Boris

NLoops – a partial Objects layer for NetLogo

NLoops ref:0.4b

brief

NetLogo & its programming language are simple to use but, none-the-less, offer a flexible platform for developing models of complex systems.

We develop various models, some small and some larger scale. With a few of our more complex models we would like to use OO features which are not available in the NetLogo programming language (NetLogo is not object oriented).

To help with this (& also as an interesting exercise) we have developed a partial objects layer to use with NetLogo. It is written in NetLogo script. These notes describe this Objects layer.

We have made some compromises both in function and syntax. Some of these result from working within NetLogo script (rather than building this as a Java extension), some result from our aim to build a light-weight implementation. These notes are primarily intended as a user-guide for the objects system but we will highlight some of our main design decisions and compromises as we discuss the object features.

There are some features you will notice from the outset...

1. all object terms (variables, procedures, etc) start with a "#" symbol – this is because NetLogo operates a single name-space and we do not want to restrict your choice of normal variable names;
2. the style of declaration of classes and the way methods are associated with program code are not like Java or C++, etc;
3. declarations of classes, methods, etc are not specified at the top-level of code (the level where procedures are defined) but must be declared inside a procedure (it is good practice to use a dedicated *setup-nloops* procedure for this).

example 1

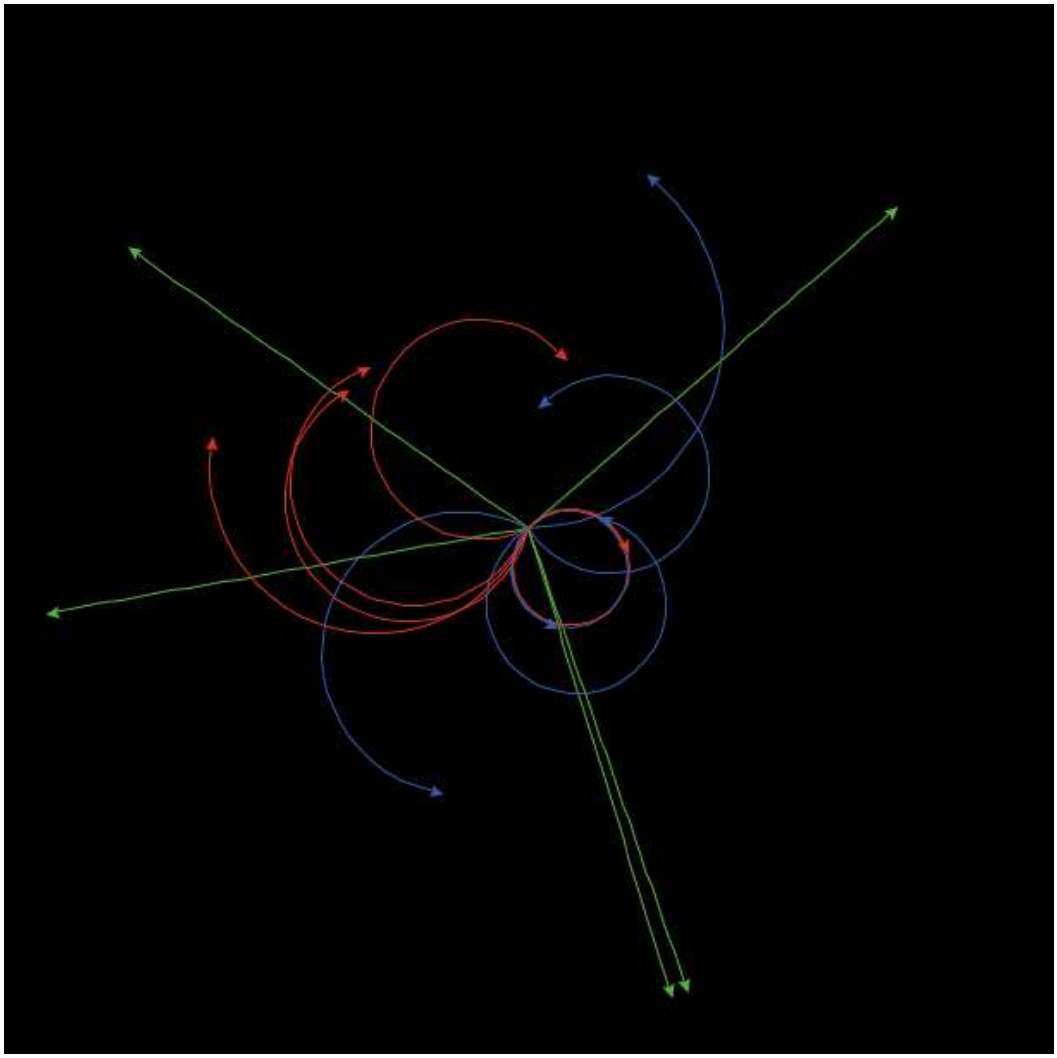
To examine some of the basic features of objects we consider an example (NLoops-eg1.nlogo) where 3 classes of turtle perform 3 different types of movement. Green turtles head in straight lines, read turtles circle right, blue turtles circle left (see output below).

You may reasonably decide that you can write this example without objects & methods and achieve the same results with less code & less fuss – this is true, we have done it too. We use this model because (we hope) it is fairly easy to understand and because we can use it to illustrate a range of object features.

We build our objects-based model by defining 4 different classes...

ant	– the super-class for all turtles that move
red-ant	– a sub-class of ant that tends to move right
blue-ant	– a sub-class of ant that tends to move left
green-ant	– a sub-class of ant that tends to move forwards

The different classes use different *methods* for moving, this is discussed more below.



setting up the model

First an includes statement loads an external .nls file containing the object definitions

```
__includes [ "NLoops.nls" ]     ;; load the objects layer
```

The "setup" button on the interface panel calls the setup procedure which then calls setup-classes & setup-turtles.

```
to setup                ;; called by an interface button
  clear-all
  setup-classes         ;; setup the class & method links
  setup-turtles         ;; setup some turtles
end
```

setup-classes is specified as below and explained in the text that followings.

```
to setup-classes
  #reset-objects
  #def-class "ant" [["step" 1]]

  #def-class "ant.red-ant"    ["bearing"]
  #def-class "ant.blue-ant"   ["bearing"]
  #def-class "ant.green-ant"  #nil

  #def-method "red-ant.go"
  #def-method "blue-ant.go"
  #def-method "ant.go"

  #def-method "ant.constructor"
end
```

#reset-objects

clear out any earlier object definitions and (if this is the first call of #reset-objects) initialise key structures.

#nil

a variable with a value [], #nil is used as an empty list value and is also used as a general null value.

#def-class

define a new class

syntax

`#def-class` *`super-class.class-name slots`*

class-name	–	the name of the class
super-class	–	the name of its super-class
slots	–	a list of the slots/variables which will be owned by all inheriting instances of this class (all slots are public)

example-1

```
#def-class "ant" [["step" 1]]
```

define a new class called “ant” with no super-class and one slot (called “step” with an initial value of 1)

example-2

```
#def-class "ant.red-ant" ["bearing"]
```

define a new class called “red-ant” with a super-class called “ant” and one slot (called “bearing”)

class slots and instance slots

all slots in the examples above are *instance slots*, that means that each instance of *ant*, for example, will have its own "step" variable. If you want variables to be owned by the class and for instances to inherit this as a shared variable use a "*class*" modifier, eg:

```
#def-class "ant" [["class.step" 1]]
```

#def-method

associate a new method with a class

syntax

```
#def-method class-name.method-name
```

class-name – the name of the class this method attaches to
method-name – the name of the method

example

```
#def-method "red-ant.go"
```

attach a method called "go" to the "red-ant" class . When instances of "red-ant" call their "go" method the NetLogo procedure called "red-ant.go" (defined later) will be used

The turtles are setup as follows...

```
to setup-turtles
  create-turtles 5
  [ set color red
    #spawn-instance "red-ant" #nil
    #=>"bearing" (5 + random 15)
  ]
  create-turtles 5
  [ set color blue
    #spawn-instance "blue-ant" #nil
    #=>"bearing" (5 + random 15)
  ]
  create-turtles 5
  [ set color green
    #spawn-instance "green-ant" #nil
  ]
end
```

#spawn-instance

make a new instance object for the current turtle

syntax
```
#spawn-instance class-name instance-slots
```

class-name – the name of the class this for this instance
instance-slots – a list of the slots which will be owned by this instance (or slots that over-ride other instance slots of this class)

example
```
#spawn-instance "red-ant" #nil
```

spawn an instance of "red-ant" for this turtle will no additional slots

setting instance slot values

syntax
```
#=> slot-name value
```

slot-name – the name of the slot to set
value – the new value for the slot

example
```
#=>"bearing" 15
```

set the "bearing" slot of the current turtle to 15

getting instance slot values

syntax
```
#<= slot-name
```

slot-name – the name of the slot to get a value from

example
```
#<="bearing"
```

get the value from the "bearing" slot of the current turtle

running the model

The "go" button on the interface panel calls the go procedure...

```
to go
  ask turtles
  [ pen-down
    #"go"                  ; ask turtles to run their "go" method
  ]
  tick
end
```

The statement #"go" asks turtles to run their "go" methods, these methods have been defined earlier in setup-classes where the statements...

```
    #def-method "red-ant.go"
    #def-method "blue-ant.go"
    #def-method "ant.go"
```

...associated the NetLogo procedure "red-ant.go" with the "go" method for *red-ants*, the "blue-ant.go" procedure with *blue-ants* and the "ant.go" procedure with *ants*. Note that *green-ants* have no specific "go" method so they will inherit theirs from the "ant" class.

In our first version of the model, the go-right, go-left & go-fwd procedures are defined as follows.

```
to red-ant.go
  right #<="bearing"        ; go right by value of "bearing" slot
  forward #<="step"         ; go forward by value of "step" slot
end

to blue-ant.go
  left #<="bearing"         ; go left by value of "bearing" slot
  forward #<="step"         ; go forward by value of "step" slot
end

to ant.go
  forward #<="step"         ; go forward by value of "step" slot
end
```

The go-right procedure does the following...

right #<="bearing"
forward #<="step"
turn right by the value of the "bearing" slot of the current turtle (#<="bearing" gets the value from the "bearing" slot) then move forward by the value of the "step" slot.

The second version of this model defines the red-ant.go and blue-ant.go procedures differently...

```
to red-ant.go
  right #<="bearing"        ; go right by value of "bearing" slot
  #"super.go"               ; call superclass method
end


to blue-ant.go
  left #<="bearing"         ; go left by value of "bearing" slot
  #"super.go"               ; call superclass method
end
```

#"super.go"

this calls the "go" method upwards in the super-class chain. The super-class for "red-ant" is "ant" so #super will call "ant.go" (the "go" method for the "ant" class) but still apply it to the current turtle.

runtime variables: #self & #my-class

these are variables that can be referred to inside methods. The variable #self always contains the name of the instance which the method was invoked on (this will be a turtle *who number* in our example. The #my-class variable contains the class name of #self.

So, if we modified the ant.go method above to include a print statement as follows...

```
to ant.go
  forward #<="step"        ; go forward by value of "step" slot
  print (word "self=" #self "\t class=" #my-class)
end
```

... we will see something like this in the output area...

```
self=2       class=red-ant
self=0       class=red-ant
self=1       class=red-ant
self=12      class=green-ant
self=8       class=blue-ant
self=11      class=green-ant
self=3       class=red-ant
self=13      class=green-ant
... etc ...
```

Note that the order of the turtles will change from move to move because NetLogo varies their ordering. Note also that you must be running a method to use #self.

constructor methods

Constructor methods are (like other languages) special methods which run when instances are created. Our second model in the series is modified to use constructor methods.

The first step is to specify a procedure to run when instances are created – in this case we write a constructor for the *ant* class. All instances of *ant* (or instances of sub-classes of *ant* that do not have their own constructors) will run this.

```
to ant.constructor
  if (#has-slot? "bearing")           ; if this class has a bearing slot
  [ #=>"bearing" (5 + random 15)      ; give it a value
  ]
end
```

Next we must add an appropriate #def-method statement to the *setup-classes* procedure. The method must be called "constructor"...

```
  #def-method "ant.constructor"
```

Finally we can remove the assignments to "bearing" slots in the *setup-turtles* procedure...

```
to setup-turtles
  create-turtles 5
  [ set color red
    #spawn-instance "red-ant" #nil
  ]
  ...etc...
```

checking on your objects

It is possible to check on the objects that have been declared, their slots and methods, by using *#show-objects*. Using *#show-objects* on the model we have been using as an example would print something like the following to the output area...

```
observer> #show-objects
[] ==> [[#islots []] [#type class]]

ant ==> [[#name ant] [#class []] [#type class] [#islots [[step 1]]]
  [go [[owner ant] [mname go] [pname go-fwd] [super []]]]
  [constructor [[owner ant] [mname constructor] [pname build-ant]
  [super []]]]]

red-ant ==> [[#name red-ant] [#class ant] [#type class]
  [#islots [[step 1] bearing]]
  [go [[owner red-ant] [mname go] [pname go-right] [super ant]]]]

blue-ant ==> [[#name blue-ant] [#class ant] [#type class]
  [#islots [[step 1] bearing]]
  [go [[owner blue-ant] [mname go] [pname go-left] [super ant]]]]

green-ant ==> [[#name green-ant] [#class ant] [#type class]
  [#islots [[step 1]]]]

2 ==> [[#name 2] [#class red-ant] [#type instance] [step 1] [bearing 14]]
3 ==> [[#name 3] [#class red-ant] [#type instance] [step 1] [bearing 10]]
1 ==> [[#name 1] [#class red-ant] [#type instance] [step 1] [bearing 5]]
0 ==> [[#name 0] [#class red-ant] [#type instance] [step 1] [bearing 8]]
4 ==> [[#name 4] [#class red-ant] [#type instance] [step 1] [bearing 16]]
9 ==> [[#name 9] [#class blue-ant] [#type instance] [step 1] [bearing 12]]
8 ==> [[#name 8] [#class blue-ant] [#type instance] [step 1] [bearing 10]]

...etc...
```

another type of method

Methods in most languages specialise on object types but there have been experiments with specialising methods on object attributes. Influenced by these aims we have provided an additional type of method. To illustrate the use of these methods we rebuild the last model again.

First, because we are going to specialise on attributes rather than object types, we simplify the classes we have & remove the unneeded declarations from *setup-classes*. Notice that we have defined a "run" method for ants, we have also included a slot called "cname" (short for color-name). Finally (to illustrate their use) we have defined "step" as a class slot.

```
to setup-classes
  #reset-objects
  #def-class "ant"  ["bearing" "cname" ["class.step" 1]]

  #def-method "ant.constructor"
  #def-method "ant.run"
end
```

Next we simplify setting up the turtles (notice we set the values for color slots)...

```
to setup-turtles
  create-turtles 5 [ #spawn-instance "ant" [["cname" "red"]  ] ]
  create-turtles 5 [ #spawn-instance "ant" [["cname" "blue"] ] ]
  create-turtles 5 [ #spawn-instance "ant" [["cname" "green"]] ]
end
```

#valof

The constructor for ant is similar to how it was in the earlier model but we also use it to set the NetLogo color variable for the turtles. This is a little contrived because we need to get the numeric value of the color name hence the use of #valof (this is primarily an issue with color not an issue with the methods we are going to build).

```
to ant.constructor
  #=>"bearing" (5 + random 15)
  set color (#valof #<="cname")
end
```

#valof takes the value of a variable who's name is held in a string, so `(#valof #<="cname")` uses #<= to get the value of the "cname" slot which will be "red", "green" or "blue". #valof then takes the value of one of these color names to get the color value. We can see how it operates in NetLogo's command centre window (note that 15 is NetLogo's representation of the color red)...

```
observer> print "red"
red
observer> print #valof "red"
15
```

Now we define three NetLogo procedures. The name of each procedure starts with the name of the "ant.run" method and the rest of the procedure name is taken from the value of the cname slot. The different words in procedure names (eg: "ant.run" and "red") are separated by hyphens.

Each of these procedures specifies the activity we want to occur for each of the different legal values of "cname" (ie: for each of the different color turtles).

```
to ant.run-red
  right #<="bearing"       ; go right by value of "bearing" slot
  forward #<="step"
end

to ant.run-blue
  left #<="bearing"        ; go left by value of "bearing" slot
  forward #<="step"
end

to ant.run-green
  forward #<="step"        ; go forward by value of "step" slot
end
```

The final step for this approach is to define a generic procedure to run the turtles which is triggered by a button on the interface.

```
to go
  ask turtles
  [ pen-down
    #do-modified "run" ["cname"] #nil
  ]
  tick
end
```

The code of interest here is...

```
    #do-modified "run" ["cname"] #nil
```

This does the following...
1. get the full name of the "run" method: "ant.run";
2. look up "cname" for the current instance ("red", "blue" or "green");
3. combine the values from 1 & 2 above into a new procedure name ("ant.run-red", "ant.run-blue" or "ant.run-green");
4. call the procedure with the derived name.

So... if a turtle has a cname="red" then "ant.run-red" will be called, if cname="blue" then "ant.run-blue" will be called.

#do-modified

syntax

```
#do-modified root-name modifiers arguments
```

root-name	–	this is the name of a pseudo method, it provides the first part of the procedure name which will take the rest of its name from the modifiers
modifiers	–	variable names who's values are used to form the rest of the procedure name
arguments	–	a list of arguments to supply to the resulting procedure

The code for this example is in the NLoops-eg-3 model. As we said before... this example can be rewritten without objects & methods to achieve the same results with less code but as models become more complex the use of objects can help to simplify your code.

arguments & reporters

In the examples above our methods have been fairly simple, they have not taken any arguments and have not returned/reported values. NetLogo allows us to define other procedural forms like *min-of-two* (below) which takes 2 arguments and returns a value...

```
to-report min-of-two [a b]
  report ifelse-value (a < b) [a] [b]
end
```

We can also define methods which take arguments and/or return/report values. This section outlines how this is done.

#do

assume that we have the following definition for a procedure with 2 arguments...

```
to ant.test [arg1 arg2]
  print (word "doing ant.test with " #self ", arg1=" arg1", arg2=" arg2)
end
```

...and a method set up as follows...

```
#def-method "ant.test"
```

...we can call this method, passing it argument values, using #do, eg:

```
observer> ask one-of turtles [ #do"test" ["mango" "melon"] ]
doing ant.test with 5, arg1=mango, arg2=melon
```

Note: in practice #"method" is a convenience abbreviation for #do"method" #nil

#report

assume we have the following definitions...

```
#def-method "ant.min"

to-report ant.min [arg1 arg2]
  print (word "doing ant.min with " #self ", arg1=" arg1", arg2=" arg2)
  report min (list arg1 arg2)
end
```

this method would be called using #report eg: `#report"min" [128 37]`

the following examples makes this call from the command centre...

```
observer> ask one-of turtles [ print #report"min" [128 37] ]
doing ant.min with 5, arg1=128, arg2=37
37
```

#do-ob & #report-ob

#do-ob & #report-ob are versions of #do & #report which allow an object to be specified. Assume that an instance called "sue" is defined as follows...

```
#make-instance "ant.sue" #nil
```

...then the "test" and "min" methods could be called as follows...

```
observer> #do-ob"sue" "test" ["cat" "dog"]
doing ant.test with sue, arg1=cat, arg2=dog

observer> show #report-ob"sue" "min" [11 22]
doing ant.min with sue, arg1=11, arg2=22
observer: 11
```

accessing slots with named objects

for the sake of example we redefine "sue" with an instance variable "x"...

```
#make-instance "ant.sue" [["x" 5]]
```

We can manipulate this using #get-ob and #set-ob, see below...

```
observer> show #get-ob"sue" "x"
observer: 10
observer> #set-ob"sue" "x" "mango"
observer> show #get-ob"sue" "x"
observer: "mango"
```