Ruby master - Feature #8564

Extend Module#attr... methods

06/24/2013 01:55 PM - Anonymous

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

Extend #attr_reader, #attr_writer, #attr_accessor syntax to accept default values, such as:

```
attr_reader foo: 42, bar: 43
```

Possibility of closures evaluated at initialization time might also be considered:

```
attr_reader baz: -> { Time.now }, quux: 42
```

History

#1 - 06/24/2013 02:10 PM - phluid61 (Matthew Kerwin)

+1 very useful pattern

#2 - 06/24/2013 02:13 PM - charliesome (Charlie Somerville)

-1 to overloading the same syntax with proc/lambda initialization.

#3 - 06/28/2013 07:33 AM - matz (Yukihiro Matsumoto)

We need to clarify how this intervene with #initialize. Any opinion?

Matz

#4 - 06/28/2013 08:44 AM - phluid61 (Matthew Kerwin)

matz (Yukihiro Matsumoto) wrote:

> We need to clarify how this intervene with #initialize. Any opinion?
>
> Matz

My suggestion is that it take effect before any explicit initialize method.

Examples:

```
class A
  attr_accessor foo: 3
  def initialize
    p @foo if defined? @foo
    @foo = 4
  end
end
p A.new.foo #=> prints 3, then 4

class B0
  attr_accessor foo: 3
end
class B1 < B0
  def initialize
    p @foo if defined? @foo
    @foo = 4
  end
end
p B1.new.foo #=> prints 3, then 4

class C0
  def initialize
    p @foo if defined? @foo
```

```
    @foo = 4
  end
end
class C1 < C0
  attr_accessor foo: 3
end
p C1.new.foo #=> prints 3, then 4
```

#5 - 02/05/2015 07:39 PM - TylerRick (Tyler Rick)

I would love to see this added to Ruby too, so that I don't have to repeat myself by **defining** attributes in one place and then **initializing** them later in initialize (as discussed in #5825 and #8563).

In the meantime, however, I've come across the fattr gem, which does exactly the same thing (plus a few extra things like defining a foo! method to *re-initialize*; I don't think attr_reader should do those extra things). (By the way, there was a great RubyTapas episode (subscription required) about fattr that got me excited about using it.)

Note that it's possible to support *both* initializing an attribute **to a Proc***and* using a block to initialize a attribute **to an expression** that is evaluated during object initialization.

Here's how fattr solves that problem... If you pass a block to fattr, it is evaluated during initialization:

```
class C
  fattr foo: 42
  fattr(:foo_2) { foo * 2 }  # evaluated during initialization
  fattr(:time) { Time.now }  # evaluated during initialization
end

p C.new.foo   #=> 42
p C.new.foo_2 #=> 84
p C.new.time  #=> 2015-02-05 ...
```

... whereas if you pass a block as the **value** for a hash key, it leaves it as a Proc and doesn't evaluate it:

```
class C
  fattr my_proc: ->{ 'my_proc' }
end
p C.new.my_proc #=> #<Proc:0x0...>
p C.new.my_proc.() #=> 'my_proc'
```

I can't think of any downside to supporting the block behavior for attr_accessor. It doesn't appear that the block syntax of attr_accessor is used for anything currently...

```
class C2
  attr_accessor(:foo_2) { foo * 2 }  # The block is never called
end

c = C2.new
p c.foo_2 #=> nil
```

Otherwise, you would be forced to add an initialize method any time you needed to initialize something to an expression that needed to be evaluated *at initialization time*, which would require duplication and mixing of initialization methods, which I think should be avoided:

```
class C
  fattr foo: 42
  attr_accessor :foo_2
  def initialize
    @foo_2 = foo * 2
  end
end

p C.new.foo   #=> 42
p C.new.foo_2 #=> 84
```