Make a configurable module to be included in objects

The main idea is to use an object and a method that returns a module.

First (and probably bad way)

```
class Extension # or a module, it doesn't matter here..
  def self.set( name )
    Module.new do
      define_method "my_#{name}" do
        "hello #{name}"
      end
    end
  end
end

class Receiver
  include Extension.set( 'mary' )
  ...
end

r = Receiver.new
r.my_mary # -> "hello mary"
```

The problem is in the unnamed module :

```
Receiver.ancestors # [Receiver, #<Module:0x00007ef79abe7b28>, ... ]

Hey, there is a solution to unnamed module (Ruby 3.3 and further only !)
but this way is still to be avoided. The name have to not be a legal ruby
constant name ('builder', 'Builder(mary)' are ok but not 'Builder' or
'Builder::Mary')!
`Module.new { ... }.set_temporary_name( "builder" )
```

Second (leading to a named module)

Modules are classes that can be subclassed..

```
class Extension < Module
  def initialize( name = 'john' )
    super() # some say this is mandatory...
    @name = name
  end

  def included( klass )
    super # better if nesting inclusions...
    ## the key here is to use a ruby closure to hold instance variable
    # First way :
    -> ( name ) do
      define_method "my_#{name}" do
        "hello #{name}"
      end
    end.call( @name )
```

```
      # Second way
      [ @name ].each do |name|
        define_method "my_#{name}" do
          "hello #{name}"
        end
      end

      # Third way
      set_method( @name )
    end

    private

    def set_method( name )
      define_method "my_#{name}" do
        "hello #{name}"
      end
    end
  end

  class Receiver
    include Extension.new( 'mary' )
    ...
  end

  Receiver.ancestors # [Receiver, #<Extension:0x00007ef795adc4e0>, ... ]
```

Syntactic sugars

```
  class Extension < Module
    def self.[]( *args )
      new( *args )
    end

    def self.For( *args )
      new( *args )
    end
    # previous code
  end

  class Receiver
    # choose the one you like..
    include Extension::For( 'mary' ) # like Shrine::Attachment( :image )
    include Extension.For( 'mary' )
    include Extension[ 'mary' ]
  end
```

Further

When arguments are optionals

Then use a module wrapper so the inclusion is not wasted by empty `[]` or `.new` .

```
  module Buildable
    def self.included( klass )
      klass.include Builder.new
    end
    def self.[ *args ]
      Builder.new *args
    end
  end

  class Receiver
    include Buildable # will include Builder.new
    include Buildable[ 'mary' ] # will include Builder.new( 'mary' )
  end
```

A complete configuration

Why not provide a complete configuration feature :

```
  class Builder < Module
    attr_accessor :name

    def initialize
      super
      yield self
    end

    def included( klass )
      -> (name) do
        define_method "my_#{name}" do
          "hello #{name}"
        end
      end.call( name )
    end
  end

  class Receiver
    include( # without parenthesis, ambiguity leads to error
      Builder.new do |config|
        config.name = 'mary'
      end
    )
  end
  r = Receiver.new
  r.my_mary # -> "hello mary"
```

Hey ! This leads to a mutable configuration !!

A complete unmutable configuration (using Ustruct)

```
  class Builder < Module
    attr_reader :config

    def initialize( **options )
      super()
      @config = Ustruct.new( options )
    end

    def included( klass )
      -> (name) do # custom methods
        define_method "my_#{name}" do
          "hello #{name}"
        end
      end.call config.name
      -> (config) do # an attribute reader for config !
        define_method :config do
          config
        end
      end.call config
    end
  end

  class Receiver
    include Builder.new( name: 'mary', flouz: 'yellow' )
  end

  r = Receiver.new
  r.my_mary # -> 'hello mary'
  r.config # -> #<Ustruct:0x00007406a26d01d8 @content={:name=>"mary", :flouz=>"yellow"}>
```