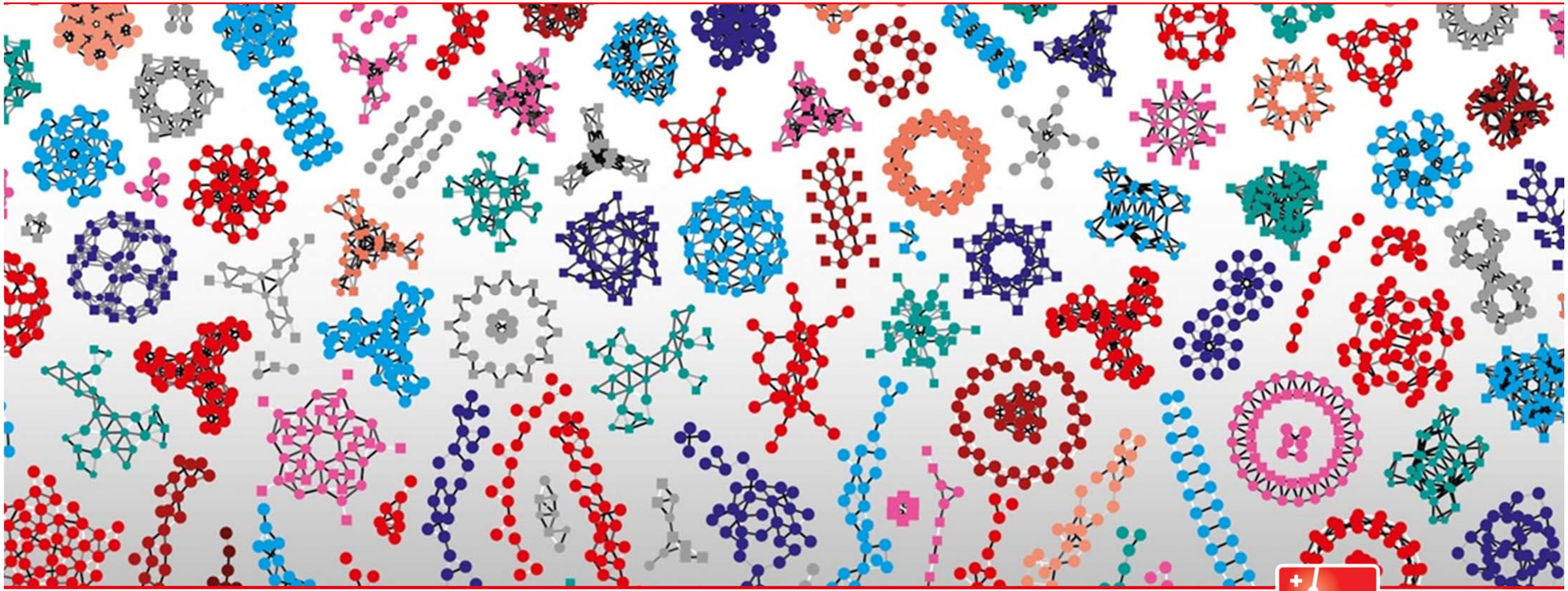


Swiss Institute of Bioinformatics

Advanced R: Object-oriented programming
September 2024

Frédéric Schütz (Frederic.Schutz@sib.swiss)

Reminder: getting information about R objects

The `summary()` command gives some brief information about an R object; its output depends on the type of object:

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```

Reminder: getting information about R objects

How does `summary()` know what to print for different objects ?

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```

Object-oriented programming in R

Fundamentals of object-oriented programming

Object: mechanism (usually data structure) that stores data and provides controlled access to it

Class: specification of the data and access mechanisms that a specific type of object supplies (blueprint)

Attribute: a piece of data owned by an object (or by a class)

Method: subroutine that provides some kind of access to an object's (or class's) data

Inheritance: reuse of attribute and method specifications from an existing class

Polymorphism: redefinition of behaviour of inherited methods

Adapted from Damian Conway, «*Introductory Object-Oriented Perl*»

Two frameworks for Object-oriented programming in R

S3

- Informal, exists since the beginning
- Widely used, in particular in the base packages

S4

- More formal and rigorous, but less interactive
- Since R 1.7
- Used systematically in some contexts, e.g. Bioconductor

The S3 system

Every object has a **class label** attached to it, either

- explicitely set (using the `class()` function)
- matrix or array
- integer
- or the same as the mode of the object (`mode()`)

Examples of classes

```
> a <- c(1,1,2,3); class(a)
[1] "numeric"

> M <- matrix(1:4, ncol=2); class(M)
[1] "matrix"

> model <- lm( y ~ x ); class(model)
[1] "lm"

> f <- factor(a); class(f)
[1] "factor"
```

Getting a `summary()` *of each of these variables*

```
> summary(a)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1.00    1.00    1.50    1.75    2.25    3.00 

> summary(M)
       V1              V2      
 Min.   :1.00   Min.   :3.00  
 1st Qu.:1.25   1st Qu.:3.25  
 Median :1.50   Median :3.50  
 Mean   :1.50   Mean   :3.50  
 3rd Qu.:1.75   3rd Qu.:3.75  
 Max.   :2.00   Max.   :4.00  

> summary(model)

Call:
lm(formula = y ~ x)

[…]
Multiple R-squared: 0.997, Adjusted R-squared: 0.9967 
F-statistic:  2695 on 1 and 8 DF,  p-value: 2.102e-11 

> summary(f)
1 2 3 
2 1 1 
```

Method dispatch*: How does R creates the right summary ?*

The `summary()` function is defined as a generic function, whose sole role is to call the right method (function) for creating a summary depending on the class of the object:

```
> summary
function (object, ...)
UseMethod("summary")
<environment: namespace:base>
```

Method dispatch: *How does R create the right summary ?*

If object `sheldon` is of class `bazinga`, when calling `summary(sheldon)`, R will search for a function called `summary.bazinga`.

If it exists, it will call it:

```
summary.bazinga(sheldon)
```

If `summary.bazinga` does not exist, R will call

```
summary.default(sheldon)
```

Method dispatch: which classes know a given method ?

```
> methods("summary")
 [1] summary.aov             summary.aovlist       summary.aspell*
 [4] summary.connection      summary.data.frame    summary.Date
 [7] summary.default         summary.ecdf*         summary.factor
[10] summary.glm             summary.infl          summary.lm
[13] summary.loess*          summary.manova        summary.matrix
[16] summary.mlm             summary.nls*          summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*   summary.POSIXct
[22] summary.POSIXlt         summary.ppr*          summary.prcomp*
[25] summary.princomp*       summary.srcfile       summary.srcref
[28] summary.stepfun         summary.stl*          summary.table
[31] summary.tukeysmooth*

   Non-visible functions are asterisked
```

Method dispatch: which classes know a given method ?

```
> methods("summary")
 [1] summary.aov             summary.aovlist       summary.aspell*
 [4] summary.connection      summary.data.frame    summary.Date
…

   Non-visible functions are asterisked
```

To see the body of a non-visible function in R:

```
getS3method("summary", "princomp")
getAnywhere("summary.princomp")
```

Method dispatch: which methods exist for a given class ?

```
> methods(class="lm")
 [1] add1.lm*           alias.lm*          anova.lm
 [4] case.names.lm*     confint.lm*        cooks.distance.lm*
 [7] deviance.lm*       dfbeta.lm*         dfbetas.lm*
[10] drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*
[16] hatvalues.lm       influence.lm*      kappa.lm
[19] labels.lm*         logLik.lm*         model.frame.lm
[22] model.matrix.lm    nobs.lm*           plot.lm
[25] predict.lm         print.lm           proj.lm*
[28] qr.lm*             residuals.lm       rstandard.lm
[31] rstudent.lm        simulate.lm*       summary.lm
[34] variable.names.lm* vcov.lm*
```

The `print()` method

```
> model

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
  -0.001372     2.014997

> class(model)
[1] "lm"
> print.lm(model)   # Equivalent to print(model)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
  -0.001372     2.014997

# See print.lm for the details of how this information is printed
```

How to create an S3 object ?

1) A function creates a structure with the attributes your object need

```
createobj <- function(param1, param2, param3, ...) {

  # The function calculates some results
  ...
  # The results (say, in variable "res") are then
  # stored in object, along with parameters (if needed)

  z <- list(res1=res[1], res2=res[2], ...)
  z$param1    <- param1
  z$param2    <- param2
  ...


  z
}
```

(*) Any R object could be used, but lists are almost always used

This structure is almost always a list (but any R object could be used)

```
createobj <- function(param1, param2, param3, ...) {

  # The function calculates some results
  ...
  # The results (say, in variable "res") are then
  # stored in object, along with parameters (if needed)

  z <- list(res1=res[1], res2=res[2], ...)
  z$param1    <- param1
  z$param2    <- param2
  ...



  z
}
```

2) It labels your list with the model name and returns the object

```
createobj <- function(param1, param2, param3, ...) {

  # The function calculates some results
  ...
  # The results (say, in variable "res") are then
  # stored in object, along with parameters (if needed)

  z <- list(res1=res[1], res2=res[2], ...)
  z$param1    <- param1
  z$param2    <- param2
  ...

  class(z) <- "myclass"
  z
}
```

3) *Create the methods the user of the class/object will need*

```
print.myclass <- function(object) {
  if (! any( class(object)=="myclass"))
    stop("Error: object is not a myclassobject.")

  # Program how the object will be "printed"
}

summary.myclass <- function(object) {
  if (! any( class(object)=="myclass"))
    stop("Error: object is not a myclass object.")

  # Program how the object will be "summarised"
}

plot.myclass <- function(object) {
  # …
}
```

4) If needed, create a new generic method

```
# Create a "reduce" method for myclass objects
reduce.myclass <- function(object) {

  if (! any( class(object)=="myclass"))
    stop("Error: object is not a myclassobject.")

  # Do something with the object
  ...
}

# Create a new generic method "reduce"
reduce <- function(object) UseMethod("reduce")
```

5) The class can then be used

```
> myobject <- createobject(param1, param2, param3, …)
> class(myobject)
[1] "myclass"

> myobject   # equivalent to   print(myobject)
# Result of the printing

> summary(myobject)
# Result of the summary function

> plot(myobject)
# Result of the plot function

> newobject <- reduce(myobject)
>
```

How to create an S3 object ?
Example: the mygsea2 package

The function that creates a "gsea" object

```
mygsea2 <- function(small.list, big.list, n.perm,
                    weights) {

  # The function calculates "res" (results)…
  ...
  # … which are then stored in a list:
  z <- list(ks.pos=res$resks[1],  ks.neg=res$resks[2],
            p.pos=res$resperm[1], p.neg=res$resperm[2])
  z$nperms     <- n.perm
  z$weights    <- weights
  z$small.list <- small.list
  z$big.list   <- big.list

  class(z) <- "gsea"
  z
}
```

3) Create the methods the user of the class/object will need

```
print.gsea <- function(object) {

  if (! any( class(object)=="gsea"))
    stop("Error: object is not a gsea object.")

  cat("GSEA analysis (", object$nperms," perms.)\n\n", sep="")
  cat("Small list: ", length(object$small.list),"\n",
      "  Big list: ", length(object$big.list),"\n\n", sep="")

  coefs <- cbind( c(object$ks.pos, object$ks.neg),
                  c(object$p.pos, object$p.neg) )
  colnames(coefs) <- c("Ks stat","P-value")
  rownames(coefs) <- c("+", "-")

  printCoefmat(coefs, P.values=TRUE, has.Pvalue=TRUE)
}
```

4) If needed, create a new generic method

```
# Create a reduce method for gsea objects
reduce.gsea <- function(object) {

  if (! any( class(object)=="gsea"))
    stop("Error: object is not a gsea object.")

  # Do something with the object
  ...
}

# Create a new generic method "reduce"
reduce <- function(object) UseMethod("reduce")
```

Shortcomings of this informal system

The user can easily access the attributes directly (although he/she should not !), as with any other R object:

```
> class(model)
[1] "lm"
> names(model)
[1] "coefficients"  "residuals"      "effects"      "rank"
[5] "fitted.values" "assign"         "qr"           "df.residual"
[9] "xlevels"       "call"           "terms"        "model"

> coef(model)               # Recommended way
 (Intercept)            x
-0.001371868  2.014997472
> model$coefficients        # Not recommended
 (Intercept)            x
-0.001371868  2.014997472
```

Shortcomings of this informal system

The user can easily modify an attribute or the class itself, and R will not complain, unless you call a method that does not work anymore.

```
> class(model)
[1] "lm"
> model$coefficients <- c(0,0)
> model

Call:
lm(formula = y ~ x)

Coefficients:
[1]  0  0

> a <- 1:10; class(a) <- "lm"
> summary(a)
Error: $ operator is invalid for atomic vectors
```

The S4 model

- The S4 model is based on the same ideas («method dispatch») than S3

- It is however implemented in a much formal and stricter way.

- It also allows for «multiple dispatch»

Defining a class

```
setClass("GSEA",
         representation( nperms="numeric", weights="numeric",
                         small.list="character",
                         big.list="character"),
         contains="genelist",
         validity=function(object) {
           length(object@weights)==length(object@big.list)
         }
         )
```

Properties of a class include:

- A **name**
- A **representation**: list of attributes (*slots*) that the object contains
- **Inheritance**
- A **prototype** that specifies default values
- A **validation** function
- etc (see `?setClass`)

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])
```

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])

> gsea <- new("GSEA", nperms="a", weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])
```

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])

> gsea <- new("GSEA", nperms="a", weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])
Error in validObject(.Object) :
  invalid class “GSEA” object: invalid object for slot "nperms"
in class "GSEA": got class "character", should be or extend
class "numeric"
```

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])

> gsea <- new("GSEA", nperms="a", weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])
Error in validObject(.Object) :
  invalid class “GSEA” object: invalid object for slot "nperms"
in class "GSEA": got class "character", should be or extend
class "numeric"

> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list="a")
```

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])

> gsea <- new("GSEA", nperms="a", weights=1:10,
              small.list=c("A", "B", "C"),
              big.list=LETTERS[1:10])
Error in validObject(.Object) :
  invalid class “GSEA” object: invalid object for slot "nperms"
in class "GSEA": got class "character", should be or extend
class "numeric"

> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("A", "B", "C"),
              big.list="a")
Error in validObject(.Object) : invalid class “GSEA” object:
FALSE
```

Slots

Attributes in S4 objects are stored in *slots*.

They are similar to the components of a list for a S3 object, but well separated:

```
> slotNames(gsea)
[1] "nperms"     "weights"    "small.list" "big.list"

> gsea@nperms
[1] 10000

> gsea$nperms
Error in gsea$nperms : $ operator not defined for this S4 class
```

Note that you can still access and modify an object’s content directly using the slots and the @ operator (and bypass any validation !), as with S3 objects, but you really, really should not (please ?)

The «`show`» method

```
setMethod("show", "GSEA",
          function(object) {
            cat("GSEA with", object@nperms,"permutations.\n")
          }
          )

> gsea
GSEA with 10000 permutations.
```

This is equivalent to
the «`print`» method in S3

How to list available methods

```
> showMethods("show")
Function: show (package methods)
object="ANY"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
...
```

How to list available methods

```
> showMethods( class="GSEA")
Function: initialize (package methods)
.Object="GSEA"
    (inherited from: .Object="ANY")

Function: show (package methods)
object="GSEA"
```

Which system should I use ?

1) «While in Rome, Do as the Romans Do»:
   e.g. If your code fits with Bioconductor, use S4

2) Use S4 is there is a strong technical reason for doing so
   e.g. if you want to use objects directly in C++ code

3) Generally, use S3 objects and methods.

4) In any case, avoid mixing S3 and S4

Adapted from Google's R Style Guide:
https://google.github.io/styleguide/Rguide.xml

How to access some information in an unknown object ?

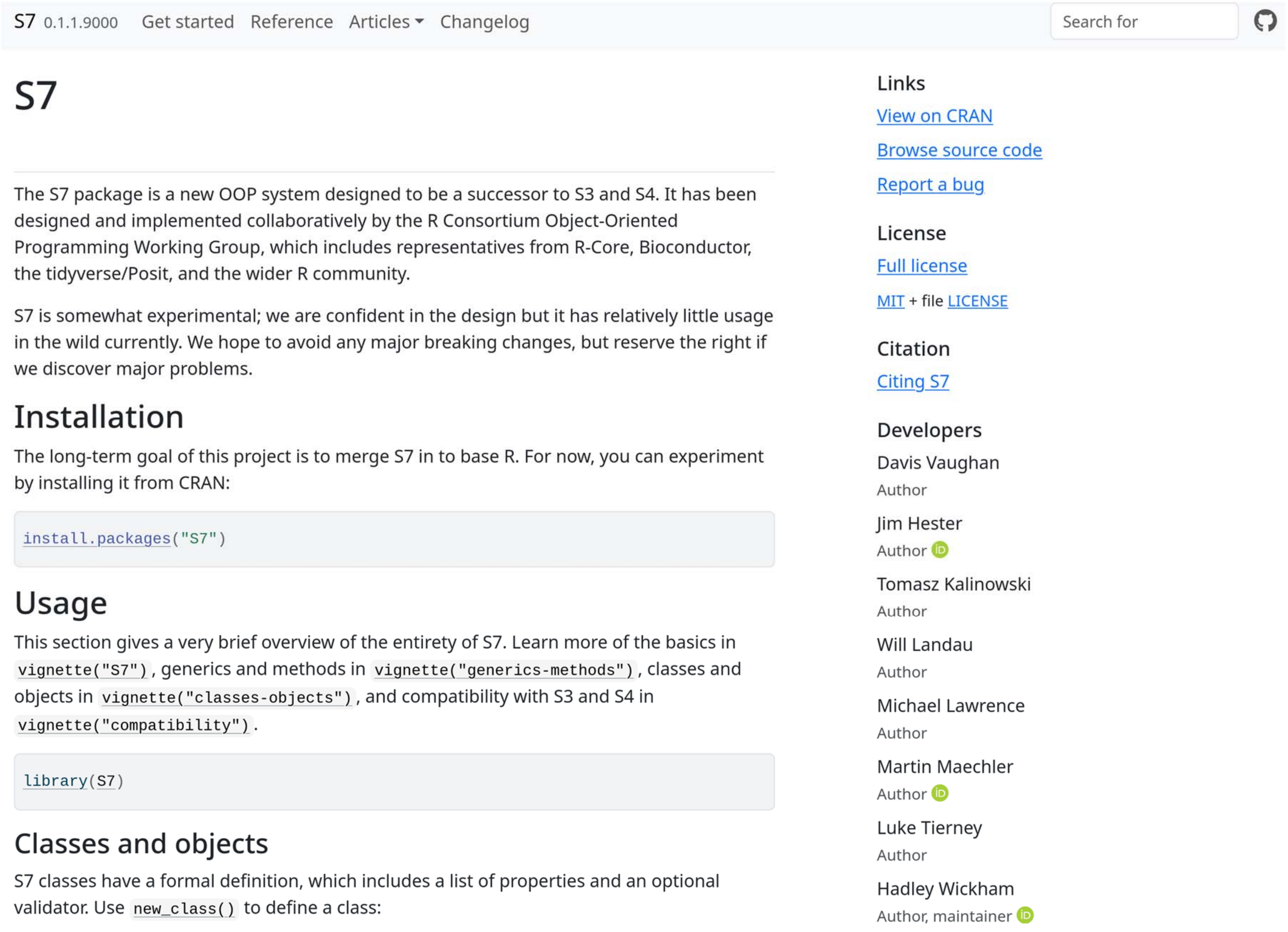
1) Look at `class(object)` (works with S3 and S4)
2) Look at its documentation
3) Find if the object is S3 or S4:
   - `names(object)` (empty for an S4 object)
   - `isS4(object)` (TRUE for an S4 object)
4) Look at the methods available for the object:
   - `methods(class="class")` for an S3 object
   - `showMethods(class="class")` for an S4 object

   and check whether one does what you need
5) Otherwise, look at its attributes (S3, $) or slots (S4, @)
6) If needed, look at a method to see how it handles the attributes:
   - `method.class` for an S3 object
   - `getMethods( "method", "class")` for an S4 object

RC: another framework for object-oriented development in R

- Introduced in R 2.12.0
- See: `?ReferenceClasses`

S7



S7

The S7 package is a new OOP system designed to be a successor to S3 and S4. It has been designed and implemented collaboratively by the R Consortium Object-Oriented Programming Working Group, which includes representatives from R-Core, Bioconductor, the tidyverse/Posit, and the wider R community.

S7 is somewhat experimental; we are confident in the design but it has relatively little usage in the wild currently. We hope to avoid any major breaking changes, but reserve the right if we discover major problems.

Installation

The long-term goal of this project is to merge S7 in to base R. For now, you can experiment by installing it from CRAN:

```
install.packages("S7")
```

Usage

This section gives a very brief overview of the entirety of S7. Learn more of the basics in `vignette("S7")`, generics and methods in `vignette("generics-methods")`, classes and objects in `vignette("classes-objects")`, and compatibility with S3 and S4 in `vignette("compatibility")`.

```
library(S7)
```

Classes and objects

S7 classes have a formal definition, which includes a list of properties and an optional validator. Use `new_class()` to define a class:

Links

View on CRAN

Browse source code

Report a bug

License

Full license

MIT + file LICENSE

Citation

Citing S7

Developers

Davis Vaughan
Author

Jim Hester
Author

Tomasz Kalinowski
Author

Will Landau
Author

Michael Lawrence
Author

Martin Maechler
Author

Luke Tierney
Author

Hadley Wickham
Author, maintainer

https://rconsortium.github.io/S7/

For more information…

- Thomas Lumley. “Programmer's Niche: A Simple Class, in S3 and S4” in R News 4/1, 2004, p. 33-36
  http://cran.r-project.org/doc/Rnews/Rnews_2004-1.pdf

- Hadley Wickham, chapters on Object-Oriented programming in the book “Advanced R”
  https://adv-r.hadley.nz/oo.html