'に変更する"
'HelloSmalltalk' copyReplaceFrom: 2 to: 5 with: 'appy'

以下4つ、5つと同じように続きます。
Cなどの言語では、引数が複数ある関数は、()で括りコンマで区切ることで指定を行います。
上記の例だとcopyReplace(2,5,'appy') などとなります。
Smalltalkの場合、引数の区切りを、コンマ記号でなく、キーワードとしてコロンが末尾についた任意の英字で書ける(この場合to:や with:)ので、非常にそれぞれの引数の意味をわかりやすくできます。

なお、キーワードメッセージは一続きでメッセージ名とみなされます。順番を入れ替えた場合は別のメッセージとなります。
anObject to: 3 from: 5と anObject from: 5 to: 3 は別ものです。片方がto:from:というメッセージ名で、後者がfrom:to:というメッセージ名になります。

SmallTip: メッセージには単項、2項、キーワードの3種類がある。



［Happy Squeaking!!］

3. Squeak 演習：オブジェクトにメッセージを送る

3. 3 複数メッセージの実行

3. 3. 1 メッセージの優先順位

単項、2項、キーワードメッセージは、組み合わせて書かれた場合、優先順位にもとづいて実行されます。
優先順位は以下のようになります。

単項 > 2項 > キーワード

例でみてみましょう。

```
'HelloSmalltalk' at: 1 negated + 7.
```

結果は $Sとなります。予想がつくでしょうか。

'HelloSmalltalk' at: 1 negated + 7. の文全体で、まずもっとも優先度の高いメッセージは 単項メッセージのnegatedです。従ってこの部分が最初に実行され、結果として-1が返ります。

```
'HelloSmalltalk' at: -1 + 7.
```

次は、2項メッセージの + が実行されます。

```
'HelloSmalltalk' at: 6.
```

そして最後はキーワードメッセージの at: の実行になります。

一見ややこしそうですが、この優先順位のおかげで、複雑なオブジェクト関の一連のやり取りも、一つの文の中にコンパクトに書くことができるのです。

実行の流れを、変数に代入する冗長な形式で書くと以下のようになります。

```
| minus index |
minus := 1 negated.
index := minus + 7.
'HelloSmalltalk' at: index
```

慣れれば、このようなやり方はやぼったく思えてきます。Smalltalkなのですから、ごたごた書かずに small talkでいきましょう。

> SmallTip: メッセージの優先度は 単項 > 2項 > キーワード の順。

また、優先順位を高めたい場合には()でくくることもできます。

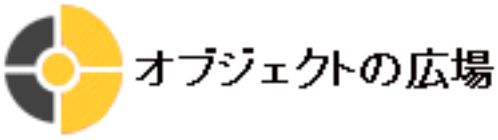
例として、

```
3 + ( 5 * 4 )
```

と書くことができます。この場合の結果は 23 になります。

SmallTip: ()でくくろことで優先度を高めることができる。

Prev. Index Next



[Happy Squeaking!!]

3. Squeak 演習：オブジェクトにメッセージを送る

3. 3 複数メッセージの実行

3. 3. 2 カスケード（メッセージの流し込み）

Smalltalkでは、特定のオブジェクトに対して連続してメッセージを送り込むための一種の省略記法が用意されています。

例えば､車を表すcarオブジェクトがあったと仮定しますと、以下のように；（セミコロン）を使ってつなげて書くことができます。

```
car start;
        turnRight;
        turnLeft;
        stop.
```

これは、

```
car start.
car turnRight.
car turnLeft.
car stop.
```

と書いたの同じことです。同じオブジェクトをレシーバとして連続的にメッセージを流し込むときに、便利な書き方です。このことからカスケード(滝)と呼ばれています。

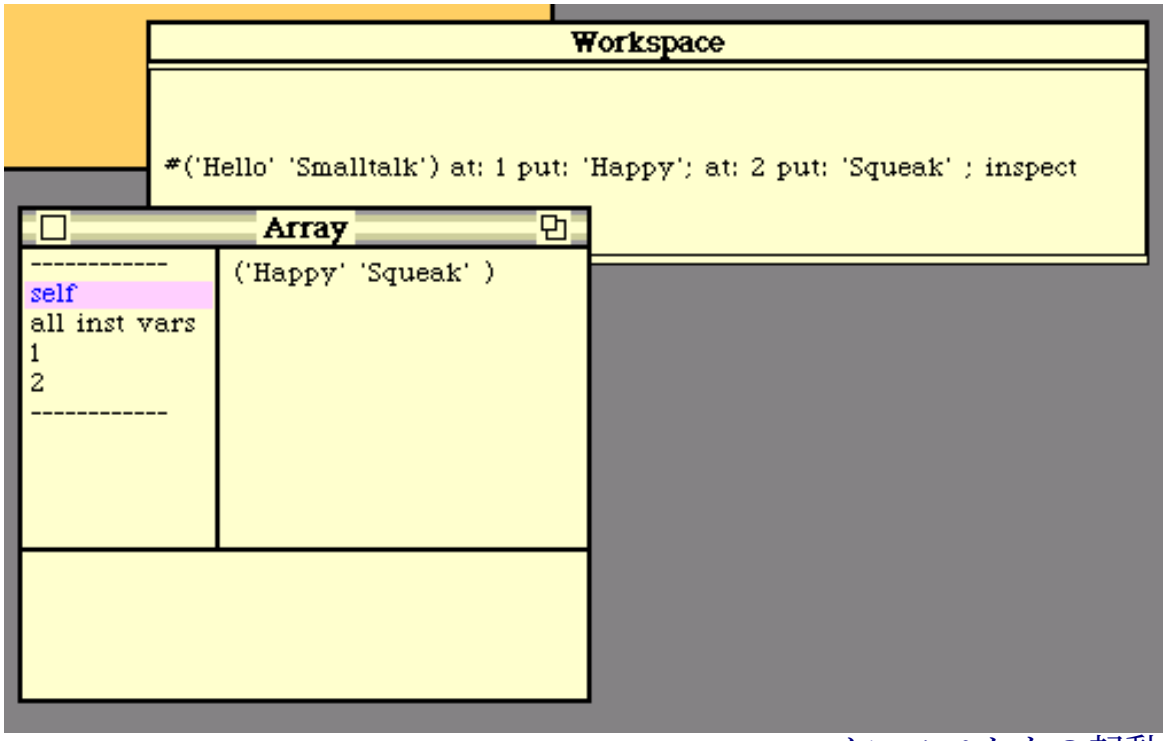
配列オブジェクトで実行してみると以下のようになります。

```
| targetArray |
targetArray := #('Hello' 'Smalltalk').
targetArray at: 1 put: 'Happy';
                        at: 2 put: 'Squeak' ;
                        inspect.
```

と書くことができます。(特に改行の必要はありません、わかりやすさのためにあえて行っています)。

> SmallTip: 同じレシーバに対し、メッセージを次々送りたい場合にカスケードの表記を使うことができる。

ちなみに上記の例を実行すると、以下のようなウィンドウが立ち上がります。



インスペクタの起動

これはインスペクタというツールです。カプセル化されていて中身の見えないはずのオブジェクトも、このツールを使えば見ることができてしまいます。

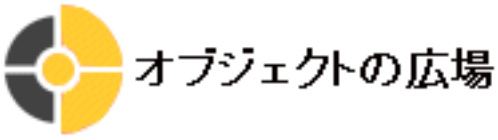
オブジェクトの中身を見たいときは、そのオブジェクトに対し、inspectというメッセージを送ればいいわけです。

余裕のある方は、以下を試してみましょう。

```
'abcd' inspect.
$c inspect.
#( $a 'complexArray' #( $a $b $c)) inspect
```

> SmallTip: オブジェクトの中身は inspectメッセージを送ることで調べることができる。

Prev. Index Next

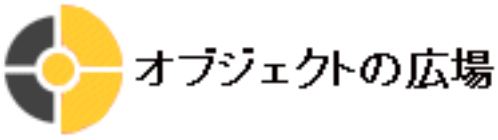


［Happy Squeaking!!］

4. クラスとインスタンス

4. 1 クラスとは

現実世界には、様々なオブジェクトが存在しています。そうしたオブジェクトはすべてバラバラに頭の中でとらえられているものなのでしょうか? この世界に存在する全てのオブジェクトを、別個の知識として頭の中に貯えていたのでは、あまりに情報量が多すぎて大変なことになってしまいます。実際にはオブジェクトの間には類似性があり、そうした類似性に着目することで、私たちは頭の中に貯えていなければならない情報の量を激減させています。
例えば道端で、田中さんのところのポチを見かけた場合、「これは犬だ」ということを無意識に頭の中で思い浮かべています。シロを見かけた場合もやはり、「これば犬」と思います。ここで私たちは、ある特定の動物に対し、「犬」という分類を行っているのです。これによってポチやシロに対する知識を別々に持たなくとも、「ポチは犬だからワンとなく。しっぽを振っているのは機嫌がいいからだ」といったことを把握することができるのです。
クラスとは、このように、多くのオブジェクトに関する共通性を取り出し、それを分類するための「もの」を表しています。英語のclassとは「組」や「階級」といったことですのでまさに文字通りの意味になっています。
クラスは分類の単位として任意に設定することができるものです。何を分類の基準とするかは、分類する人や、場合によって異なってきます。例えば「犬」という分類の単位だけでは、ペットブリーダーにとっては多少おおざっぱすぎる単位でしょう。「ポメラニアン」や「チワワ」、「柴犬」といったより細かな分類が必要になります。オブジェクト指向のシステムを作る際にも、この「クラス分け」がきちんと行われていないと、概念の整理がされていないわかりにくいものになってしまいます。



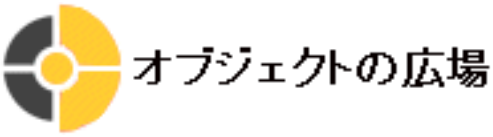
［Happy Squeaking!!］

4. クラスとインスタンス

4. 2 インスタンスとは

特定のクラスに属し分類分けされる「もの」を「インスタンス」と読んでいます。例えば「猫」というクラスに対しては、「タマ」「ミケ」といったインスタンスが存在します。Instanceとは「具現化したもの」「実体」といった意味です。
クラスもインスタンスもともに「もの」ですので、両者ともにオブジェクトとして考えることができます。但しこの2種類のオブジェクトは互いの役目が違います。クラスはインスタンスを分類するために存在する抽象的なもので、インスタンスは、特定クラスにより分類される具体的なものです。
通常インスタンスは具対物として実際の処理を担当します。「犬」、「猫」といった抽象的な分類の概念オブジェクトだけでは何も具体的な処理をおこなわせることはできませんが、「ポチ」や「タマ」ならば、電話番をさせたりお使いにいかせたりできます。(タマは所詮猫なのでだめでしょうが)。

Prev. Index Next



［Happy Squeaking!!］

4. クラスとインスタンス

4. 3 設計図、工場としてのクラス

クラスは単純に分類のために存在するだけでなく、クラスに属するインスタンスを作り出す工場のような役割を持ちます。自分に属するインスタンスを定めるために、クラスは「（自分のところに属する）インスタンスとはこれこれこういうもの」という「インスタンスの仕様」「設計図」が定義されたものになります。この定義に基づいて実体としてのインスタンスが作られていくことになります。

例えば、「会社員」クラスについて考えてみましょう。
「会社員とは、これこれこのようなものである」という記述が「会社員クラス」になります。

会社員クラス:

会社員クラスでは会社員のインスタンスを以下のように定める 属性として: 「社員番号、入社年度、名前」 を持つ。 操作として: 「名刺を渡す、仕事をする」 ができる。 操作のやり方: 「名刺を渡す」には… 「仕事をする」には…

会社員インスタンスが作られたときに、インスタンスは属性として「社員番号、入社年度、名前」をもち、外部に提供する操作として「名刺を渡す、仕事をする」という定義がされています。さらに、定義された操作に対応して実際の操作のやり方も含んでいます。

この設計図に基づいて会社員のインスタンスが作られます。
例えば、山田さんという会社員がいたとすると以下のようになります。

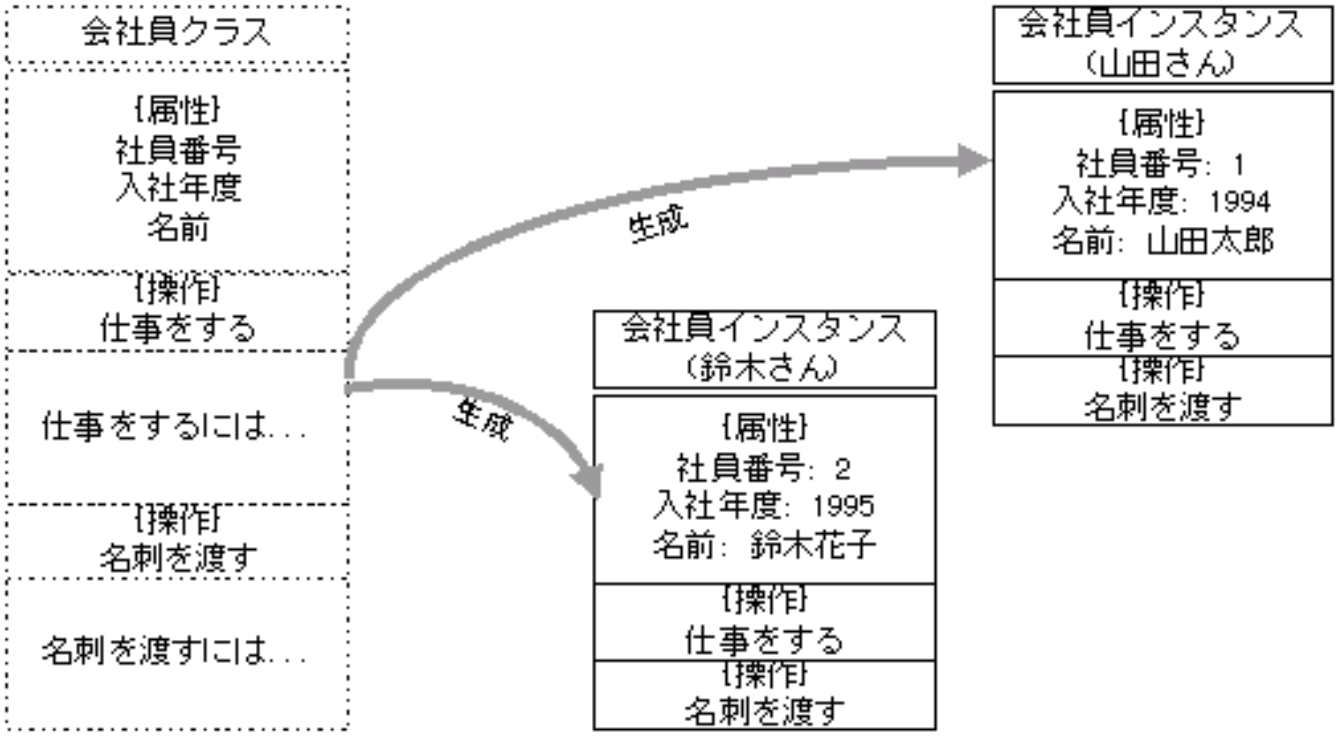
会社員インスタンス

属性: 1234567(社員番号)、1994(入社年度)、山田 太郎(名前) 操作: 名刺を渡す、仕事をする

クラスは、インスタンスの設計図を持っており、外部からの「インスタンスがほしい」という要求に反応して設計図からインスタンスを生成し、返却することになります。インスタンスは、クラスで定義された属性に対応した具体的な属性値を持ちます。外部からの操作を呼び出す刺激を受け取ると、クラスで定義された操作のやり方(メソッド)に基づいて実際の動作を行います。

クラスがインスタンスを作るといっても、クラスが、インスタンスを大量に集合として持っているわけではありません。クラスはできあいのインスタンスの入れ物ではなく、インスタンスを生成するためものです。
クラスは、「インスタンスの定義情報」を情報として持ち、「インスタンス生成」という操作を持つ工場オブジェクトなのです。

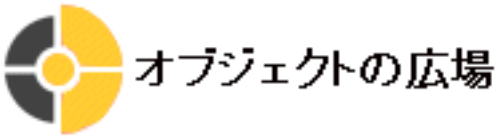
図で示すと以下のようになります。



クラスからのインスタンスの生成

オブジェクト指向での処理は、いくつものオブジェクトが処理中に作成され、それらが互いにメッセージを送りあう相互作用として行われていきます。
この場合、大部分のオブジェクトはインスタンスであり、時折、インスタンスを作るため、クラスオブジェクトに対し、作成用のメッセージが飛び交うことになります。

Prev. Index Next



［Happy Squeaking!!］

5. Squeak 演習：クラスを作成する

5. 1 銀行口座クラスの生成

それでは、Squeakの中で実際にクラスを作成してみましょう。
この演習では、まず銀行口座クラスを作成し、そこから銀行口座のインスタンスを生成してみることにします。

銀行口座とはそもそのどのようなものとして定義できるでしょうか。

属性として、

口座番号
現在の残高

を持ち、

操作として、

初期化
預け入れ
残高照会

を持つというのはどうでしょうか。実際にはもっと細かな定義ができると思いますが、練習ですのでこの程度に止めておきます。

それでは、銀行口座クラスをSqueak内に定義することにしましょう。
現バージョンのSqueakは日本語の入力ができませんので、クラスの名前は、便宜的にAccountというふうにします。
ワークスペースを開き、以下のように書き、"do it"してみてください。

```
Object subclass: #Account
        instanceVariableNames: 'id money'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'BankApplication'
```

実行すると、AccountクラスがSqueak内に作成されます。

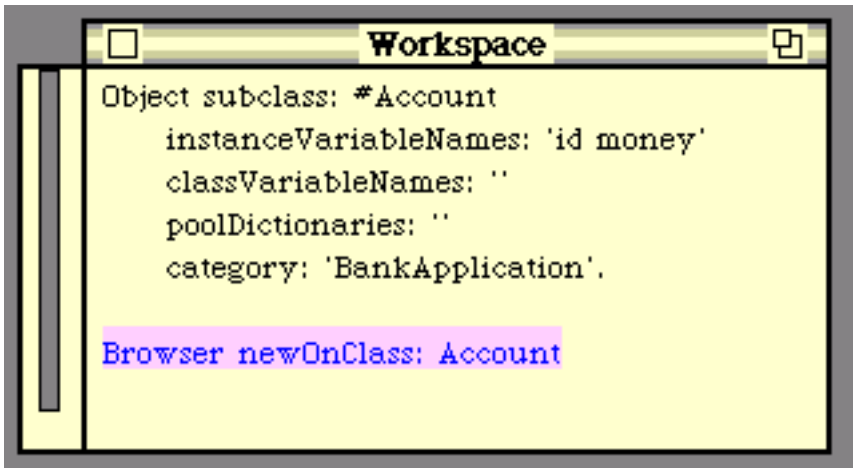
勘の鋭い方はお気づきかもしれませんが、この実行文は、前回までの演習で行ったものと同じ、オブジェクトに対するメッセージ送信の形をとっています。

subclass:instanceVariableNames:classVariableNames:poolDictionaries:category: というキーワードメッセージを'Object'という名前のオブジェクトに送っています。が、最初ですのでここではあまり詮索せずにとりあえず実行してください。

2行目のinstanceVariableNames: 'id money'のところで、インスタンスが持つべき属性として、id(口座番号)とmoney(残高)の二つを定義しています。操作に関してはまだ一切の定義はおこなっていません。3行目と4行目はまだ取り上げていませんので無視します。 5行目のcategory: では、これから作成する銀行アプリケーションが共通して置かれるクラスカテゴリを示しています。

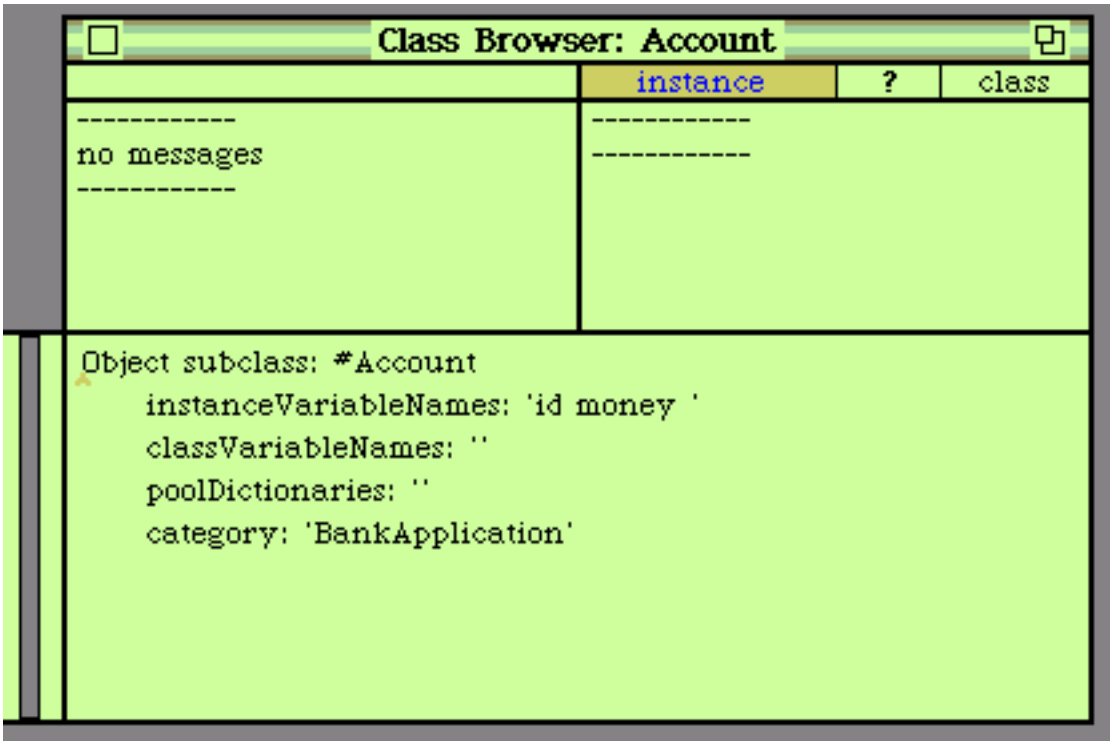
> クラスカテゴリは、多くのクラスを意味的に纏め上げるためのパッケージの単位を表します。Squeakのイメージ内では、700近いクラスが既に存在しています。これらがフラットに存在していたとすると、どれがどのようなクラスなのかわかりにくいものになってしまいます。そのためSqueakでのクラスは必ず何らかのクラスカテゴリに属しています。 クラスカテゴリはJavaでのpackageにほぼ該当しますが、名前空間を規定するものではありません。また、一部のSmalltalkでは、クラスカテゴリがなく、別の方式をとっているものもあります。

できたクラスは、クラスブラウザによってみることができます。Browser newOnClass: Account. とワークスペースで"do it"してみてください。



作成されたクラスをブラウザで見る

以下のようなクラスブラウザが立ち上がります。



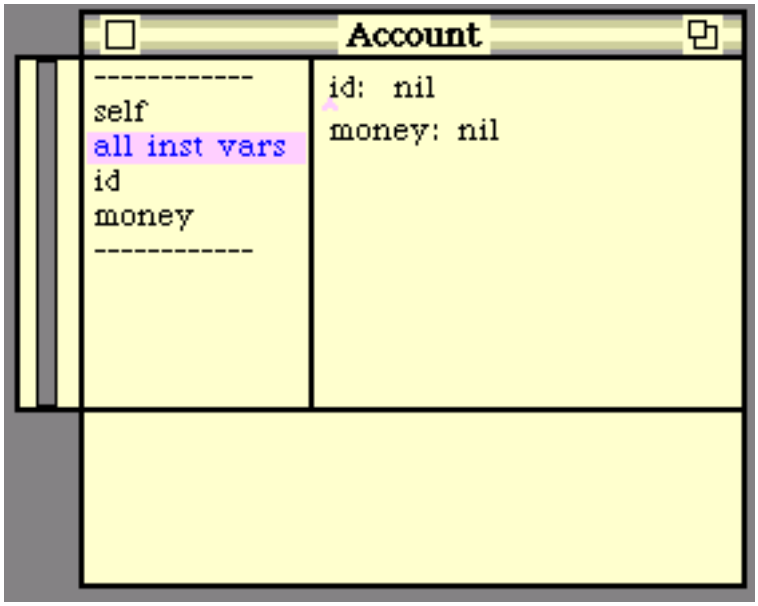
Accountクラスについてのクラスブラウザ

いまの時点ですでにAccountクラスから、Accountのインスタンスを作成することができます。

ワークスペースで以下を実行してみましょう。

```
| account |
account := Account new.
account inspect.
```

以下のようにインスペクタが開きます。インスペクタの左上の窓で、クラスで定義されている属性の名前、インタンスが実際に保持する属性の値を見ることができます。Idとmoneyフィールドの値はどのようになっているでしょうか。



Accountインスタンスのインスペクト

値はどちらも nil になっています。変数に対する代入を特に行わない場合、Smalltalk では、変数の初期値は必ず nilオブジェクト (何も無い、空という意味)になります。

SmallTip: Smalltalkでは変数の初期値は nilオブジェクトとなる。

さて、これだけではAccountのインスタンスは何もすることができません。id や、moneyといった属性が定義されているのにも関わらず、そこに値を代入することも、値を返すことすらできないのです。これはAccountのクラスに操作がまだ定義されていないからです。

更につづく


Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

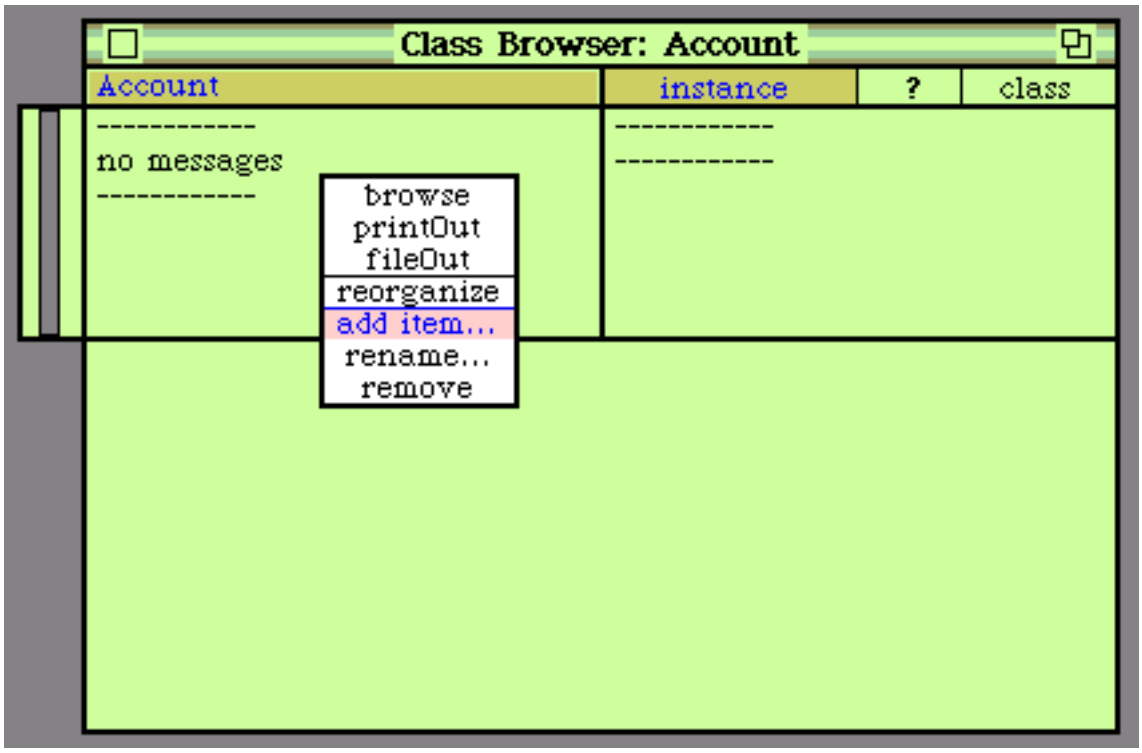
演習：クラスを作成する

5. 1 銀行口座クラスの生成 つづき

それでは次に操作を定義していくことにしましょう。
まずは手始めに、nilの値が入っているid, moneyの変数を初期化できるようにします。

操作の定義は先ほどのクラスブラウザ上で行います。

右上の'no messages ’と表示されている部分から、右クリックで"add item…"を選択してください。

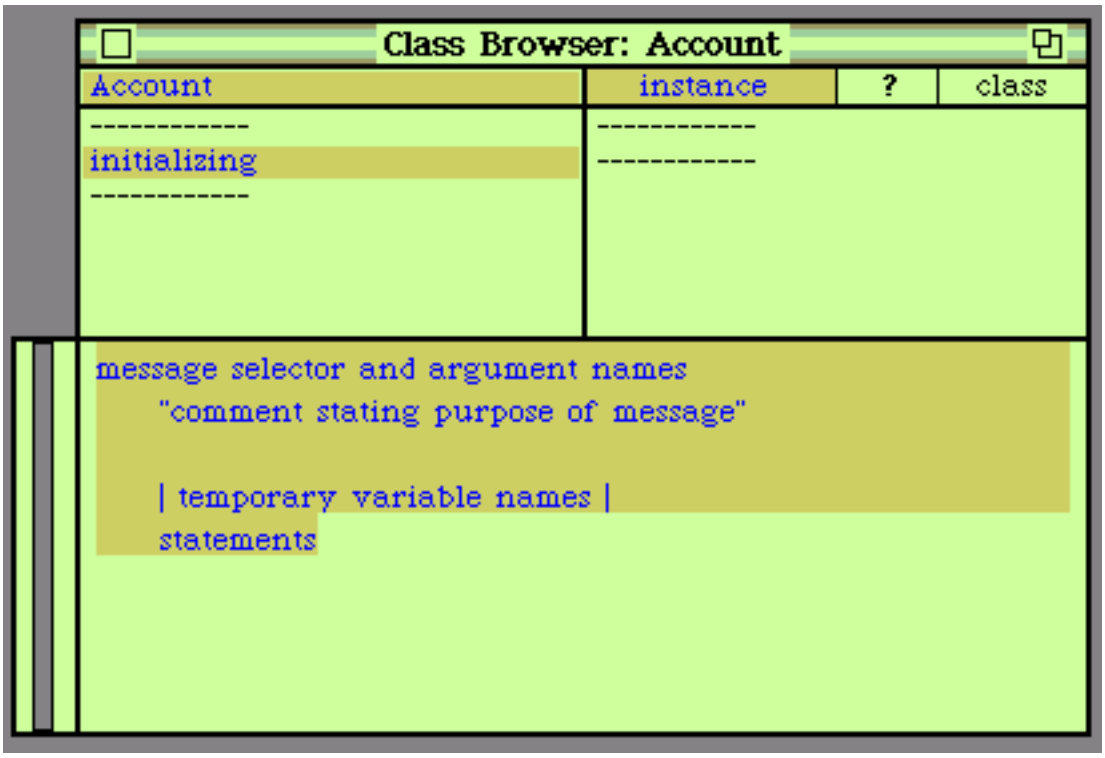


クラスブラウザ上からの操作の追加

すると、カテゴリ名を聞いてきます。このカテゴリは、先ほど説明したクラスのカテゴリではなく、これから定義しようとしている操作のカテゴリ(メッセージカテゴリ)です。
メッセージカテゴリはクラスカテゴリと同様に、メソッドを意味的に纏め上げる単位です。別名プロトコルと呼ばれています。

SmallTip: 操作を纏めるための意味的なもとまりとしてメッセージカテゴリがある。

ここでは初期化用の操作を定義するので、カテゴリ名を"initializing"としておきます。
カテゴリを入力するとブラウザの一番下の部分がハイライトされます。



操作定義のテンプレートの表示

ここに表示されているのは、操作を記述するためのテンプレートです。
このテキストを編集することによって操作をAccountクラスに定義することができます。

テンプレートは以下のようになっています。

```
message selector and argument names
        "comment stating purpose of message"

        | temporary variable names |
        statements
```

1行目で操作の名前を記述します。
初期化操作なので名前は initializeです。

2行目はコメントを書く欄になっています。単なる変数の初期化処理を書くだけですのでコメントは不要でしょう。

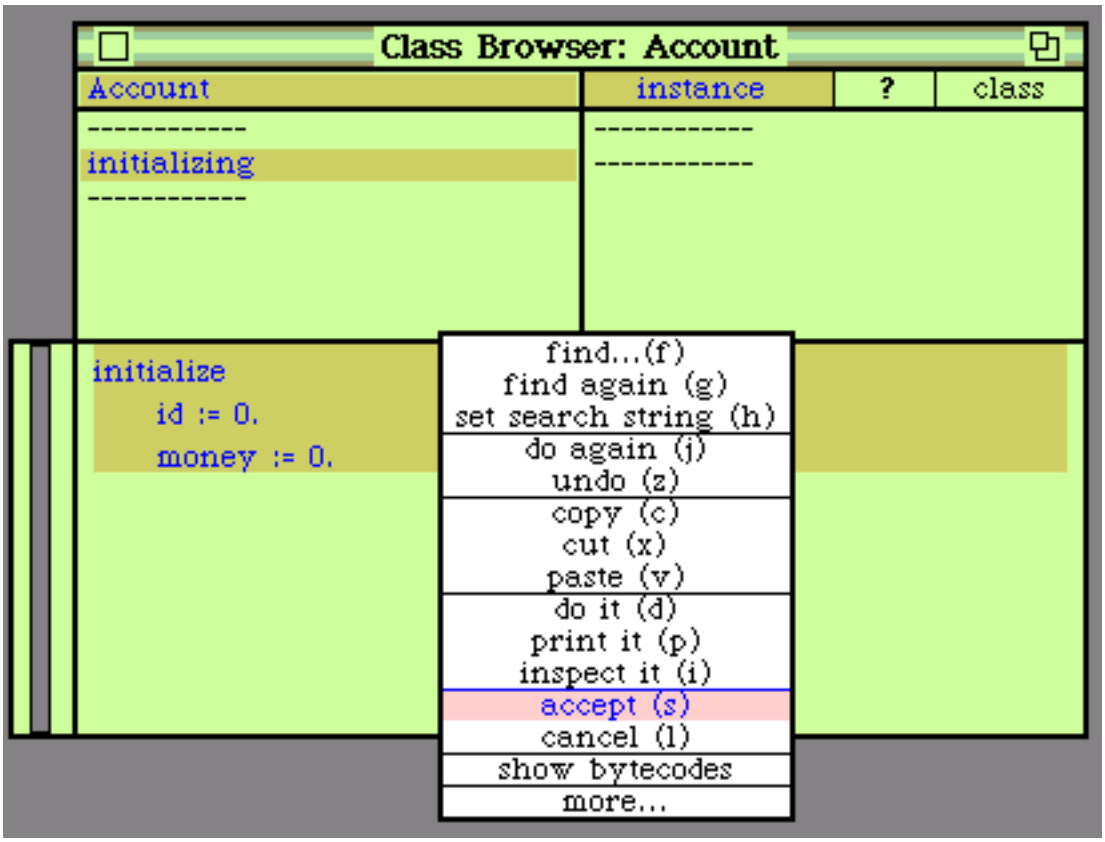
3行目はこの操作内でテンポラリ的に使う変数を定義する欄です。
今回は使いませんので消してしまってください。

4行目意向で実際の処理を書きます。
属性として、インスタンスはid、moneyを持つことになりますので、これを適当な値に初期化するようにします。両方とも0の代入でいいでしょう。

以下のようなコードになります。

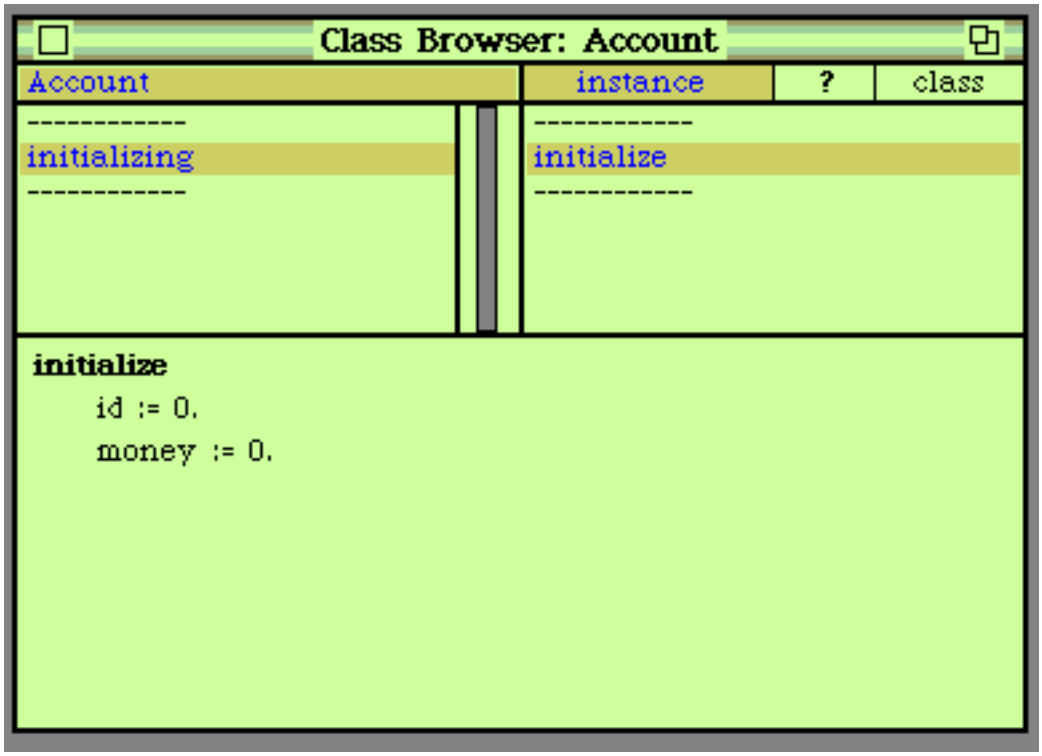
```
initialize
        id := 0.
        money := 0.
```

書き終わったところで右クリックをして、"accept" (受け入れ)を選択してください。



操作の"accept"の実行

これによりAccountクラスにinitializeという操作が定義されたことになります。
以後は右上のリストにinitializeが表示されます。



Accountクラスに定義されたinitialize操作

同様の手順で残高照会と預け入れができるようにします。左上のリストをからポップアップメニューを出し、"add item.."でメッセージカテゴリを追加してください。銀行口座の外部サービスとなる操作が収められるカテゴリということで、"services ”とでもつけておきましょう。

まず残高照会としてgetBalance操作を定義します。
インスタンスの現在の残高は、moneyという変数に属性として代入されていると仮定していますので、単純にmoney変数の値を返すように記述します。

結果として以下のようなコードになります。

```
getBalance
        ^money
```

書き終わったら"accept"をお忘れなく。

つづいて、預け入れとしてdeposit:の操作を定義します。

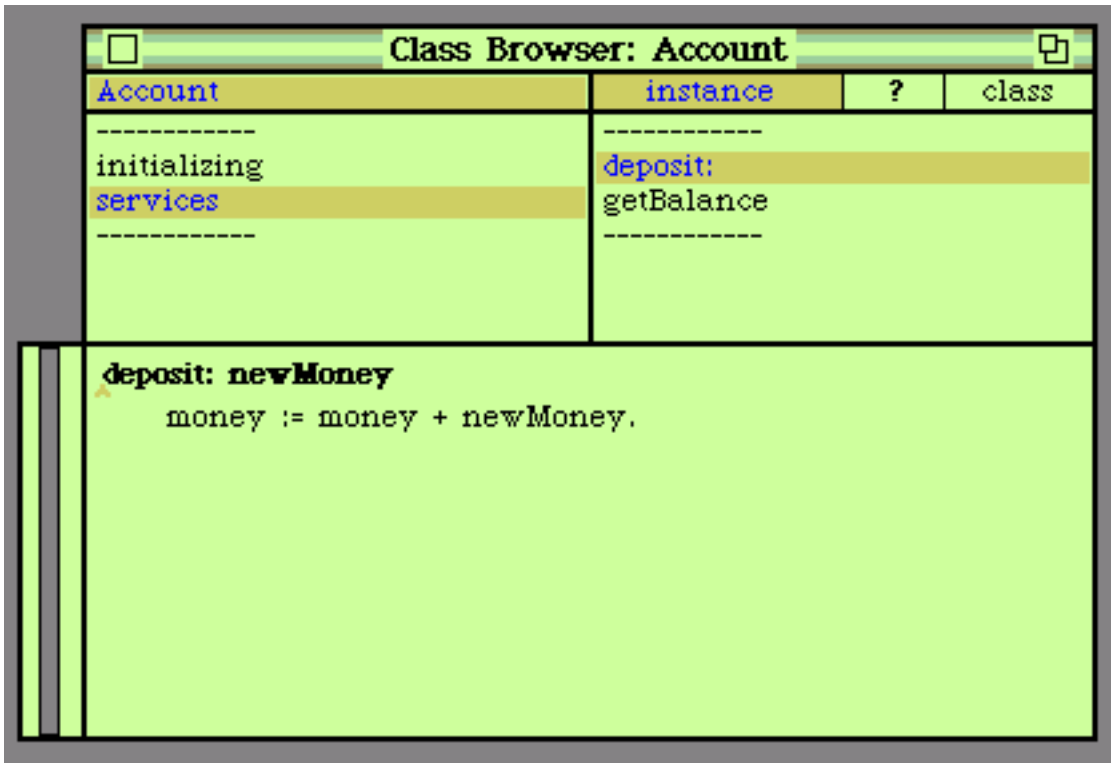
今、クラスブラウザの下の部分に表示されている getBalanceのコード部分を以下のように書き換えてください。(メッセージカテゴリが同一の"services"のため)

```
deposit: newMoney
        money := money + newMoney.
```

同様に"accept"です。

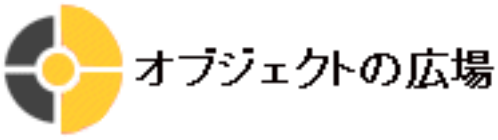
これで3つの操作がすべてAccountクラスに定義されたことになります。

Accountクラスの作成がこれで一応完了しました。



全ての操作が定義されたAccountクラス

ここで acceptした処理内容は、まだ実行されているわけではありません。ここでは、あくまで、「このクラスからインスタンスが作られたときに操作のふるまいとしてどのようなことをするか」という処理の手順を定めたのみです。
いわばシナリオを書いたのみであり、実際の配役がされ、動き出すのはインスタンスができてからです。



[Happy Squeaking!!]

5. Squeak 演習：クラスを作成する

5. 2 銀行口座インスタンスの生成、実行

それではワークスペースで、今作られた銀行口座クラスからインスタンスを生成して、メッセージを送ってみましょう。

```
| myAccount |
myAccount := Account new.
myAccount initialize.
myAccount deposit: 10000.
myAccount inspect.
```

2行目で、Accountクラスにnewメッセージを送りAccountインスタンスを生成させています。作られたインスタンスは、myAcccountという変数に代入されます。

以後はmyAccount変数に代入されたAccountインスタンスに対してメッセージが送信されていきます。

最初につくられた段階では、myAccountは

myAccount	
id	nil
money	nil

になっています。
Initializeのメッセージが送られると、インスタンスは、自分のクラスで定義されたinisitlizeの動作を行います。

initilizeは定義によるとメソッドが

```
initialize
        id := 0.
        money := 0.
```

なので、起動後は、myAccountは

myAccount	
id	0
money	0

になります。

次はdeposit: 10000 のメッセージ送信になります。
Accountクラスのメソッド定義に従い、Accountインスタンスは処理を行います。

```
deposit: newMoney
        money := money + newMoney.
```

のnewMoneyの部分に10000が入り起動されることになります。結果moneyの値が増えます。

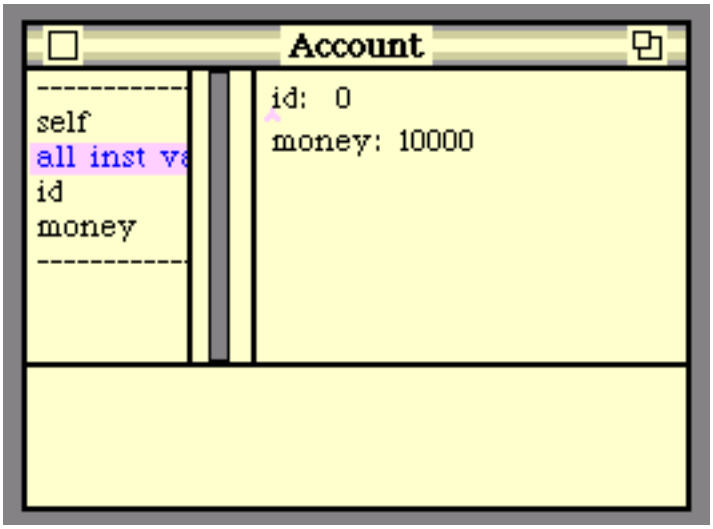
起動後の、myAccountは

myAccount	
id	0
money	10000

となります。こうしたインスタンス内部の処理は、カプセル化されており、外部からは見ることはできません。外部からは単に「なになにして」とメッセージを送るだけです。

今回作成した銀行口座ではメソッドの処理が非常に単純なのでそれほどのメリットは感じられないかもしれませんが、より複雑な処理を行うような場合には、この「内部の複雑な処理が見えない」というのは、呼び出し側にとっては煩雑で無関係な部分を気にせずにすむので利点がでてきます。何か少しの指示を出せば、オブジェクトが複雑な仕事をこなしてくれる、いわば「よきにはからえ」状態というわけです。

最後に、inspectメッセージが送られます。このメッセージによりインスペクタが開き、中身が確認できます。



実行後のAccountインスタンスの状態

次はインスペクタでなくトランスクリプトに残高の値を表示してみましょう。
文法の復習もかねてカスケードも使ってみます。Transcriptを開き、以下を実行しま

しょう。

```
| myAccount |
myAccount := Account new.
myAccount initialize;
            deposit: 10000;
            deposit: 2000;
            deposit: 300.
Transcript show: 'my account balance is ';
                 show: myAccount getBalance printString.
```

セミコロンとピリオドの違いに注意しましょう。
また、最後のprintStringは、myAccount getBalanceの結果が数字オブジェクトなので、それを文字列に変換するために必要です。Transcript のshow: では文字列オブジェクトしか受け付けないのです。
メッセージの優先順位を思い出してください。myAccount getBalance printString のあとでTrnascript show: の実行になります。

SmallTip: 数字を文字列オブジェクトに変換するにはprintStringメッセージを使う

更につづく

Prev. Index Next

オブジェクトの広場

[Happy Squeaking!!]

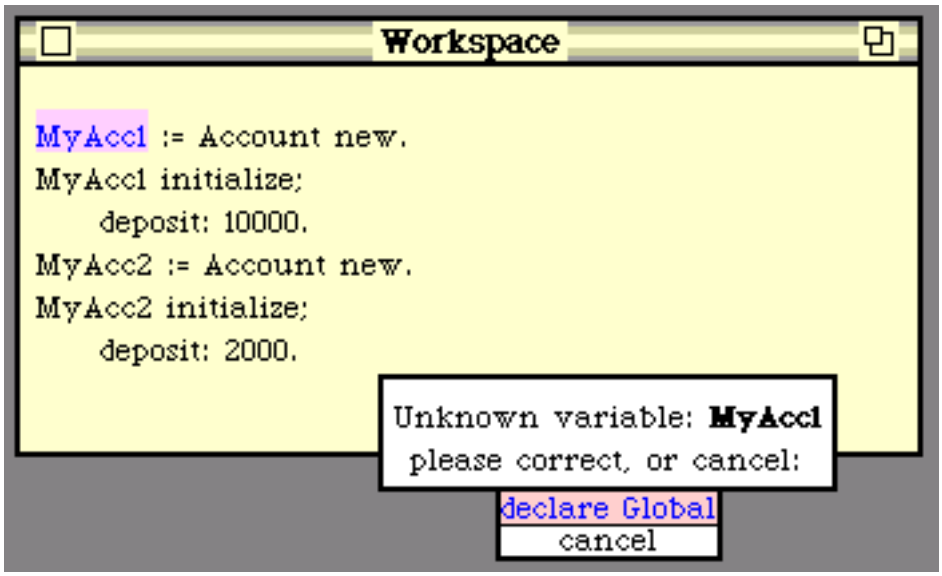
5. Squeak 演習：クラスを作成する

5. 2 銀行口座インスタンスの生成、実行 つづき

クラスはインスタンスの工場ですので、インスタンスを幾つでも生成することができます。例として、以下のコードを実行("do it")してみましょう。

```
MyAcc1 := Account new.
MyAcc1 initialize;
        deposit: 10000.
MyAcc2 := Account new.
MyAcc2 initialize;
        deposit: 2000.
```

Accountクラスから二つのAccountインスタンスを生成しています。
今回はいつものようにテンポラリ変数として変数を定義していません。従って"do it"の途中で、「グローバル変数にするか」と聞いてきます。「する」と答えて続けましょう。



グローバル変数の定義

グローバル変数は、テンポラリ変数と違い、実行後にすぐ消えさってしまうのではなく、Smalltalk(Squeak)のイメージファイルの環境が存在しつづける限り、その値が保たれます。Smalltalkでは、グローバル変数は通常クラスを格納するために使われます。また、まれによく使うインスタンスがグローバル変数に入っていることもあります。先ほどから使われているTranscriptがその例です。今回はMyAcc1とMyAcc2の属性の値を幾つかの実行に分けて観察したいのであえてこグローバル変数を使う方法をとっていますが、モジュール化の観点から多用は避けるべきです。

Smalltalkでは、グローバル変数は、通常の変数との区別をつけるために大文字始まりになる

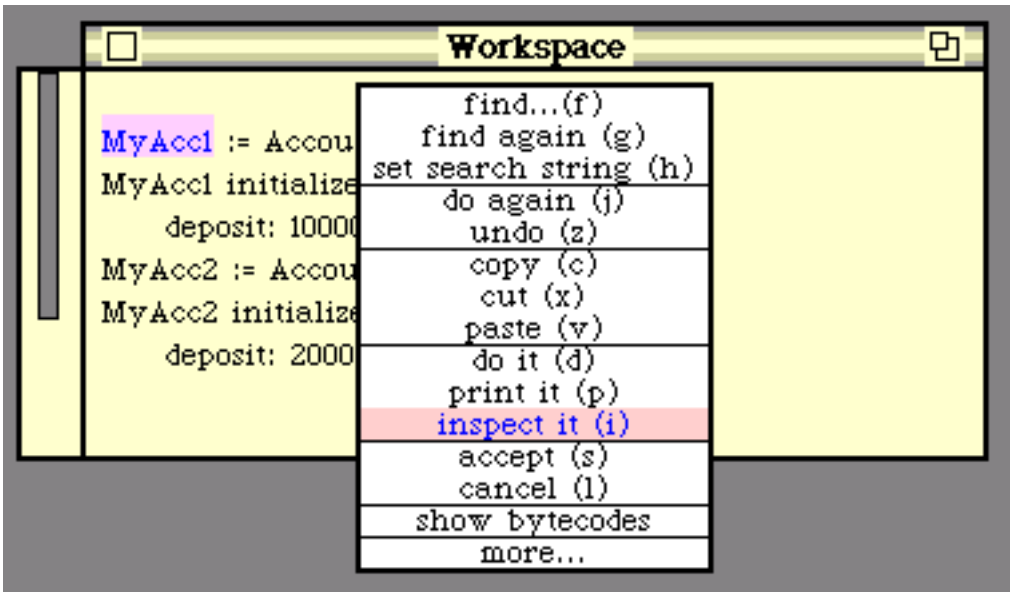
のが習慣です。

SmallTip: グローバル変数は大文字で始まる

インスタンスは個々の実体として存在するのでそれぞれの属性値は違いに影響をうけることはありません。

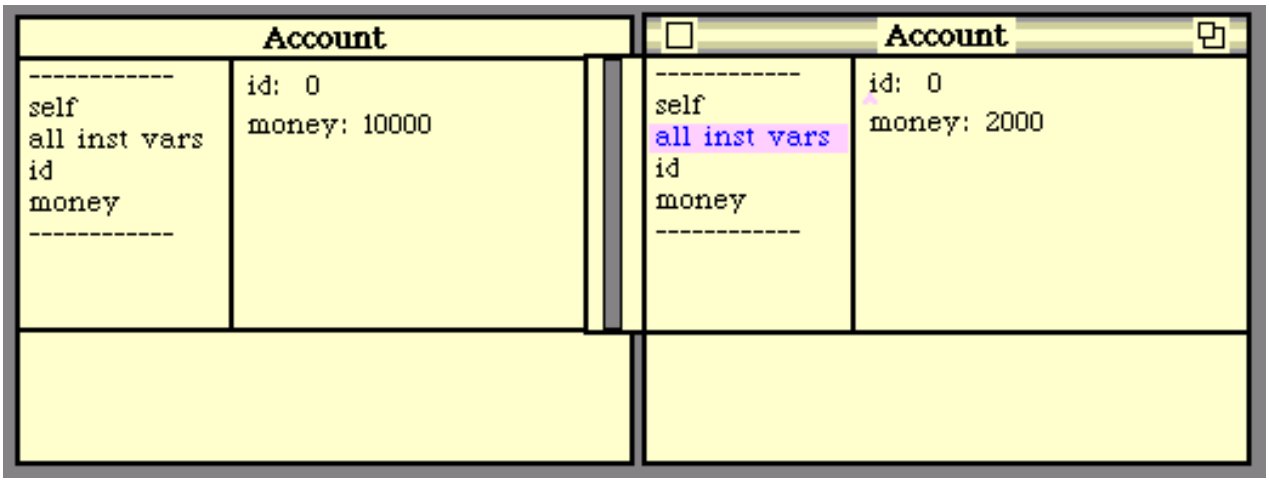
MyAcc1、MyAcc2をそれぞれインスペクトしてみましょう

グローバル変数なので、直接選択してインスペクトできます。ポップアップメニューで"inspect it "を選択します。



グローバル変数を直接inspect

MyAcc1、MyAcc2の両方のインスペクタを開くと、互いが別個の属性値を持っていることが確認できます。



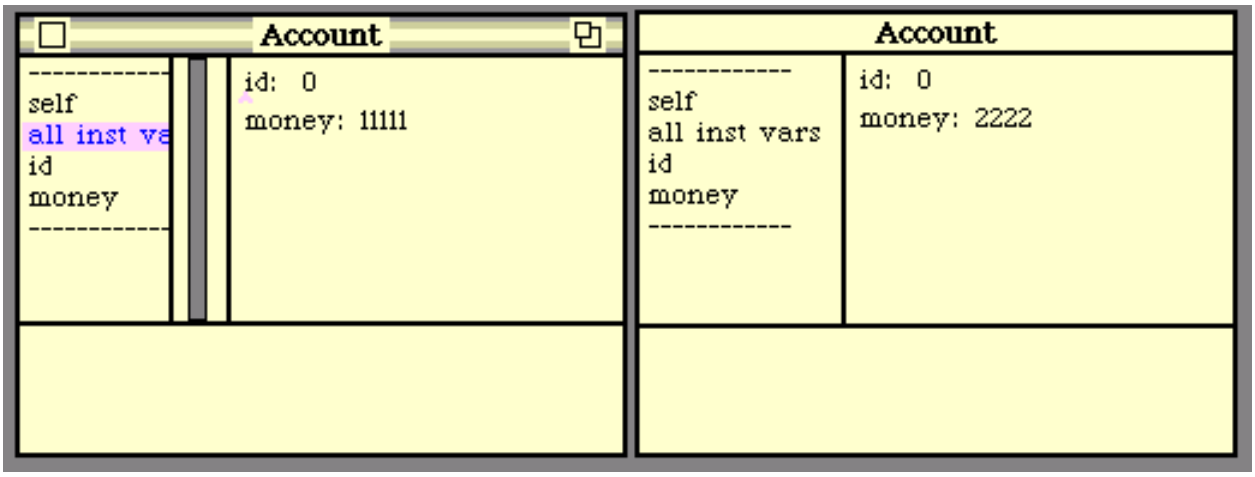
2つのAccountインスタンス

続いて、MyAcc1、MyAcc2にdeposit:メッセージを送ります。

MyAcc1 deposit: 1111.
MyAcc2 deposit: 222.

インスペクタを再びみると値が書き換わることが確認できます。(インスペクタの表示を

アップデートさせるため、リストの選択をし直してください。)



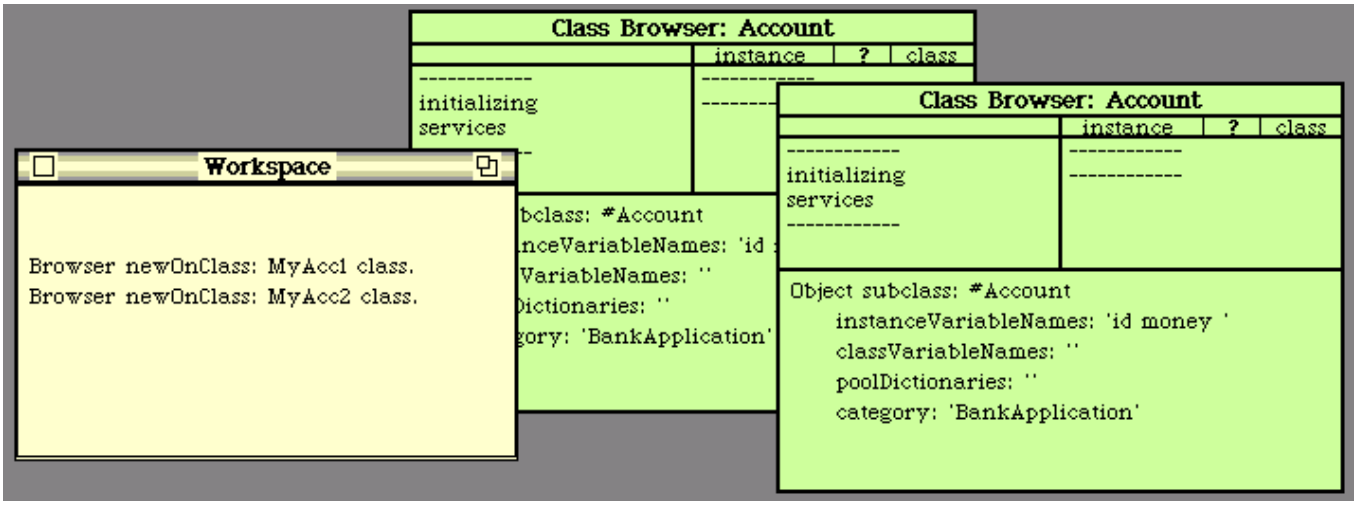
値の書き換わった2つのAccountインスタンス

一方でどのような属性を持つかという定義、および操作に対応した動作の仕方(メソッド)はクラスに記述され共有されていることになります。

両方のインスタンスからクラスをたどり、ブラウズしてみることにしましょう。

以下を一文づつ順番に実行します。
Browser newOnClass: MyAcc1 class.
Browser newOnClass: MyAcc2 class.

MyAcc1、MyAcc2の別個のインスタンスに対して、それぞれにclassメッセージを送って「あなたのクラスは?」と尋ねています。結果としてブラウザは、同じAccountクラスを引数として受け取って表示します。



Accountインスタンスは同じ定義情報から生成されている

SmallTip: インスタンスに対して、生成したクラスを問い合わせるには"class"メッセージを用いる

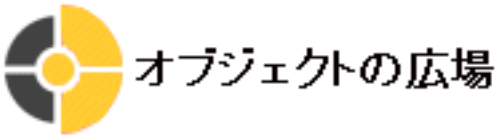
クラスとインスタンスについて大体イメージが湧きましたでしょうか。
余裕のあるかたは以下の課題にチャレンジしてみてください。

課題1: Accountクラスに"引き出し"、"id設定"の操作を追加する。

###

さて、盛り上がってきたところですが紙面の都合で今回はここまでです。
次回は継承とポリモルフィズムの解説に入ります。お楽しみに。

Prev. Index Next



［Happy Squeaking!!］

6. 参考文献

●洋書

"Handbook of Programming Languages Vol.1 Object-Oriented Programming Languages" Macmillan Technology Publishing 1998

Part II のSmalltalk編に、Adele GoldbergによるSmalltalkの歴史、 Allen Wirfs-Brocksによる文法解説がある。文法解説は非常に簡潔で一見の価値あり。

Let's "do it"!!


Prev. Index