# Package 'aoos'

May 7, 2017

**Type** Package

**Title** Another Object Orientation System

**Version** 0.5.0

**Date** 2017-05-06

**BugReports** https://github.com/wahani/aoos/issues

**URL** https://wahani.github.io/aoos

**Description** Another implementation of object-orientation in R. It provides syntactic sugar for the S4 class system and two alternative new implementations. One is an experimental version built around S4 and the other one makes it more convenient to work with lists as objects.

**Depends** methods, R(>= 3.2.0)

**Imports** magrittr, utils, roxygen2

**License** MIT + file LICENSE

**Suggests** testthat, knitr, rbenchmark, R6, rmarkdown

**Encoding** UTF-8

**VignetteBuilder** knitr

**ByteCompile** TRUE

**Collate** 'DC-Binary.R' 'DC-Show.R' 'DC-aoos.R' 'DC-defineClass.R' 'DC-public-interfaces.R' 'NAMESPACE.R' 'RL-Infix.R' 'RL-envHelper.R' 'RL-retList.R' 'S4-expressions.R' 'S4-generics.R' 'S4-generics-test.R' 'S4-roxygen-parser.R' 'S4-types.R' 'S4RC-Accessor.R' 'S4RC-Private.R' 'S4RC-defineRefClass.R'

**RoxygenNote** 6.0.1

**NeedsCompilation** no

**Author** Sebastian Warnholz [aut, cre]

**Maintainer** Sebastian Warnholz <wahani@gmail.com>

**Repository** CRAN

**Date/Publication** 2017-05-07 05:33:32 UTC

# R topics documented:



---

`.genericTest` *Generic Test*

---

## Description

This generic function only exists to test that the rexygen2 parser work correctly. Just ignore it.

## Usage

```
.genericTest(x, ...)

## S4 method for signature 'numeric'
.genericTest(x, ..., methodParam = function() 1)
```

## Arguments

| | |
|---|---|
| `x` | Object |
| `...` | Object |
| `methodParam` | Object |

## Accessor-class *Accessor class*

### Description

This is a virtual class to be contained in other class definitions. It overrides the default accessor `$` and is intended to be used with the aoos class system (`defineClass`). Inherit from this class if you want to access public fields in the same way you access lists.

### Usage

```
## S4 method for signature 'Accessor'
x$name

## S4 replacement method for signature 'Accessor'
x$name <- value
```

### Arguments

| | |
|---|---|
| `x` | object |
| `name` | member name |
| `value` | value to assign to. |

## aoos-class *Class aoos*

### Description

This is an environment with some methods. Every class defined by `defineClass` will inherit from aoos. Summary will show a list of public and private members with approximated memory usage.

### Usage

```
## S4 method for signature 'aoos'
show(object)

## S4 method for signature 'aoos'
x$name

## S4 replacement method for signature 'aoos'
x$name <- value

## S3 method for class 'aoos'
summary(object, ...)

## S4 method for signature 'aoos'
as.environment(x)
```

### Arguments

| | |
|---|---|
| object | object |
| x | object |
| name | member name |
| value | value to assign to. Will throw an error. |
| ... | arguments passed to method (not used). |

---

## Binary-class *Binary-class*

---

### Description

This is a virtual class to be contained in other class definitions. It can be used to define binary operators, e.g. + or -, inside an aoos class definition (defineClass).

### Details

At the moment you can define binary operators as methods by naming them as .<binaryOperator> (see the example). This is implemented for the following operators: +, -, *, /, %%, ^, <, >, ==, >=, <=, &.

### Examples

```
Rational <- defineClass("Rational", contains = c("Show", "Binary"), {

  numer <- 0
  denom <- 1
  .g <- 1

  .gcd <- function(a, b) if(b == 0) a else Recall(b, a %% b)

  init <- function(numer, denom) {
    .self$.g <- .gcd(numer, denom)
    .self$numer <- numer / .g
    .self$denom <- denom / .g
  }

  show <- function() {
    cat(paste0(.self$numer, "/", .self$denom, "\n"))
  }

  ".+" <- function(that) {
    Rational(numer = numer * that$denom + that$numer * denom,
             denom = denom * that$denom)
  }

  neg <- function() {
    Rational(numer = -.self$numer,
```

```
            denom = .self$denom)
  }

  ".-" <- function(that) {
     .self + that$neg()
  }

})

rational <- Rational(2, 3)
rational + rational
rational$neg()
rational - rational
```

---

`defineClass` *Define a new class*

---

## Description

This is an experimental implementation of reference classes. Use `defineRefClass` or `retList` instead. `defineClass` has side effects. The constructor is the return value of `defineClass`.

## Usage

```
defineClass(name, expr, contains = NULL)

private(x)

## S4 method for signature 'public'
private(x)

public(x = NULL, validity = function(x) TRUE)

## S4 method for signature '`function`'
public(x = NULL, validity = function(x) TRUE)

## S4 method for signature 'private'
public(x = NULL, validity = function(x) TRUE)

## S4 method for signature 'public'
public(x = NULL, validity = function(x) TRUE)
```

## Arguments

| | |
|---|---|
| `name` | character name of the class |
| `expr` | expression |
| `contains` | character name of class from which to inherit |
| `x` | an object made public |
| `validity` | function to check the validity of an object |

## Details

`defineClass` creates a S4-Class which can be used for standard S4 method dispatch. It will also set the method 'initialize' which need not to be changed. If you want to have some operations carried out on initialization use a function definition named `init` as part of `expr`. The return value from `defineClass` is the constructor function. It has the argument `...` which will be passed to `init`.

All classes defined with `defineClass` inherit from class "aoos" which is a S4-class containing an environment. In that environment `expr` is evaluated; for inheritance, all `expr` from all parents will be evaluated first.

Everything in `expr` will be part of the new class definition. A leading dot in a name will be interpreted as private. You can use `public` and `private` to declare private and public members explicitly. If `x` in a call to `public` is a function it will be a public member function (method). For any other class the return value of `public` is a get and set method. If called without argument it will get the value, if called with argument it will set the value. You can define a validity function which will be called whenever the set method is called. Objects which inherit from class `environment` can be accessed directly, i.e. not via get/set methods. If you want to access fields without get/set methods, you can use the class `Accessor-class`.

## See Also

`Accessor-class`, `Binary-class`, `Show-class`

## Examples

```
test <- defineClass("test", {
  x <- "Working ..."
  .y <- 0
  doSomething <- public(function() {
    self$.y <- .y + 1
    cat(x(), "\n")
    invisible(self)
  })
})
instance <- test()
## Not run:
instance$.y # error

## End(Not run)
instance$doSomething()$doSomething()
instance$x()
instance$x(2)
instance$x()

# Example for reference classes as field
MoreTesting <- defineClass("MoreTesting", {
  refObj <- test()
})
instance <- MoreTesting()
instance$refObj$x()
```

defineRefClass *Define a Reference Class*

## Description

This is a wrapper around `setRefClass`. All arguments are defined in an expression (instead of lists) which improves readability of the code. Besides that, no additional features are added.

## Usage

```
defineRefClass(expr)
```

## Arguments

| | |
|---|---|
| expr | an expression |

## See Also

Private-class

## Examples

```
## Not run:
  vignette("Introduction", "aoos")

## End(Not run)

# Minimal example:
Test <- defineRefClass({
  Class <- "Test" # this is passed as argument to setRefClass
  x <- "character" # all objects which are not functions are fields
  do <- function() cat("Yes, Yes, I'm working...") # a method
})

test <- Test()
test$x <- "a"
test$do()

# Inheritance and privacy:
pTest <- defineRefClass({
  Class <- "pTest"
  # Privacy is solved by inheriting from a class 'Private' which redefines
  # the methods for access.
  contains <- c("Test", "Private") # passed as argument to setRefClass

  .y <- "numeric" # this is going to be 'private'

  doSomething <- function() {
    .self$.y <- 42
    cat(x, .y, "\n")
```

```
    invisible(.self)
  }
})

instance <- pTest()
instance$x <- "Value of .y:"
instance$doSomething()

# A notion of privacy:
stopifnot(inherits(try(instance$.y), "try-error"))
stopifnot(inherits(try(instance$.y <- 2), "try-error"))
```

---

envCopy *Helpers for environments*

---

## Description

Functions to help working with environments.

## Usage

```
envCopy(from, to)

envMerge(x, with)
```

## Arguments

| | |
|---|---|
| from | environment |
| to | environment |
| x | environment |
| with | environment |

## Details

envCopy tries to copy all objects in a given environment into the environment 'to'. Returns the names of copied objects.

envMerge will merge x and with. Merge will copy all objects from x to with. Prior to that, the environment of functions are changed to be with iff functions in x have environment x; else the environment of functions are preserved.

## See Also

retList where these are relevant.

## parser_%m% *Parser for roxygen documentation*

### Description

These functions are used by roxygen2 for generating documentation.

### Usage

```
"parser_%m%"(call, env, block)

"parser_%g%"(call, env, block)

"parser_%type%"(call, env, block)
```

### Arguments

| | |
|---|---|
| call | a call |
| env | an environment |
| block | is ignored |

## print.Print *S3 helper classes*

### Description

There is no formal class definition for S3. Simply add 'Infix' or 'Print' to the class attribute and it inherits the methods. It is the same as Binary-class or Show-class just for S3. This is inteded to be used with retList.

### Usage

```
## S3 method for class 'Print'
print(x, ...)

## S3 method for class 'Infix'
e1 + e2

## S3 method for class 'Infix'
e1 - e2

## S3 method for class 'Infix'
e1 / e2

## S3 method for class 'Infix'
```

```
e1 %% e2

## S3 method for class 'Infix'
e1 ^ e2

## S3 method for class 'Infix'
e1 < e2

## S3 method for class 'Infix'
e1 > e2

## S3 method for class 'Infix'
e1 == e2

## S3 method for class 'Infix'
e1 >= e2

## S3 method for class 'Infix'
e1 <= e2

## S3 method for class 'Infix'
e1 & e2

## S3 method for class 'Infix'
!x

## S3 method for class 'Infix'
as.environment(x)
```

## Arguments

| | |
|---|---|
| x | an object |
| ... | arguments passed to the local print method. |
| e1 | lhs operand |
| e2 | rhs operand |

## Details

The lhs is coerced with `as.environment` and in that environment the binary operators must be found and named as `.<binaryOperator>` (see the example for `retList`). This is implemented for the following operators: `+`, `-`, `*`, `/`, `%%`, `^`, `<`, `>`, `==`, `>=`, `<=`, `&`. Also part of the operators you can implement with Infix is `!`, although it is unary.

## See Also

Binary-class, retList

`Private-class` *Private class*

## Description

This is a virtual class to be contained in other class definitions. It overrides the default subset functions `$` and `[[` such that private member of a class can not be accessed. Private is every object which has a name with a leading "." (`grepl("^\\.", name)`). After this check the standard method for class 'envRefClass' is called or an error is reported.

## Usage

```
## S4 method for signature 'Private'
x$name

## S4 replacement method for signature 'Private'
x$name <- value

## S4 method for signature 'Private'
x[[i, j, ...]]

## S4 replacement method for signature 'Private'
x[[i, j, ...]] <- value
```

## Arguments

| | |
|---|---|
| `x` | the object |
| `name` | name of field or method |
| `value` | any object |
| `i` | like name |
| `j` | ignored |
| `...` | ignored |

## See Also

defineRefClass

## Examples

```
ClassWithPrivateField <- defineRefClass({
  Class <- "ClassWithPrivateField"
  contains <- "Private"

  .p <- "numeric"

  getP <- function() .p
  setP <- function(v) .self$.p <- v
```

```
})

test <- ClassWithPrivateField()
stopifnot(inherits(try(test$.p), "try-error"))
stopifnot(inherits(try(test$.p <- 2), "try-error"))
stopifnot(inherits(try(test[[".p"]]), "try-error"))
stopifnot(inherits(try(test[[".p"]] <- 2), "try-error"))
```

---

`publicFunction` *Constructors for public members*

---

## Description

These functions are used internally. You should not rely on them. Use `public` instead.

## Usage

```
publicFunction(fun)

publicValue(x = NULL, validity = function(x) TRUE)

## S4 method for signature 'publicEnv'
x$name
```

## Arguments

| | |
|---|---|
| `fun` | function definition |
| `x` | a default value |
| `validity` | an optional validity function for the set method. Returns TRUE or FALSE. |
| `name` | name of member in refernece object |

---

`retList` *Generic constructor function*

---

## Description

This functions can be used to construct a list with class attribute and merged with another list called super. The constructed list will contain (by default) all visible objects from the environment from which `retList` is called.

## Usage

```
retList(class = NULL, public = ls(envir), super = list(),
  superEnv = asEnv(super), mergeFun = envMerge, envir = parent.frame())

funNames(envir = parent.frame())

asEnv(x)

stripSelf(x)
```

## Arguments

| | |
|---|---|
| class | character giving the class name. |
| public | character with the names to include. |
| super | a list/object to be extended. |
| superEnv | environment where new methods will live in. |
| mergeFun | function with two arguments. Knows how to join/merge environments - `mergeFun(envir, superEnv)`. Default: envMerge. |
| envir | this is the environment you want to convert into the list. Default is the environment from which the function is called. |
| x | a list |

## Details

`funNames` returns the names of functions in the environment from which it is called.

`asEnv` trys to find an environment for x. If x is NULL or an empty list, the function returns `NULL`. (Else) If x has an attribute called `.self` it is this attribute which is returned. (Else) If x is a list it is converted to an environment.

## See Also

ls, +.Infix, print.Print

## Examples

```
# To get a quick overview of the package:
vignette("Introduction", "aoos")

# To get more infos about retList:
vignette("retListClasses", "aoos")

# To get some infos about performance:
vignette("performance", "aoos")

# A simple class with one method:
Test <- function(.x) {
  getX <- function() .x
  retList("Test")
```

```
}

stopifnot(Test(2)$getX() == 2)

# A second example inheriting from Test
Test2 <- function(.y) {
  getX2 <- function() .x * 2
  retList("Test2", super = Test(.y))
}

stopifnot(Test2(2)$getX() == 2)
stopifnot(Test2(2)$getX2() == 4)

### Rational numbers example with infix operators and print method

Rational <- function(numer, denom) {

  gcd <- function(a, b) if(b == 0) a else Recall(b, a %% b)

  g <- gcd(numer, denom)
  numer <- numer / g
  denom <- denom / g

  print <- function(x, ...) cat(paste0(numer, "/", denom, "\n"))

  ".+" <- function(that) {
    Rational(numer = numer * that$denom + that$numer * denom,
             denom = denom * that$denom)
  }

  ".-" <- function(that) {
    if (missing(that)) {
      Rational(-numer, denom)
    } else {
      .self + (-that)
    }
  }

  # Return only what should be visible from this scope:
  retList(c("Rational", "Infix", "Print"),
          c("numer", "denom", "neg", "print"))

}

rational <- Rational(2, 3)
rational + rational
rational - rational
```

---

Show-class *Show class*

---

## Description

This is a virtual class to be contained in other class definitions. It overrides the default show method and is intended to be used with the aoos class system (defineClass). The show method will simply look for a method show defined as member of a class definition.

## Usage

```
## S4 method for signature 'Show'
show(object)
```

## Arguments

| | |
|---|---|
| object | an object inheriting from Show |

## See Also

defineClass

## Examples

```
ClassWithShowMethod <- defineClass("ClassWithShowMethod", contains = "Show", {
  show <- function() print(summary(.self))
})

ClassWithShowMethod()
```

---

# %g% *Wrapper for writing S4 generics and methods*

---

## Description

These are two wrappers around setGeneric and setMethod. A relevant difference is that generics and methods are stored in the environment in which %g% and %m% are called and not in the top-environment. Furthermore both functions have side effects in that they will call globalVariables for the arguments and name of the generic.

## Usage

```
lhs %g% rhs

lhs %m% rhs
```

## Arguments

| | |
|---|---|
| lhs | see details |
| rhs | the body as an expression |

## Details

The Syntax for the left hand side:
`[<valueClass>:]<genericName>(<argList>)`
- `valueClass` optional, is the class of the return value (see setGeneric)
- `genericName` the name of the generic function
- `argList` are `name = value` or `name ~ type` expressions. Name-Value expressions are just like in a function definition. Name-Type expressions are used to define the signature of a method (see setMethod). See %type% and the examples how to work with them.

## Examples

```
# A new generic function and a method:
numeric : generic(x) %g% standardGeneric("generic")
generic(x ~ numeric) %m% x
generic(1)

# Polymorphic methods in an object:
Object <- function() {
  numeric : generic(x) %g% standardGeneric("generic")
  generic(x ~ numeric) %m% x
  retList("Object")
}
Object()$generic(1)

# Class Unions:
## This generic allows for return values of type numeric or character:
'numeric | character' : generic(x) %g% standardGeneric("generic")

## This method also allows for numeric or character as argument:
generic(x ~ character | numeric) %m% x
generic(1)
generic("")
```

---

`%type%` *Types*

---

## Description

This function can be used to define new S4-classes which are called Type. They have an initialize method and in the introduced syntax init-method and S4-class definition build a unit, hence a type. This simply captures a typical `setClass` then `setMethod("initialize", ...)` pattern where often some redundancy is introduced. The function has side effects due to calling `setClass`, `setMethod` and assigning the constructor function to the types name.

## Usage

```
lhs %type% rhs
```

## Arguments

| | |
|---|---|
| lhs | an expression of the form:<br>[<parent-name>:]<type-name>([<slots>])<br>- <parent-name> optional, the name of the S4-class/type to inherit from, seperated by :<br>- <type-name> the name for the new type and constructor function.<br>- <slots> optional, name = value or name ~ type expressions. Name-Value expressions are used to construct a prototype. From the prototype the class of the slot will be inferred. They are also the defaults in the type constructor. Name-Type expressions define the classes of the slots. If no value (or type) is supplied, ANY is assumed. |
| rhs | the body of the initialize method as expression. It will be called with .Object and ... as arguments. .Object should be the return value. With .Object there is an instance of the type on which assertions can be formulated. Prior to the body (rhs) .Object <- callNextMethod() will be evaluated which enables proper initialization of your type and its inherited fields. See initialize for details. |

## Details

Name-Type expressions are also used in %m%. Besides this you can formulate type unions in type expressions or the inheritance structure. This has a side effect in that setClassUnion is called. Whenever you write a type you can replace the name by an expression of the form: type1 | type2. Outside the slots or argument list of a method these expressions have to be quoted. In this example the following expression is evaluated for you: setClassUnion("type1ORtype2", c("type1", "type2")).

## Examples

```
# This will create an S4-class named 'Test' with two slots; x = "numeric"
# and y = "list"; prototype: list(x = 1, y = list()); and an initialize
# method where some checks are performed.

Test(x = 1, y = list()) %type% {
  stopifnot(.Object@x > 0)
  .Object
}

# This will create an S4-class named 'Numeric' with a slot and some tests.

numeric : Numeric(metaInfo = character()) %type% {
  stopifnot(length(.Object) > 0)
  stopifnot(all(.Object > 0))
  .Object
}

# This will create an S4-class with slots, where the constructor function has
# no defaults. All slots will allow for ANY type.

Anything(x, y ~ ANY, z = NULL) %type% .Object
## Not run:
```

```
  Anything() # error because x and y are missing

## End(Not run)

# Type Unions:
'character | numeric' : Either(either ~ character | numeric) %type% .Object
Either("", 1)
```

# Index