Clément Béra ~ Smalltalk, Tips 'n Tricks

Mind twister: Toying with Object Kernels

17 - *Wednesday* - *Feb 2016*

Posted by Clement Bera in Cog, Pharo

≈ 1 Comment

Hi everyone,

Today we are going to toy with object kernels in Pharo :-). This is one thing Smalltalks using VMs inheriting from the Deutsch-Schiffman line as Cog are very good at, so I don't see why we shouldn't do it.

Toying with class-oriented system

Let's create a behavior class. As mentioned in the definition of Behavior:

> *" Behavior:*
> *the way in which one acts or conducts oneself, especially toward others."*

Hence a Behavior describes the way the objects interacts with other objects. That means a behavior describes the messages the object can understand or not and how they answer to it. As we will toy with a class-oriented object kernel for now, it also implies the lookup. Let's write some code:

```
Object subclass: #KHBehavior
instanceVariableNames: 'superBehavior methodDictionary'
classVariableNames: ''
package: 'Kernel-Hacking'
```

This `KHBehavior` class is concrete. Its instances can be put anywhere in a class hierarchy to specify messages understood by an object.

I let you generate the accessors…

Let's now try to use it. We will build a toy class somewhere in the system to mess around.

```
Object subclass: #Lion
instanceVariableNames: ''
classVariableNames: ''
package: 'DontOpenThatPackageEver'
```

Ok ! We have a `Lion` class inheriting from `Object`. Close the class browser opened on this class and never open it again. I'm not kidding, we are going to mess with that class far beyond IDE support, so close it before Morphic's red squares start popping around.

Let's now open a `Workspace` (open a playground and execute `Workspace open`). You can consider using the playground directly, but due to the usage of non-IDE compatible features, you may encounter problems.

Let's run this code:

```
Lion new isAnimal.
```

As expected, it raises a `DoesNotUnderstand` error. One can now close the debugger window.

We are going to inject a `KHBehavior` in the lookup hierarchy of `Lion` to add behavior from our defined structures.

Let's first create a methodDict holding the `isAnimal` method. As the OpalCompiler can't compile a method without precising a class in which it is compiled, we use `Object` as the class even if it does not any sense.

```
methodDict := MethodDictionary new
at: #isAnimal put: (Object compiler compile: 'isAnimal ^ true');
yourself.
```

We can now create our user-defined behavior. For convenience we inherit from `Object`. Inheriting from `nil` or `ProtoObject` could work too. I don't do it because I am scared the IDE will be way too confused and objects not defining `doesNotUnderstand:` are difficult to work with as the excepted behavior when they don't understand a message is to freeze the image, crash the VM or undefined. In addition I think it makes sense that `Lion` instance still have the behavior defined in `Object`

```
animalBehavior := KHBehavior new
superBehavior: Object;
methodDictionary: methodDict;
yourself.
```

Let's now inject it in the hierarchy of `Lion`. The first instance variable of `Lion`, as any class, is the superclass / superBehavior where the lookup has to be performed if the class does not implement a given selector. We now change that instance variable to point to our defined behavior. We can't use regular APIs such as `superclass:` because they are many guards in the system for unaware developers. So as we know this is the first instance variable, we will use the reflective API `instVarAt:put:` directly. Let's do this !

```
Lion instVarAt: 1 put: animalBehavior.
```

As we have hacked the class hierarchy, we need to flush the VM lookup internal caches. The easiest way is to flush everything with primitive 89. Such code can do it:

```
Object flushCache
```

We can now run:

```
Lion new isAnimal.
```

and it answers `true` ! Because the runtime has found the method in our animal behavior.

We can also try:

```
Lion new hash.
```

and it answers correctly the lion's hash looking up from `Lion` to `Object` through our animalBehavior.

Ok ! We have successfully designed an object that describes the behavior of objects and use it (at full performance) on top of our runtime.

There is no similar objects to the one we created by default in Pharo currently. I guess it's a design decision not to allow programmers to do that by default. I discussed about this with Jean-Baptiste Arnaud recently and he said that another Smalltalk, Bee Smalltalk, has similar objects.

Ok, let's stop discussing and let's now subclass it to add the ability to instantiate objects :-).

```
KHBehavior subclass: #KHInstanciator
instanceVariableNames: 'instanceFormat'
classVariableNames: ''
package: 'Kernel-Hacking'
```

`KHInstanciator`, in addition to specifying the behavior of objects, can instantiate them ! The `instanceFormat` instance variable is a `SmallInteger` that encodes the way the primitive `new` should instantiate objects. For example, on top of Spur runtimes, the bottom bits of the instanceFormat are mapped as follow:

- bit 1 to 16, 16 bits field, describes the instance size, i.e., the number of fixed fields the object has. If one subclasses `KHInstanciator` to make regular classes, the fixed fields would correspond to named instance variables. But `KHInstanciator` does not have any mapping between names and fields, so in its opinion, the fixed fields are just fields you can access with a specific bytecode.
- bit 17 to 21, 5 bits field, describes instance specification. There are different specification supported by the VM, such as byte objects (the object's fields are bytes, an example is ByteArray), variable sized pointer object (the object have a variable number of field based on the primitive `new:`) or fixed-sized objects such as `Point`.

Let's add the primitive `new` in `KHInstantiator` to make it able to instantiate.

```
KHInstanciator>>new
< primitive: 70 >
```

Let's now try to use it. For our first `KHInstantiator`, we will create an instanciator with no fixed sized fields. In order to avoid encoding by hand the instance format, we will reuse the one of existing classes.

In a similar way to the `KHBehavior`, we will create a method dictionary. The `KHInstanciator` created will inherit from the behavior we defined before for fun :-).

```
methodDict2 := MethodDictionary new
at: #isMonkey put: (Object compiler compile: 'isMonkey ^ true');
yourself.
monkey := KHInstanciator new
superBehavior: animalBehavior;
methodDictionary: methodDict2;
instanceFormat: UndefinedObject format;
yourself.
```

Let's instantiate and toy !

```
aMonkey := monkey new
```

`aMonkey isMonkey` effectively answers true. We have succeeded !

What about fixed fields ?

Well let's try. We can redefined our monkey instanciator using `KHBehavior format` as instance format to specify a fixed-sized object with 2 instance variables. Instances of monkey will then have 2 fixed-sized slots as instances of `KHBehavior`. We can try this way:

```
monkey := KHInstanciator new
superBehavior: animalBehavior;
methodDictionary: methodDict2;
instanceFormat: KHBehavior format;
yourself.
```

The problem is, how to access the two fixed fields ? There are different ways of doing it.

One way uses the reflection API `instVarAt:` and `instVarAt:put:`.

```
monkey methodDictionary
at: #field1 put: (Object compiler compile: 'field1 ^ self instVarAt: 1');
at: #field1: put: (Object compiler compile: 'field1: value ^ self instVarAt: 1
put: value')
```

As we can see it works fine: `aMonkey field1: #KingKong. aMonkey field1` answers #KingKong.

Another way is to use the bytecodes / quick primitives allowing unchecked access to fixed fields. Obviously one can't use the compiler directly with the structures we manually defined (they have no notion of instance variables of things like that), so one needs to build a compiler based on his use case that generate correct field access bytecodes or hack the whole thing by installing methods from other class in the method dictionary. We will do the latter.

```
monkey methodDictionary
at: #field2 put: MessageSend>>#selector;
at: #field2: put: MessageSend>>#selector:
```

And, as expected, `aMonkey field2: #Banane. aMonkey field2` answers #Banane.

We could also change the instance specification to create variable sized objects or other kind of objects, add the primitive `new:` in `KHBehavior`, and use `at:` and `at:put:` primitives on the instances, but I'm too lazy to do so. Do it yourself.

By the way, all what we did until now works **at full performance** on the Cog VM, i.e., as fast as if one would run Smalltalk code.

Toying with prototype-oriented system

Now let's try to build a simple Prototype object kernel. Inspired on Javascript, we will do prototypes that have:

- properties, basically anything the object has (state, behavior, maybe more).
- a prototype, to which they delegate property access when they can't handle it to be able to use inheritance.

First prototype model

In this model, the prototype will use compiled methods to execute code.

To build the first prototype, we will use a class from the Pharo Object model.

```
Object subclass: #PrototypeBuilder
instanceVariableNames: 'prototype properties'
classVariableNames: ''
package: 'Banane'
```

We will add in that object a few methods, the prototype will reuse methods from this method dictionary as it is not obvious to compile methods in the prototypes (we would need to build a compiler).

Prototypes will be initialized to delegate property access to `Object` by default. This is not mandatory but it allows use to have errors instead of VM crashes when the program or the IDE does something wrong or unsupported by the IDE with the prototypes.

```
PrototypeBuilder>>initialize
prototype := Object.
properties := self class methodDict copy.
```

We then add a convenient printing method not to be confused while displaying prototypes:

```
PrototypeBuilder>>printOn: s
s << 'prototype '.
self hash printOn: s
```

Lastly, we add the code that generate the first prototype:

```
PrototypeBuilder class>>newPrototype
| copy proto |
copy := self copy.
proto := copy new.
copy becomeForward: proto.
^ proto
```

We use `becomeForward:` instead of `primitiveChangeClassTo:` as it is much more flexible.

So we can now write `PrototypeBuilder newPrototype` which would be the equivalent of `{ }` in Javascript.

Let's make a simple example:

```
prototype := PrototypeBuilder newPrototype .
prototype class == prototype
```

It answers true, everything is fine.

Let's now add code to access easily the properties.

```
PrototypeBuilder>>propertyAt: name put: value
properties at: name put: value

PrototypeBuilder>>propertyAt: name
properties at: name ifPresent: [ :prop | ^ prop ].
^ prototype == Object
ifTrue: [ 'undefined' ]
ifFalse: [ prototype propertyAt: name ]
```

Any object can now access its properties. Let's try:

```
prototype := PrototypeBuilder newPrototype .
prototype propertyAt: #state1 put: 'a prototype !'.
prototype propertyAt: #state1.
```

The code correctly answers 'a prototype !' !. Let's add behavior instead of plain state to the object:

```
prototype propertyAt: #foo put: (Object compiler compile: 'foo ^ #foo').
prototype foo.
```

The code correctly answers #foo !

So, what if we try to run another property than a compiled method ?
`prototype state1.`

Well, we end up in an error. The runtime attempts to run the String as a method, but it does not work. In Javascript we would have something like 'TypeError: property is not a function'. The error message is different but that's the excepted behavior.

Another difference is that in Javascript accessors are generated automatically for each state. One could edit the code of `propertyAt:put:`. I'm too lazy to do so, and it is strange to do it because the Smalltalk syntax is ambiguous with the given model (`prototype propertyAt: #state1` should answer the getter or the state itself ?.

ok, up to now we made objects (i.e., state and behavior) without any inheritance.

So, what about inheritance ? Well, as in Javascript, you can change the prototype of an object to share properties. Properties are looked up along the prototype chain and answer 'undefined' if nothing is found.

Let's generate the accessor for `prototype` in `PrototypeBuilder` and create a few prototypes to mess around.

```
prototype1 := PrototypeBuilder newPrototype .
prototype2 := PrototypeBuilder newPrototype .
prototype1 propertyAt: #foo1 put: (Object compiler compile: 'foo ^ #foo1').
prototype1 propertyAt: #state1 put: 'state1'.
prototype2 propertyAt: #foo2 put: (Object compiler compile: 'foo ^ #foo2').
prototype2 propertyAt: #state2 put: 'state1'.
prototype2 prototype: prototype1.
```

Ok, we have a prototype1 with two properties, and prototype2 with two new properties inheriting from prototype1.

Let's check a few things:

```
prototype2 foo2. "Answers #foo2"
prototype2 foo1. "Answers #foo1"
prototype2 propertyAt: #state2. "Answers #state2"
prototype2 propertyAt: #state1. "Answers #state1"
```

```
prototype1 foo2. "Does not understand"
prototype1 foo1. "Answers #foo1"
prototype1 propertyAt: #state2. "Answers 'undefined'"
prototype1 propertyAt: #state1. "Answers #state1"
```

The model looks all right and quite similar to Javascript. If we change the prototypes later, we now have:

```
prototype1 propertyAt: #secondFoo1 put: (Object compiler compile: 'foo ^
#secondFoo1').
prototype2 propertyAt: #secondFoo2 put: (Object compiler compile: 'foo ^
#secondFoo2').

prototype1 secondFoo1. "Answer #secondFoo1"
prototype1 secondFoo2. "Does not understand"
prototype2 secondFoo1. "Answer #secondFoo1"
prototype2 secondFoo2. "Answer #secondFoo2"
```

That's it !

second prototype model

The first model was fairly fast (it reused the existing runtime to do lookups, hence the VM lookup caches), though it was still far from a usable model. This second model is dog slow but much closer to Javascript.

The problem is that in javascript functions can be handed around and can encapsulate the outer environment like closures. In this second model, we will use closures to execute code for this purpose. Let's do this (use a new `PrototypeBuilder` class):

```
PrototypeBuilder2>>printOn: s
s nextPutAll: 'prototype '.
self hash printOn: s

PrototypeBuilder2 class>>newPrototype
| copy proto |
copy := self copy.
proto := copy new.
copy becomeForward: proto.
^ proto

PrototypeBuilder2>>propertyAt: name put: value
properties at: name put: value
```

```
PrototypeBuilder2>>propertyAt: name
properties at: name ifPresent: [ :prop | ^ prop ].
^ prototype == Object
ifTrue: [ 'undefined' ]
ifFalse: [ prototype propertyAt: name ]

PrototypeBuilder2>>doesNotUnderstand: message
^ message numArgs = 0
ifTrue: [ self propertyAt: ('__',message selector) asSymbol ]
ifFalse: [ self
propertyAt: ('__',message selector allButLast) asSymbol
put: message arguments first ]

"example"
prototype := PrototypeBuilder2 newPrototype.
prototype foo: [ #foo ].
prototype foo. "Answers the closure"
prototype foo value. "Executes the closure".

"example with arguments"
prototype bar: [ :x | x + 1 ].
prototype bar value: 5.
```

Each behavior defined (each closure) can encapsulate things from where they were defined. For example:

```
| temp |
temp := 0.
prototype foo: [ temp := temp + 1 ].
```

Then in another Do It, you can have:

```
prototype foo value
```

which increases the value from the other environment and answers a new integer each time.

The `prototype` relationship can be used in a similar way to the first model.

However, this model does not use at all the available runtime, hooking mainly on `#doesNotUnderstand:`, which is dog slow.

What's next ?

Well it would be fun to build Mixins, a kernel similar to Mist with multiple inheritance but no overrides and other things. I love to toy with Objects as method (we had a glimpse in the article when attempting to run a String as a method). But this will be for another day.

... !

thought on “Mind twister: Toying with Object Kernels”

.. Pingback: Smalltalk en vrac (25) | L'Endormitoire

1