Package 'proto'

October 14, 2022

Version 1.0.0

Title Prototype Object-Based Programming

Description An object oriented system using object-based, also
called prototype-based, rather than class-based object oriented ideas.

License GPL-2

URL https://github.com/hadley/proto

BugReports https://github.com/hadley/proto/issues

Suggests testthat, covr

RoxygenNote 5.0.1.9000

NeedsCompilation no

Author Hadley Wickham [cre],
Gabor Grothendieck [aut],
Louis Kates [aut],
Thomas Petzoldt [aut]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2016-10-29 00:23:07

R topics documented:

proto-package *Object-Oriented Programming with the Prototype Model*

Description

Object-oriented programming with the prototype model. "proto" facilitates object-oriented programming using an approach that emphasizes objects rather than classes (although it is powerful enough to readily represent classes too).

Examples

```
cat("parent\n")
oop <- proto(x = 10, view = function(.) paste("this is a:", .$x))
oop$ls()
oop$view()

cat("override view in parent\n")
ooc1 <- oop$proto(view = function(.) paste("this is a: ***", .$x, "***"))
ooc1$view()

cat("override x in parent\n")
ooc2 <- oop$proto(x = 20)
ooc2$view()
```

proto *Prototype object-based programming*

Description

proto creates or modifies objects of the proto object oriented system.

Usage

```
proto(. = parent.env(envir), expr = { }, envir = new.env(parent =
  parent.frame()), ..., funEnvir = envir)

as.proto(x, ...)

## S3 method for class 'environment'
as.proto(x, ...)

## S3 method for class 'proto'
as.proto(x, ...)

## S3 method for class 'list'
```

```
as.proto(x, envir, parent, all.names = FALSE, ...,
  funEnvir = envir, SELECT = function(x) TRUE)

is.proto(x)
```

Arguments

.	the parent object of the new object. May be a proto object or an environment.
expr	a series of statements enclosed in braces that define the variables and methods of the object. Empty braces, the default, may be used if there are no variables or methods to add at this time.
envir	an existing prototype object or environment into which the variables and methods defined in expr are placed. If omitted a new object is created.
...	for proto these are components to be embedded in the new object. For as.proto.list these are arguments to pass to proto in the case that a new object is created. for $.proto the method is evaluated at these arguments.
funEnvir	the environment of methods passed via ... are automatically set to this environment. Normally this argument is omitted, defaulting to envir; however, one can specify FALSE to cause their environment to not be set or one can specify some other environment or proto object to which their environment is to be set.
x	a list.
parent	a prototype object or environment which is to be used as the parent of the object. If envir is specified then its parent is coerced to parent.
all.names	only names not starting with a dot are copied unless all.names is TRUE.
SELECT	a function which given an object returns TRUE or FALSE such that only those for which SELECT returns TRUE are kept in the returned proto object.
list	list whose components are an alternate way to specifying arguments in place of ...{}

Details

The proto class is an S3 subclass of the R environment class. In particular this implies that proto objects have single inheritance and mutable state as all environments do. The proto function creates and modifies objects of the proto class. It (1) sets the parent of codeenvir to parent, (2) evaluates expr in the envir environment and (3) lazily evaluates the arguments in ...{} in the parent environment resetting the environment of any functions (where the resetting is also done lazily). All such functions are known as methods and should have the receiver object as their first argument. Conventionally this is . (i.e. a dot). Also .that and .super variables are added to the environment envir. These point to the object itself and its parent, respectively. Note that proto can be used as a method and overridden like any other method. This allows objects to have object-specific versions of proto. There also exist that() and super() functions which have the same purpose as .that and .super but do not rely on the .that and .super. .that, .super, that() and super() can only be used within methods that have their object as their environment. In addition that() and super() may only be used within the top level of such methods (and not within functions within such methods).

as.proto is a generic with methods for environments, proto objects and lists.

as.proto.list copies each component, el, of the list x into the the environment or proto object envir for which FUN(el) is TRUE. Components whose name begins with a dot, ., are not copied unless all.names is TRUE (and FUN(el) is TRUE). The result is a proto object whose parent is parent. If envir is omitted a new object is created through a call to proto with parent and ...{} as arguments. If parent is also omitted then the current environment is the parent. Note that if parent is a proto object with its own proto method then the proto method of the parent will override the one described here in which case the functionality may differ.

$ can be used to access or set variables and methods in an object.

When $ is used for getting variables and methods, calls of the form obj$v search for v in obj and if not found search upwards through the ancestors of obj until found unless the name v begins with two dots ... In that case no upward search is done.

If meth is a function then obj$meth is an object of class c("instantiatedProtoMethod", "function") which is a proto method with the first, i.e. proto slot, already filled in. It is normally used in the context of a call to a method, e.g. obj$meth(x,y). There also exists print.instantiatedProtoMethod for printing such objects. Be aware that an instantiated proto method is not the same as a proto method. An instantiated proto method has its first argument filled (with the receiver object) whereas the first argument of a proto method does not. If it is desired to actually return the method as a value not in the context of a call then use the form obj$with(meth) or obj[[meth]] which are similar to with(obj, meth) except that the variation using with will search through ancestors while [[will not search through ancestors). The difference between obj$meth and obj$with(meth) is that in the first case obj implicitly provides the first argument to the call so that obj$meth(x,y) and obj$with(meth)(obj,x,y) are equivalent while in the case of obj$with(meth) the first argument is not automatically inserted.

$.proto also has a multiple argument form. If three or more arguments are present then they specify the arguments at which the instantiated method is to be evaluated. In this form the receiver object must be specified explicitly. This form can be used in situations where the highest speed is required such as in the inner loops of computations.

The forms .that$meth and .super$meth are special and should only be used within methods. .that refers to the object in which the current method is located and .super refers to the parent of .that. In both cases the receiver object must be specified as the first argument – the receiver is not automatically inserted as with other usages of $.

$ can be used to set variables and methods in an object. No ancestors are searched for the set form of $. If the variable is the special variable .super then not only is the variable set but the object's parent is set to .super.

A with method is available for proto objects.

is.proto(p) returns TRUE if p is a prototype object.

str.proto is provided for inspecting proto objects.

Value

proto and as.proto all return proto objects.

Note

proto methods can be used with environments but some care must be taken. Problems can be avoided by always using proto objects in these cases. This note discusses the pitfalls of using environments for those cases where such interfacing is needed.

If e is an environment then e$x will only search for x in e and no further whereas if e were a proto object its ancestors will be searched as well. For example, if the parent of a proto object is an environment but not itself a proto object then .super$x references in the methods of that object will only look as far as the parent.

Also note that the form e$meth(...) when used with an environment will not automatically insert e as the first argument and so environments can only be used with methods by using the more verbose e$meth(e, ...). Even then it is not exactly equivalent since meth will only be looked up in e but not its ancestors. To get precise equivalence write the even more verbose with(e, meth)(e,...).

If the user has a proto object obj which is a child of the global environment and whose methods use .super then .super will refer to an environment, not a proto object (unless the global environment is coerced to a proto object) and therefore be faced with the search situation discussed above. One solution is to create an empty root object between the global environment and obj as in this diagram Root <- obj$.super <- proto(.GlobalEnv) where Root is the root object. Now .super references will reference Root, which is a proto object so search will occur as expected. proto does not provide such a root object automatically but the user can create one easily, if desired.

Although not recommended, it possible to coerce the global environment to a proto object by issuing the command as.proto(.GlobalEnv). This will effectively make the global environment a proto root object but has the potential to break other software, although the authors have not actually found any software that it breaks.

See Also

as.list, names, environment

Examples

```
oo <- proto(expr = {
   x = c(10, 20, 15, 19, 17)
   location = function(.) mean(.$x) # 1st arg is object
   rms = function(.) sqrt(mean((.$x - .$location())^2))
   bias = function(., b) .$x <- .$x + b
})

debug(oo$with(rms)) # cannot use oo$rms to pass method as a value
undebug(oo$with(rms)) # cannot use oo$rms to pass method as a value

oo2 <- oo$proto( location = function(.) median(.$x) )
oo2$rms()      # note that first argument is omitted.
oo2$ls()       # list components of oo2
oo2$as.list()  # contents of oo2 as a list
oo2            # oo2 itself
oo2$parent.env() # same
oo2$parent.env()$as.list() # contents of parent of oo2
oo2$print()
oo2$ls()
oo2$str()
oo3 <- oo2
oo2$identical(oo3)
oo2$identical(oo)
```

```
# start off with Root to avoid problem cited in Note
Root <- proto()
oop <- Root$proto(a = 1, incr = function(.) .$a <- .$a+1)
ooc <- oop$proto(a = 3) # ooc is child of oop but with a=3
ooc$incr()
ooc$a      # 4

# same but proto overridden to force a to be specified
oop$proto <- function(., a) { .super$proto(., a=a) }
## Not run:
ooc2 <- oop$proto() # Error. Argument "a" is missing, with no default.

## End(Not run)
ooc2 <- oop$proto(a = 10)
ooc2$incr()
ooc2$a # 11

# use of with to eliminate having to write .$a
o2 <- proto(a = 1, incr = function(.) with(., a <- a+1))
o2c <- as.proto(o2$as.list()) # o2c is a clone of o2
o2d <- o2$proto()  # o2d is a child of o2
o2$a <- 2
o2c$a  # a not changed by assignment in line above
o2d$a # a is changed since a not found in o2d so found in o2

p <- proto(a = 0, incr = function(., x) .$a <- .$a + x)
pc <- p$proto(a = 100)

p$incr(7)
p$incr(x=7)
p$a
```

Index