RubyIdioms (Ruby)

Ruby supports idiomatic programming. Please put your favorite Ruby idioms here. They don't have to be specific to Ruby.

Using idioms does not always lead to the most readable code, but they can provide convenient shortcuts in many situations.

I want this value, unless it evaluates to false in which case I want another value instead

Normally you would write something like

```
  if myvar
    return myvar
  else
    return another_value
  end
```

But you can use the `||` operator instead:

```
return myvar || another_value
```

You can also use the `||=` operator to assign a value to a variable if it evaluates to false. That is, instead of writing

```
myvar = another_value unless myvar
```

or

```
  unless myvar
    myvar = another_value
  end
```

you could write

```
myvar ||= another_value
```

handy to put anything or anthing else, if it doesn,t exist(I didn't know about that possibility for a long time):

```
puts page_title || "Untitled"
```

I want this value, unless it is nil in which case I want another value instead

Please note the difference between this idiom and the idiom above. In order to clearly see the different behavior assume myvar is `false`

```
myvar.nil? ? other : myvar
```

And setting myvar if it is nil (again not if it is false)

```
myvar = newvalue if myvar.nil?
```

while

```
myvar ||= newvalue
```

corresponds to

```
myvar = newvalue unless myvar
```

Invoke this method on my object, unless I have no object

If you want to invoke a method on an object in a variable, but in some cases the variable could be `nil`, you could do this:

```
if myvar
  return myvar.size
else
  return nil
end
```

But using the `&&` operator you can turn this into:

```
return myvar && myvar.size
```

If you want to return something other than `nil` if both `myvar` and `myvar.size` are `nil`, you can use the `||` operator again:

```
return myvar && myvar.size || 0
```

... or use the ternary operator:

```
return myvar ? myvar.size : 0
```

Run this code only when the file is the main program

Sometimes it is useful to add code to a file that is executed only if the file is run as the main program, but is ignored otherwise. Here's how:

```
if $PROGRAM_NAME == __FILE__
    do_stuff
end
```

Some applications of this idiom:

- Include test code at the bottom of a library file, so that running the file as a program tests the file.
- Require a program's file into a test module, so that its functions can be tested without running the whole program.
- Provide a Ruby API to a program's functionality.

Destroy this object when it goes out of scope

C++ programmers are used to using the "construction acquires, destruction releases" idiom to control the management of external resources such as files or sockets. They are often put off by Ruby because garbage collection and finalizers don't act like objects on the stack with destructors.

Ruby's blocks can be used to the same end. In the class that represents an external resource, define a class method that takes a block. That method

- allocates an instance of the class, thereby acquiring the resource
- passes the instance to the block, and then
- releases the resource when the block returns.

Here's an abstract example of this idiom in use:

```
Resource.open( identifier ) do |resource|
    process( resource )
end
# resource is now closed
```

The implementation of the class method must use a `begin...ensure` statement to ensure that the resource is always released:

```
def Resource.open( identifier )  # :yield: resource
    resource = Resource.new( identifier )
    yield resource
ensure
    resource.close
end
```

A practical example is the `open` method of the `File` class.

Often, though, the behaviour of new is conditional, depending on whether a block is provided. For instance, `File.open` behaves just like `File.new` *unless* you provide a block. This way, programmers can choose between the interfaces. Applying that to our general `Resource` class, we get this:

```
def Resource.open( identifier )  # :yield: resource
    resource = Resource.new( identifier )
    if block_given?
        begin
            yield resource
        ensure
            resource.close
        end
    else
        return resource
    end
end
```

Thus:

```
r = Resource.open( x )              # -> some Resource object

Resource.open( x ) do |r|
  # use 'r' ...
end
```

Hashes of lists

Sometimes, I want to fill out a hash with keys mapped to lists of arbitrary elements. Instead of the

cumbersome approach:

```
h[key] = [] unless h.has_key? key
h[key] << val
```

which requires three key lookups, you can turn it into

```
ary = (h[key] ||= [])
ary << val
```

or even shorter, but more unreadable:

```
(h[key] ||= []) << val
```

This code still takes two lookups, as `h[key] ||= []` is equivalent to `h[key] = h[key] || []`. An alternative is:

```
a = h.fetch(key) { h[key] = [] }
a << val
```

This involves one lookup in the common case, and two lookups when creating the sub-list.

You could also consider using the following:

```
h = Hash.new []
h[key] <<= val
```

but note that this doesn't quite work:

```
irb(main):001:0> h = Hash.new []
{}
irb(main):002:0> h[1] <<= 2; h[2] <<= 3
[2, 3]
irb(main):003:0> h
{1=>[2, 3], 2=>[2, 3]}
```

Only one array is ever created; we want a new array for every key. So instead we write:

```
h = Hash.new { [] }
h[key] <<= val
```

but this trick only works on Ruby 1.7 onward.

You can also use this form:

```
h = Hash.new{ |h,v| h[v]=[] }
h[key] << val
```

Jump to first non-false expression

Blame HalFulton for this (see [ruby-talk:58095]). His toy language at grad school included a `test` statement. It's similar enough to the `case` statement that ...

For example:

```
case true
when x < 0;             puts "x < 0"
when 0 <= x && x < 1;   puts "x in interval [0,1)"
when 1 <= x && x < 2;   puts "x in interval [1,2)"
```

```
when 2 <= x;             puts "x >= 2"
end
```

In fact, you can even omit the "true" and have the same semantics:

```
case
when x < 0;              puts "x < 0"
when 0 <= x && x < 1;    puts "x in interval [0,1)"
when 1 <= x && x < 2;    puts "x in interval [1,2)"
when 2 <= x;             puts "x >= 2"
end
```

Further, you can take advantage of Ruby's functional nature:

```
puts case
when x < 0;             "x < 0"
when 0 <= x && x < 1;   "x in interval [0,1)"
when 1 <= x && x < 2;   "x in interval [1,2)"
when 2 <= x;            "x >= 2"
end
```

easy to access elements in array

```
val = [1, 2, 3]
a, b, c = val      # Observe this particular line
p a    #=> 1
p b    #=> 2
p c    #=> 3
```

This can be useful when you want a method to return multiple values.

```
def test
   [42, "ruby"]
end
val, str = test
```

Or, if you want just the second value:

```
val = test[1]
```

from my_hash['foo'] to my_hash.foo

If you want to reference a value corresponding to a key in hashes you may use a *dotted* notation by several means (take a look at [ruby-talk:92897] too).

- Hash could be extended to act like a Struct

```
class Hash
  def method_missing(meth,*args)
    if /=$/=~(meth=meth.id2name) then
      self[meth[0...-1]] = (args.length<2 ? args[0] : args)
    else
      self[meth]
    end
  end
end
x = { 'name'=>'Gavin', 'age'=>31 }
x.weight = 171
x.feet = ['left','right']
puts x.name , x.weight , x.foo , x.inspect
```

```
#=>Gavin
#=>171
#=>nil
#=>{"name"=>"Gavin", "weight"=>171, "feet"=>["left", "right"], "age"=>31}
```

- You may get a Struct from a Hash using [Dan Berger's code snippet on RubyForge] *Is this worthwhile, given that OpenStruct is in the standard library?*

```
class Hash
   def to_struct(struct_name)
      Struct.new(struct_name,*keys).new(*values)
   end
end
h = {:name=>"Dan"}
s = h.to_struct("Foo")
puts "name: " + s.name
```

- OpenStruct merges Hash and Struct behaviours together. It is like a hash, but it uses the `obj.key` notation.

```
require 'ostruct'
x = OpenStruct.new('name' => 'Gavin', 'age' => 31)
x.weight = 171
x.feet = ['left', 'right']

puts x.name       # -> "Gavin"
puts x.weight     # -> 171
puts x.foo        # -> nil
puts x.feet       # -> ['left', 'right']
```

Automagical Variable Instantiation with a Hash

If you create a hash with a default value, you can get automagical instantiation for that type. For instance:

```
mlock = Hash.new {|h,k| h[k] = Mutex.new }
```

then later on throughout my code when I need a mutex I say

```
mlock["entry"].synchronize do
  # blah
end
```

and then later on when I need another I use

```
mlock["exit"].synchronize do
  # exit blah
end
```

This prevents me from needing to declare a new mutex in `initialize` every time I need another mutex in that class.

-- CharlesComstock

Simulating Java's synchronized keyword

First thing we do, is add the synchronized method to object:

```
class Object
```

```
  def synchronized(obj)
     @_class_locker ||= Hash.new {|h,k| h[k] = Mutex.new }
     @_class_locker[obj.id].synchronize { yield }
  end
end
```

Once we have this, we can use synchronized(variable_name), just like you can in java!

Example:

```
require 'thread'

mytest = "hello world"

thread1 = Thread.new do
  # java like syntax
  synchronized(mytest) {
    puts "1: mytest: #{mytest}"
    sleep 1
    mytest = 'goodbye world'
  }
end

thread2 = Thread.new do
  # ruby syntax
  synchronized mytest do
    puts "2: mytest: #{mytest}"
  end
end

thread1.join
thread2.join
```

Results in:

```
1: mytest: hello world
2: mytest: goodbye world
```

Or, for a more ruby'ish example that does not leave any dependencies on the Object class:

```
require 'thread'
module JavaEmulation
  def synchronized(on)
     @_class_locker ||= Hash.new {|h,k| h[k] = Mutex.new }
     @_class_locker[on.id].synchronize { yield }
  end
end

class MyClass
  include JavaEmulation
  def start_threads
    mytest = "hello world"

    thread1 = Thread.new do
      synchronized(mytest) {
        puts "1: mytest: #{mytest}"
        sleep 1
        mytest = 'goodbye world'
      }
    end

    thread2 = Thread.new do
```

```
      synchronized mytest do
        puts "2: mytest: #{mytest}"
      end
    end

    thread1.join
    thread2.join
  end
end

MyClass.new.start_threads
```

Initializing complex objects

If an object has a lot of data that needs to be provided on initialization, the traditional method call arrangement is unsuitable, because you need to remember the order of arguments, and you can't omit arguments that you don't care about. Ruby offers some relief with variable argument lists (`def foo(x, y, *args)`) and default values (`def foo(x, y=5, *args)`, but for seriously complicated objects, these are typically not enough.

Passing a hash is one option, using keys of the hash to name the attributes you wish to set. This allows arbitrary ordering, and selective omission (to allow default values). See KeywordArguments for a look at this technique.

But I prefer the following:

```
  complex_object = ComplexObject.new do |c|
    c.width = 15
    c.height = 20
    c.colour = :red
    c.position = [12, 4]
  end
```

The benefits:

- it's easy to code for (see below)
- the syntax is arguably more natural than the hash approach (it's very Rubyesque)
- it's self-documenting; you can look at the documentation for the `ComplexObject` class to see which attributes you can set

The disadvantages:

- all attributes used must be public

-Hey, just create a Struct to be passed to the block. Then this disadvantage disappears.

The implementation:

```
  class ComplexObject
    attr_accessor :colour, :position, :width, :height, :depth, :label

      #
      # Width and height _must_ be specified; all others are optional.
      #
    def initialize  # :yield: c
        # Set default values.
      @colour = :white
      @position = [0,0]
      @depth = 0
```

```
      @label = ""
      yield self
        # If all attributes are optional, we would code:
        #   yield self if block_given?
      if @width.nil? or @height.nil?
        raise TypeError, "Incompletely specified ComplexObject"
      end
    end
  end
```

This, to me, is a very natural coding style. The default values are easily seen in the code (usually browsable via RDoc). The post-condition is easily specified. All in eight clear lines of code for a class with six attributes.

Morover, this style is extremely useful when creating hierarchies of composite objects. i.e.: FXRuby implements this idiom to support well structured GUI code.

```
FXMainWindow.new($app, ...){|mainwindow|
   FXHorizontalFrame.new(mainwindow){|hf|
      FXLabel.new(hf){|label| ... }
      FXButton.new(hf){|button| .. }
   }
}
```

As you can see, the hierarchic composition of the GUI is reflected by the structure of the code which increases readability and maintainability. I use this idiom whenever I need to build complex composite objects. --henon

Initializer alternatives

Ruby allows only one initializer method for a class, which is a problem if you want initialization to be done in different ways, or with different arguments.

One of the dynamic invocation features of Ruby can be used to relieve this, by writing a class in the following idiom:

```
  class Example

    def initialize( method_name, *parameters_array )
      send(method_name, parameters_array)
    end

  private
    def initialize1( parameters_array )
      # construct in one way
    end

    def initialize2( parameters_array )
      # construct in another way
    end

  end
```

where the arguments are collected into an array, and a name string is used to delegate to a particular method.

Such a class can then be instantiated variously by the client like this:

```
  e1 = Example.new( 'initialize1', 1, 2, 3 )
  e2 = Example.new( 'initialize2', 'a', 'b' )
```

which practically gives overloaded initializer methods.

(Ideally, it would be preferable to be able to instantiate like this:

```
e = Example.new_alternative1( 1, 2, 3 )
```

There is a hack for that:

```
class Example

  def initialize( method_name, *parameters_array )
    send(method_name, parameters_array)
  end

  def self.new_alternative1(*parameters_array)
    return Example.new('initialize1', parameters_array)
  end

private
  def initialize1( parameters_array )
    # construct in one way
  end

  def initialize2( parameters_array )
    # construct in another way
  end

end
```

It would work better through a mixin that metaclass-hacks to allow for easily flagging a method "initializer", though.)

Wonder of the When-be-splat

On [redhanded] _Why pointed out that * (the splat) can be used to unpack an array for a branch of a case statement.

```
BOARD_MEMBERS = ['Jan', 'Julie', 'Archie', 'Stewick']
HISTORIANS = ['Braith', 'Dewey', 'Eduardo']

case name
when *BOARD_MEMBERS
 "You're on the board!  A congratulations is in order."
when *HISTORIANS
 "You are busy chronicling every deft play."
end
```
