# R Classes

**Physalia course 2023**

**Instructor:** Jacques Serizay

# Some vocabulary

- Everything in R is an object.

# Some vocabulary

- Everything in R is an object.

- Class = blueprint, i.e. description of a data structure (attributes, methods)

- Object = instance of a class

# Some vocabulary

- Everything in R is an object.

- Class = blueprint, i.e. description of a data structure (attributes, methods)

- Object = instance of a class

We can think of *classes* like a sketch (prototype) of a house. It contains all the details about the floors, doors, windows…

House is the *object*. As, many houses can be made from a description, we can create many objects from a class.

# Examples

```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> L <- list(x = x, y = y)
> linmod <- lm(L)
```

# Examples

```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> L <- data.frame(x = x, y = y)
> linmod <- lm(L)
```

```
> class(x)
[1] "numeric”

> class(y)
[1] "numeric”

> class(L)
[1] "data.frame”

> class(linmod)
[1] "lm"
```

# Some vocabulary

- While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference (R6) class systems.
- We are going to focus on S3 and S4.

# Basic S3 classes

- logical: `` `TRUE`, `FALSE` ``
- numeric: `` `1.4` ``
- integer: `` `1L` ``
- character: `` `”Hello”` ``
- list: `` `list(…)` ``
- function: `` `function(x) {}` ``

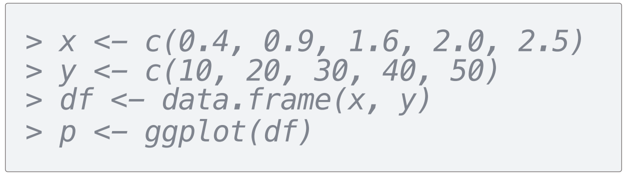
# Advanced S3 classes

- data.frames: `data.frame(…, …, …)`
- matrices: `matrix(…)`
- arrays : `array(…)`
- lm (Linear models) : `lm(…)`
- tbl : `tibble(…)`
- gg : `ggplot(…)`
- …

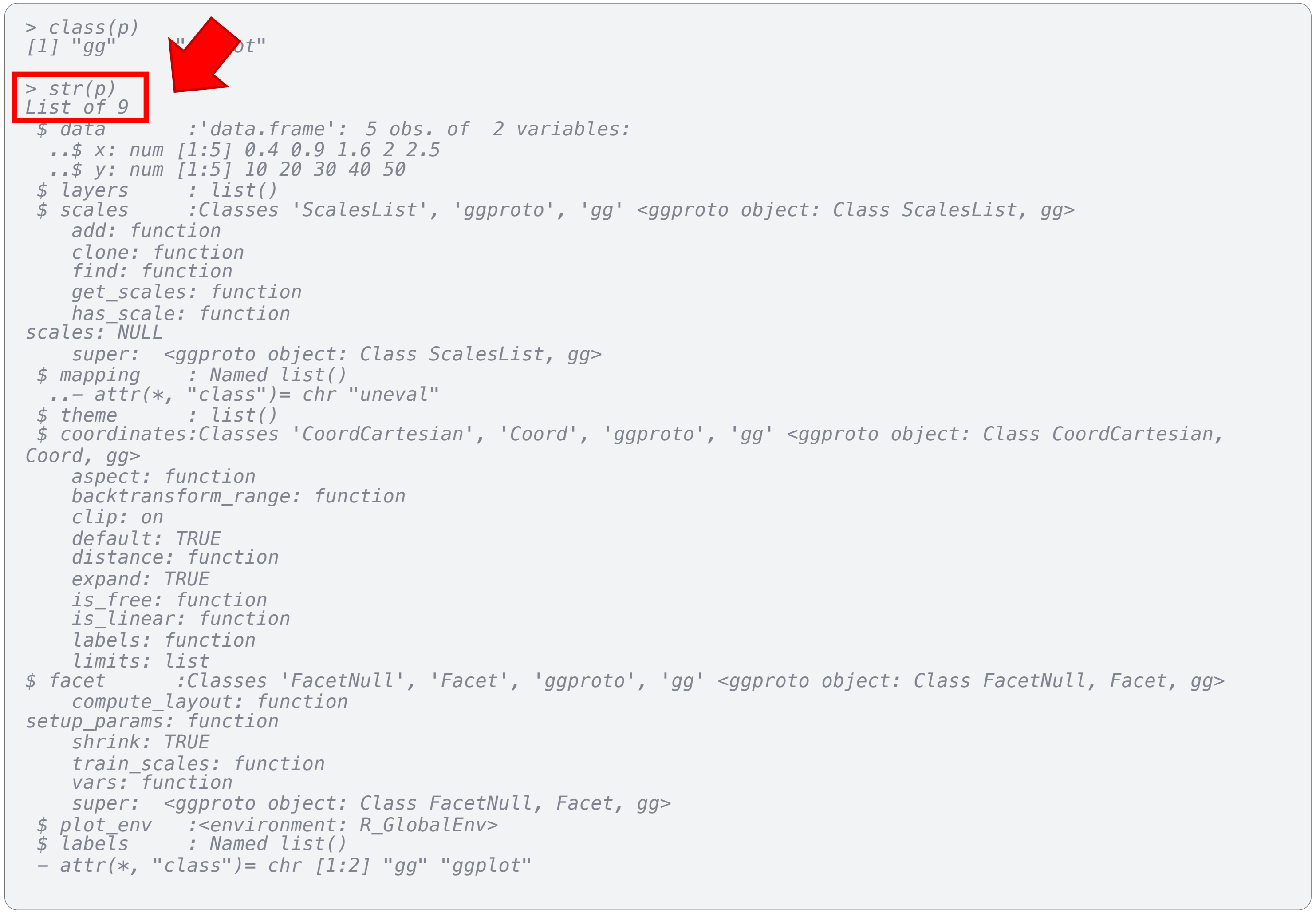
# Advanced S3 classes

- data.frames: `data.frame(…, …, …)`
- matrices: `matrix(…)`
- arrays : `array(…)`
- lm (Linear models) : `lm(…)`
- tbl : `tibble(…)`
- gg : `ggplot(…)`
- …

Advanced S3 classes are actually just lists!!



```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> df <- data.frame(x, y)
> p <- ggplot(df)
```



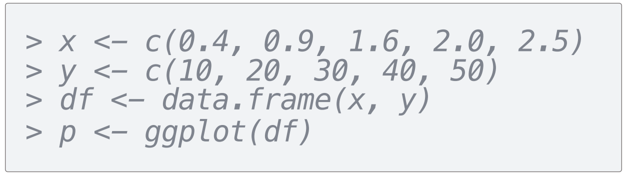
```
> class(p)
[1] "gg"     "ggplot"

> str(p)
List of 9
 $ data       :'data.frame':  5 obs. of  2 variables:
  ..$ x: num [1:5] 0.4 0.9 1.6 2 2.5
  ..$ y: num [1:5] 10 20 30 40 50
 $ layers     : list()
 $ scales     :Classes 'ScalesList', 'ggproto', 'gg' <ggproto object: Class ScalesList, gg>
    add: function
    clone: function
    find: function
    get_scales: function
    has_scale: function
scales: NULL
    super:  <ggproto object: Class ScalesList, gg>
 $ mapping    : Named list()
  ..- attr(*, "class")= chr "uneval"
 $ theme      : list()
 $ coordinates:Classes 'CoordCartesian', 'Coord', 'ggproto', 'gg' <ggproto object: Class CoordCartesian,
Coord, gg>
    aspect: function
    backtransform_range: function
    clip: on
    default: TRUE
    distance: function
    expand: TRUE
    is_free: function
    is_linear: function
    labels: function
    limits: list
$ facet      :Classes 'FacetNull', 'Facet', 'ggproto', 'gg' <ggproto object: Class FacetNull, Facet, gg>
    compute_layout: function
setup_params: function
    shrink: TRUE
    train_scales: function
    vars: function
    super:  <ggproto object: Class FacetNull, Facet, gg>
 $ plot_env   :<environment: R_GlobalEnv>
 $ labels     : Named list()
 - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

# Advanced S3 classes

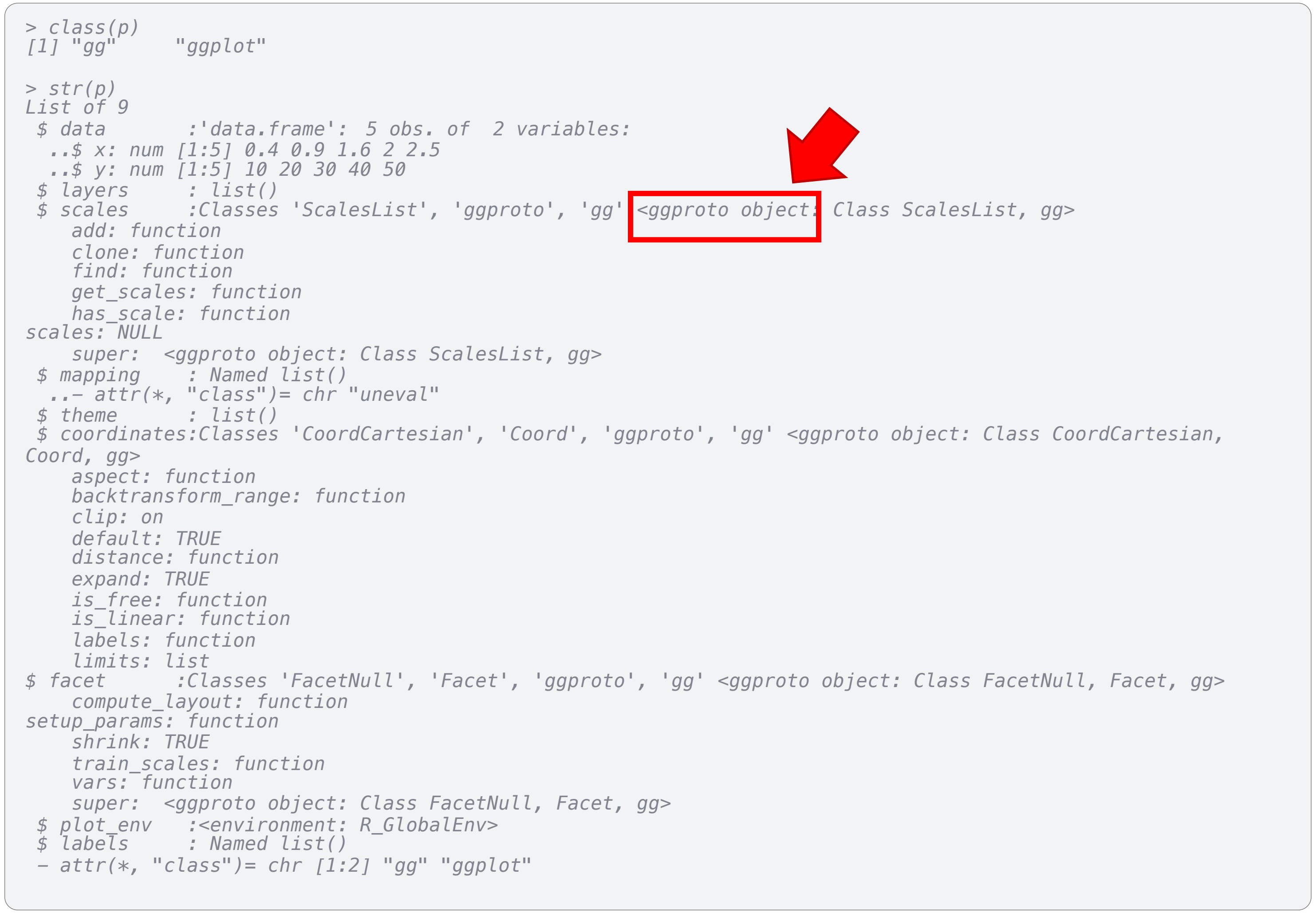
- data.frames: `data.frame(…, …, …)`
- matrices: `matrix(…)`
- arrays : `array(…)`
- lm (Linear models) : `lm(…)`
- tbl : `tibble(…)`
- gg : `ggplot(…)`
- …

Advanced S3 classes are actually just lists!!

An S3 class instance (object) does not necessarily contain only S3 class instances within its list.



```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> df <- data.frame(x, y)
> p <- ggplot(df)
```



```
> class(p)
[1] "gg"     "ggplot"

> str(p)
List of 9
 $ data       :'data.frame':  5 obs. of  2 variables:
  ..$ x: num [1:5] 0.4 0.9 1.6 2 2.5
  ..$ y: num [1:5] 10 20 30 40 50
 $ layers     : list()
 $ scales     :Classes 'ScalesList', 'ggproto', 'gg' <ggproto object: Class ScalesList, gg>
    add: function
    clone: function
    find: function
    get_scales: function
    has_scale: function
scales: NULL
    super:  <ggproto object: Class ScalesList, gg>
 $ mapping    : Named list()
  ..- attr(*, "class")= chr "uneval"
 $ theme      : list()
 $ coordinates:Classes 'CoordCartesian', 'Coord', 'ggproto', 'gg' <ggproto object: Class CoordCartesian,
Coord, gg>
    aspect: function
    backtransform_range: function
    clip: on
    default: TRUE
    distance: function
    expand: TRUE
    is_free: function
    is_linear: function
    labels: function
    limits: list
$ facet      :Classes 'FacetNull', 'Facet', 'ggproto', 'gg' <ggproto object: Class FacetNull, Facet, gg>
    compute_layout: function
setup_params: function
    shrink: TRUE
    train_scales: function
    vars: function
    super:  <ggproto object: Class FacetNull, Facet, gg>
 $ plot_env   :<environment: R_GlobalEnv>
 $ labels     : Named list()
 - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

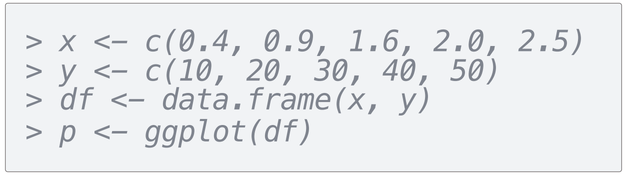
# Advanced S3 classes

- data.frames: `data.frame(…, …, …)`
- matrices: `matrix(…)`
- arrays : `array(…)`
- lm (Linear models) : `lm(…)`
- tbl : `tibble(…)`
- gg : `ggplot(…)`
- …

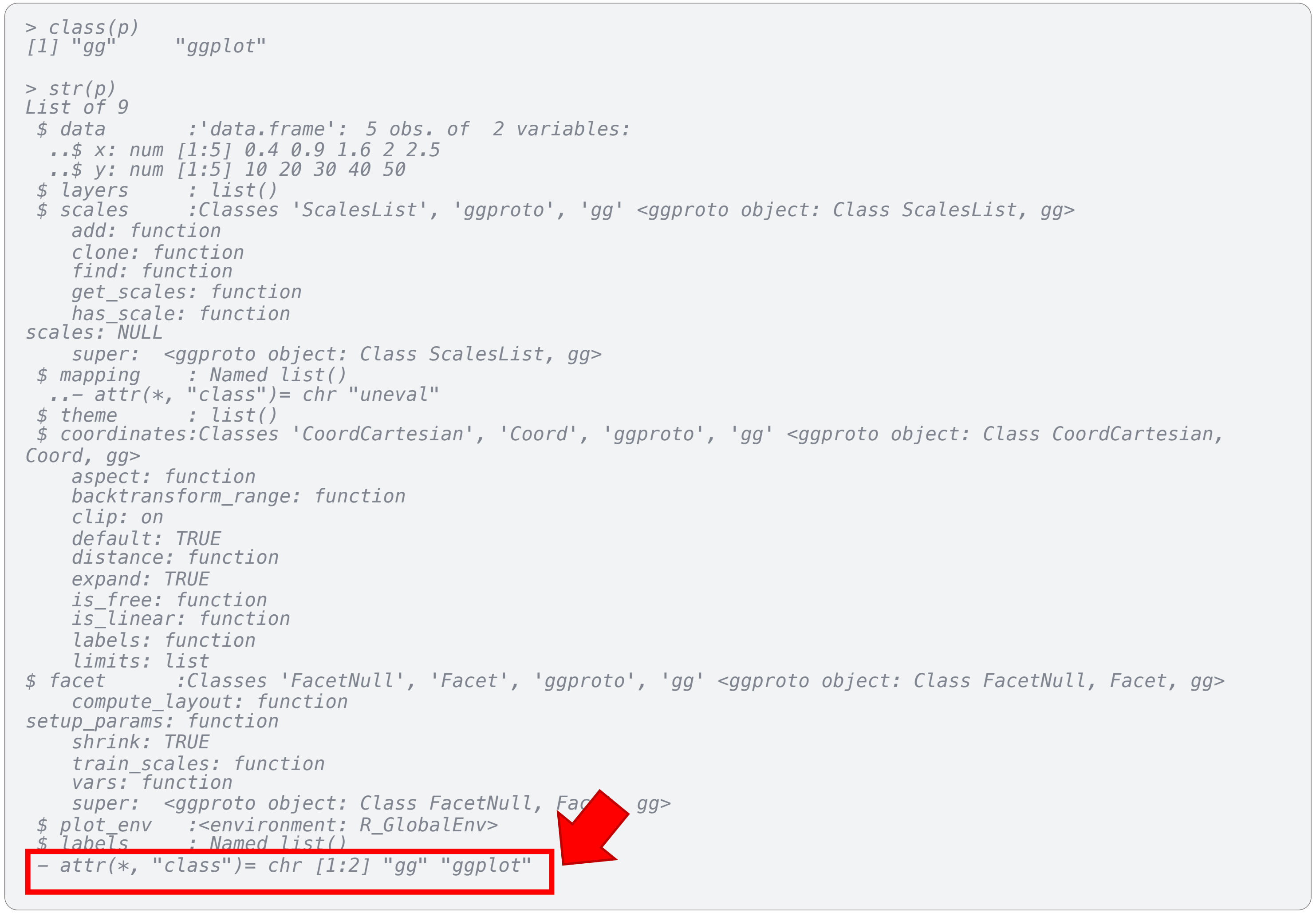
Advanced S3 classes are actually just lists!!

An S3 class instance (object) does not necessarily contain only S3 class instances within its list.

An instance of an S3 class can be created by simply adding a "class" attribute!



```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> df <- data.frame(x, y)
> p <- ggplot(df)
```



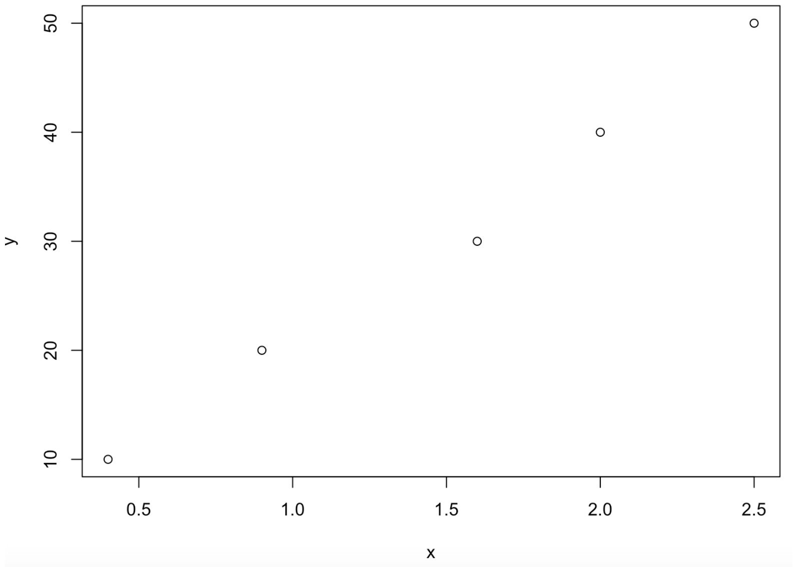
```
> class(p)
[1] "gg"     "ggplot"

> str(p)
List of 9
 $ data       :'data.frame':  5 obs. of  2 variables:
  ..$ x: num [1:5] 0.4 0.9 1.6 2 2.5
  ..$ y: num [1:5] 10 20 30 40 50
 $ layers     : list()
 $ scales     :Classes 'ScalesList', 'ggproto', 'gg' <ggproto object: Class ScalesList, gg>
    add: function
    clone: function
    find: function
    get_scales: function
    has_scale: function
scales: NULL
    super:  <ggproto object: Class ScalesList, gg>
 $ mapping    : Named list()
  ..- attr(*, "class")= chr "uneval"
 $ theme      : list()
 $ coordinates:Classes 'CoordCartesian', 'Coord', 'ggproto', 'gg' <ggproto object: Class CoordCartesian,
Coord, gg>
    aspect: function
    backtransform_range: function
    clip: on
    default: TRUE
    distance: function
    expand: TRUE
    is_free: function
    is_linear: function
    labels: function
    limits: list
$ facet      :Classes 'FacetNull', 'Facet', 'ggproto', 'gg' <ggproto object: Class FacetNull, Facet, gg>
    compute_layout: function
setup_params: function
    shrink: TRUE
    train_scales: function
    vars: function
    super:  <ggproto object: Class FacetNull, Fac      gg>
 $ plot_env   :<environment: R_GlobalEnv>
 $ labels     : Named list()
 - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

# Creating your own S3 class

An instance of an S3 class can be created by simply adding a “class” attribute.

→ **This makes the integrity of S3 objects rather fragile!**

```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> df <- data.frame(x, y)
> class(df)
[1] "data.frame"

> attributes(df)
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4 5

> df
    x  y
1 0.4 10
2 0.9 20
3 1.6 30
4 2.0 40
5 2.5 50

> plot(df)
```



# Creating your own S3 class

An instance of an S3 class can be created by simply adding a “class” attribute.

→ **This makes the integrity of S3 objects rather fragile!**

```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> df <- data.frame(x, y)
> class(df)
[1] "data.frame"

> attributes(df)
$names
[1] "x" "y"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4 5

> df
    x  y
1 0.4 10
2 0.9 20
3 1.6 30
4 2.0 40
5 2.5 50

> plot(df)
```

```
> attr(df, 'class') <- 'lm'
> class(df)
[1] ”lm"
```

```
> df

Call:
NULL

No coefficients


> plot(df)
Error in x$terms %||% attr(x, "terms") %||%
stop("no terms component nor attribute") :
  no terms component nor attribute
```

# Creating your own S3 class

To avoid this in package development, one typically relies on “**constructor**” functions

```
growth <- function(ID, sex, age, len){
   out <- list(
     ID = ID,
     sex = sex,
     data = data.frame(age = age, len =
     len)
   )
   class(out) <- "growth"
   return(out)
}
```

# Creating your own S3 class

To avoid this in package development, one typically relies on “**constructor**” functions

```
growth <- function(ID, sex, age, len){
   out <- list(
      ID = ID,
      sex = sex,
      data = data.frame(age = age, len =
      len)
   )
   class(out) <- "growth"
   return(out)
}
```

```
> alice <- growth(”Alice", ”Female", 0:10,
(0:10)^2)

> class(alice)
[1] "growth"

> alice
$ID
[1] "Alice"

$sex
[1] "Female"

$data
   age len
1    0   0
2    1   1
3    2   4
4    3   9
5    4  16
6    5  25
7    6  36
8    7  49
9    8  64
10   9  81
11  10 100

attr(,"class")
[1] "growth"
```

# S4 classes to the rescue!

S4 works very similarly to S3 except it has formal definitions of classes. These definitions describe the class “fields” and structures.

# S4 classes to the rescue!

S4 works very similarly to S3 except it has formal definitions of classes. These definitions describe the class “fields” and structures.

- S4 objects fields are called “**slots**” (they are elements of a **list-like** object)

# S4 classes to the rescue!

S4 works very similarly to S3 except it has formal definitions of classes. These definitions describe the class “fields” and structures.

- S4 objects fields are called “**slots**” (they are elements of a **list-like** object)

- S4 classes have strictly required properties:

    - Name. A class identifier, which should be UpperCamelCase (by convention).

    - Slots. A named list of names and permitted classes for slots. For instance with growth we might have list(ID="character", age="numeric”).

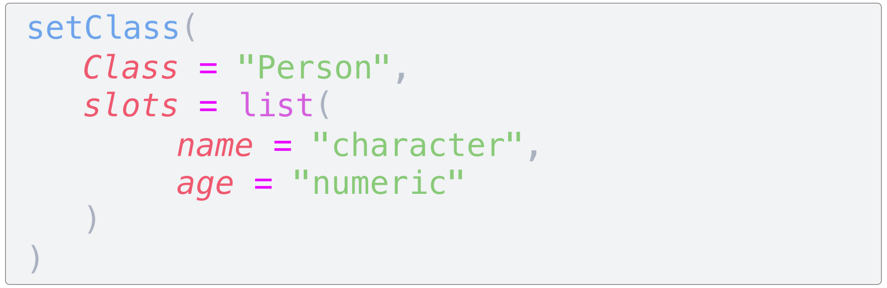
    - Contains. A string explicitly giving the classes it inherits from (i.e. contains).

    - Validity. A method that tests if an object is valid.

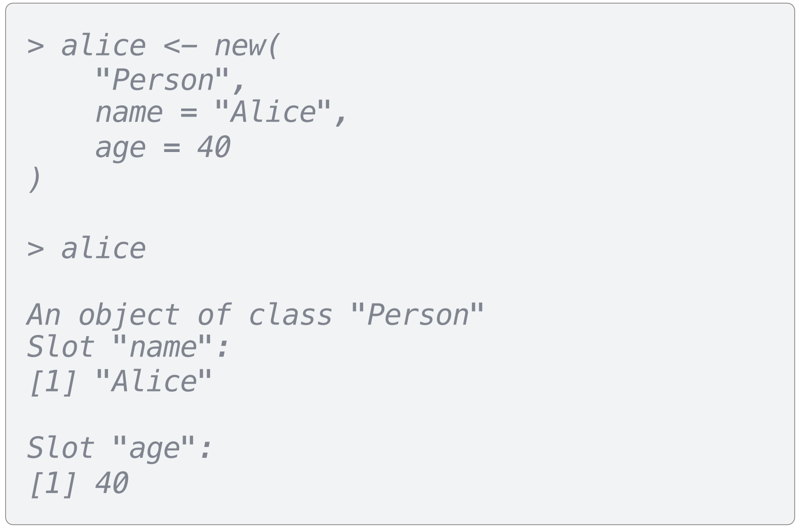
    - Prototype. A method that defines default slot values.

# S4 classes to the rescue!

methods::new() is used to create a new object with an S4 class



```
setClass(
   Class = "Person",
   slots = list(
        name = "character",
        age = "numeric"
   )
)
```



```
> alice <- new(
    "Person",
    name = "Alice",
    age = 40
)

> alice

An object of class "Person"
Slot "name":
[1] "Alice"

Slot "age":
[1] 40
```

# S4 classes to the rescue!

Here again, constructor function should be the #1 priority to write.

Generally, the constructor function bears the same name than the S4 class itself.

```r
setClass(
     Class = "Person",
     slots = list(
     name = "character",
     age = "numeric"
   )
)

Person <- function(name, age) {
   methods::new(
     "Person",
     name = name,
     age = age
   )
}
```

```
> bob <- Person(
    name = "Bob",
    age = 32
  )

> bob

An object of class "Person"
Slot "name":
[1] "Bob"

Slot "age":
[1] 32
```

# S4 classes to the rescue!

The inflexibility of S4 classes reduces errors because the information is known, and valid.

Slots
Contains

Validity

```
setClass(
     Class = "Person",
     slots = list(
     name = "character",
     age = "numeric"
   )
)

Person <- function(name, age) {
   methods::new(
     "Person",
     name = name,
     age = age
   )
}
```

```
> matthew <- Person(
    name = "Matthew",
    age = 'twelve’
)

Error in validObject(.Object) :
  invalid class “Person” object: invalid object for slot "age" in
class "Person": got class "character", should be or extend class
"numeric”

> henry <- Person(name = 'Henry', job = 'worker’)

Error in Person(name = "Henry", job = "worker") :
  unused argument (job = "worker")
```

# Some vocabulary

- Class = blueprint, i.e. description of a data structure (attributes, methods)

- Object = instance of a *class*

# Some vocabulary

- Class = blueprint, i.e. description of a data structure (attributes, methods)

- Object = instance of a *class*

- Method = function that only operates on certain *classes*

# Some vocabulary

- Class = blueprint, i.e. description of a data structure (attributes, methods)

- Object = instance of a *class*

- Method = function that only operates on certain *classes*

2 objects: 1 of class *house* and 1 of class *canvas*.

→ The “paint” *method* will not do the same thing for the 2 objects: it changes the color of the *house* or paints a painting on the *canvas*).

# Some vocabulary

- Class = blueprint, i.e. description of a data structure (attributes, methods)
- Object = instance of a *class*
- Method = function that only operates on certain *classes*

2 objects: 1 of class *house* and 1 of class *canvas*.

→ The "paint" *method* will not do the same thing for the 2 objects: it changes the color of the *house* or paints a painting on the *canvas*).

→ R is *polymorphic*: a given function, when applied to objects from different classes, returns a tailored result.

# Examples

```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> L <- list(x = x, y = y)
> linmod <- lm(L)
```

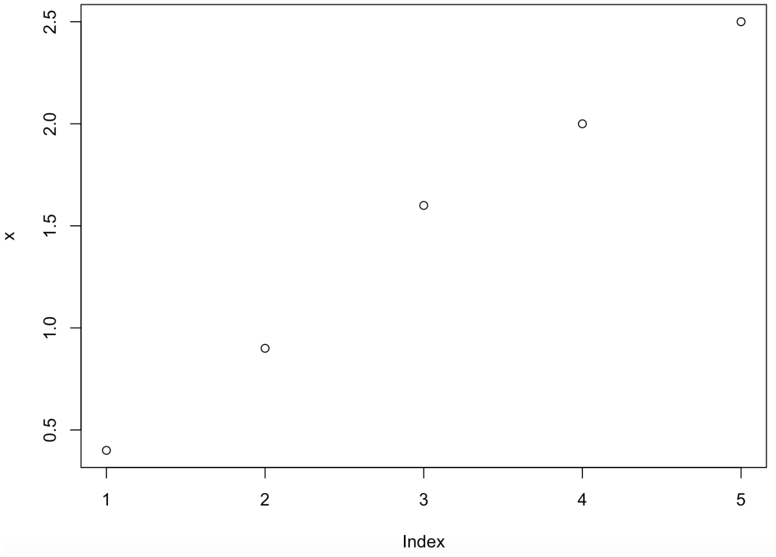
```
> class(x)
[1] "numeric”

> class(y)
[1] "numeric”

> class(L)
[1] "data.frame”

> class(linmod)
[1] "lm"
```

# Examples

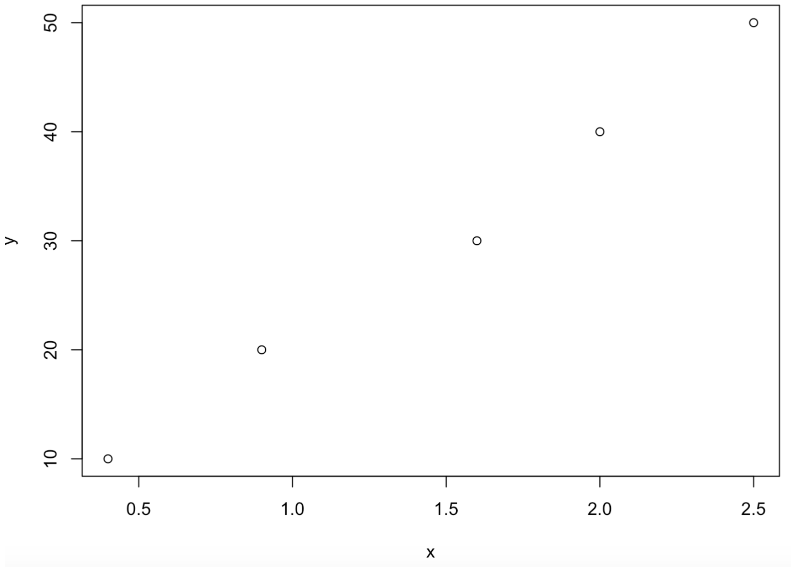
```
> x <- c(0.4, 0.9, 1.6, 2.0, 2.5)
> y <- c(10, 20, 30, 40, 50)
> L <- list(x = x, y = y)
> linmod <- lm(L)
```

```
> class(x)
[1] "numeric”

> class(y)
[1] "numeric”

> class(L)
[1] "data.frame”

> class(linmod)
[1] "lm"
```

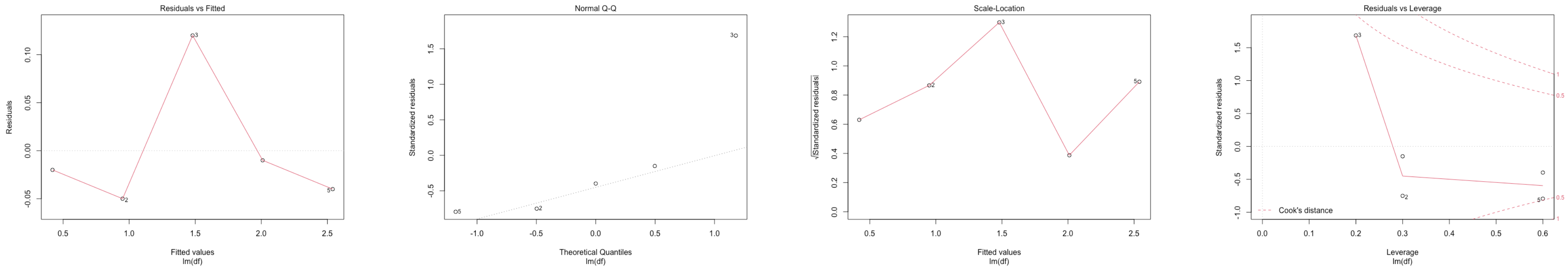
```
plot(x)
```



```
plot(L)
```



```
plot(limod)
```



# Creating S3 methods

In S3, the **generic method** detects the class type and dispatches (calls) arguments to the appropriate **class-specific method**.

Example:

- `plot` is a generic function
- `plot.lm` is a class-specific (`lm`) function

## Creating S3 methods

1. If the generic function does not exist, one need to create it:

```
myfun <- function(x) UseMethod(”myfun”)
```

2. In this case, a default method should also be created!

```
myfun.default <- function(x) “default method for `myfun`”
```

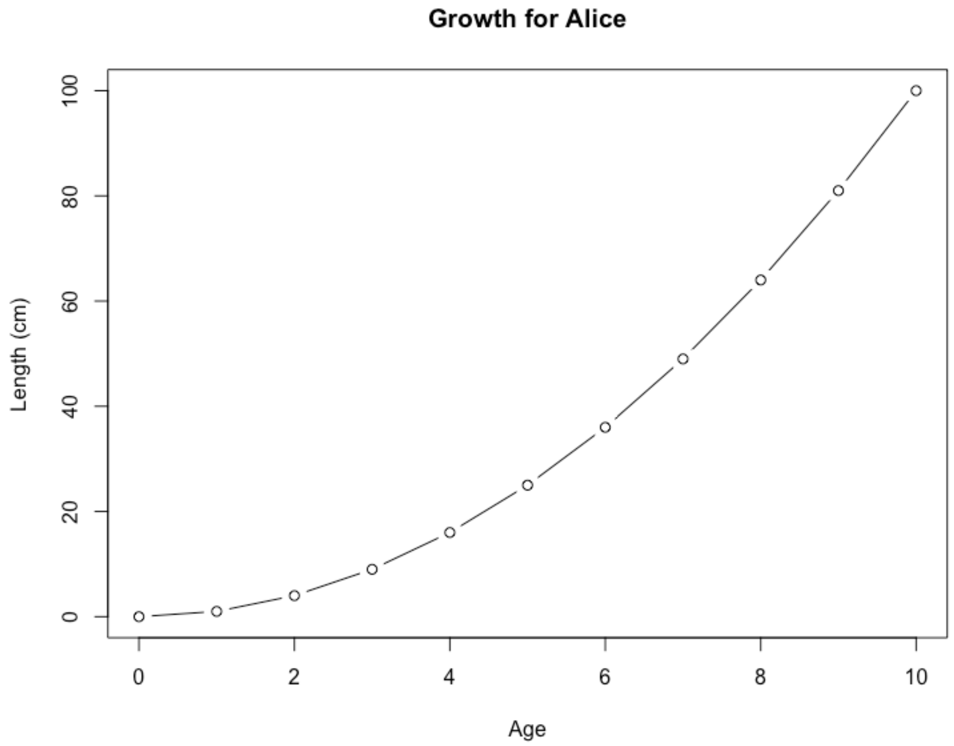
3. The class-specific method created by a developed is named <`generic>.<class>` (e.g. `plot.lm`):

```
myfun.myclass <- function(x) {…}
```

# Creating S3 methods

```r
#Class constructor
growth <- function(ID, sex, age, len){
   out <- list(
      ID = ID,
      sex = sex,
      data = data.frame(age = age, len =
      len)
   )
   class(out) <- "growth"
   return(out)
}

#1. Generic function
growthPlot <- function(x)
    UseMethod("growthPlot")

#2. Default method
growthPlot.default <- function(x)
   "Unknown class"

#3. Class-specific method
growthPlot.growth <- function(object){
    d <- object$data
    plot(
        d$age, d$len, type="b", xlab="Age",
        ylab="Length (cm)",
        main=paste("Growth for", object$ID)
    )
}
```

```
> alice <- growth(”Alice", ”Female", 0:10,
(0:10)^2)

> growthPlot(alice)
```



# Creating S4 methods

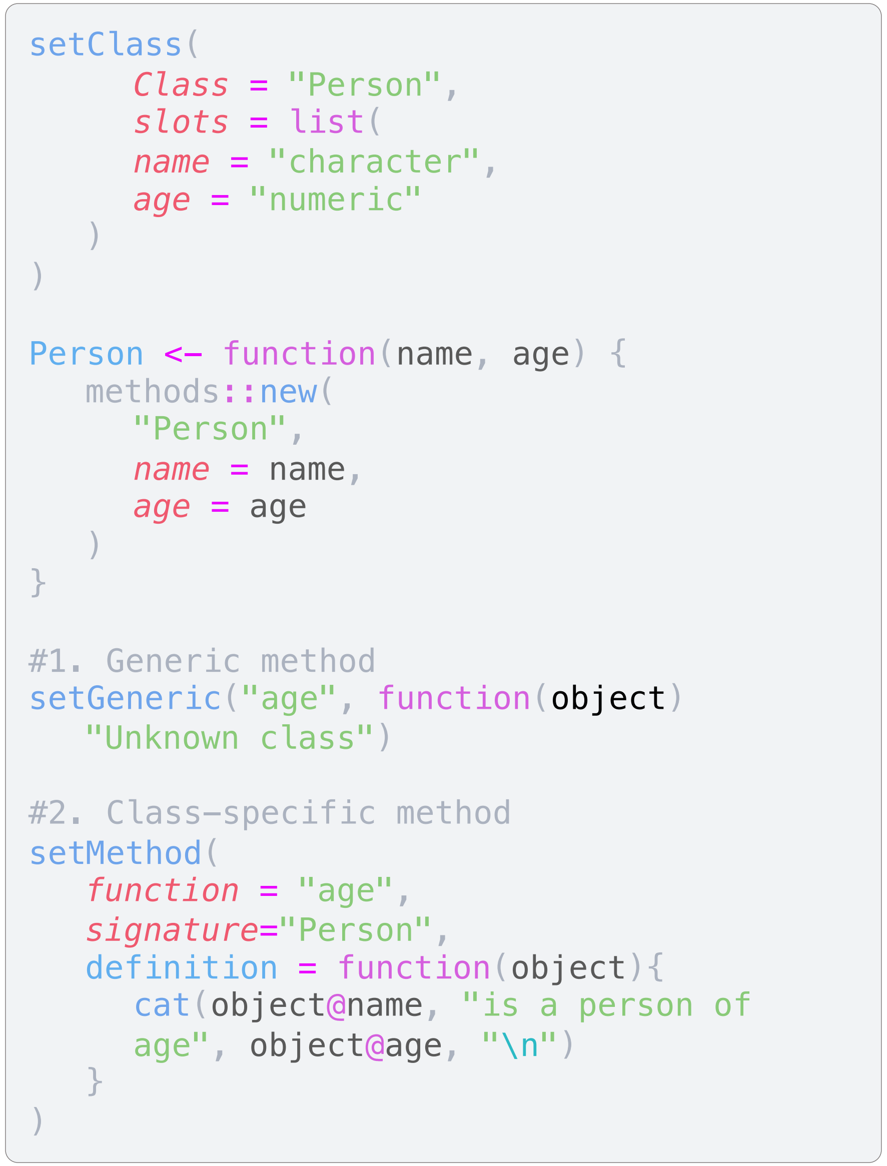
S4 methods behave very similarly to S3 methods. How methods are initiated is slightly different.

1. Just like for S3 methods, a generic method is required for any function. If it does not already exist (though it is likely!), one need to set it, with **setGeneric** (instead of <generic> in S3).

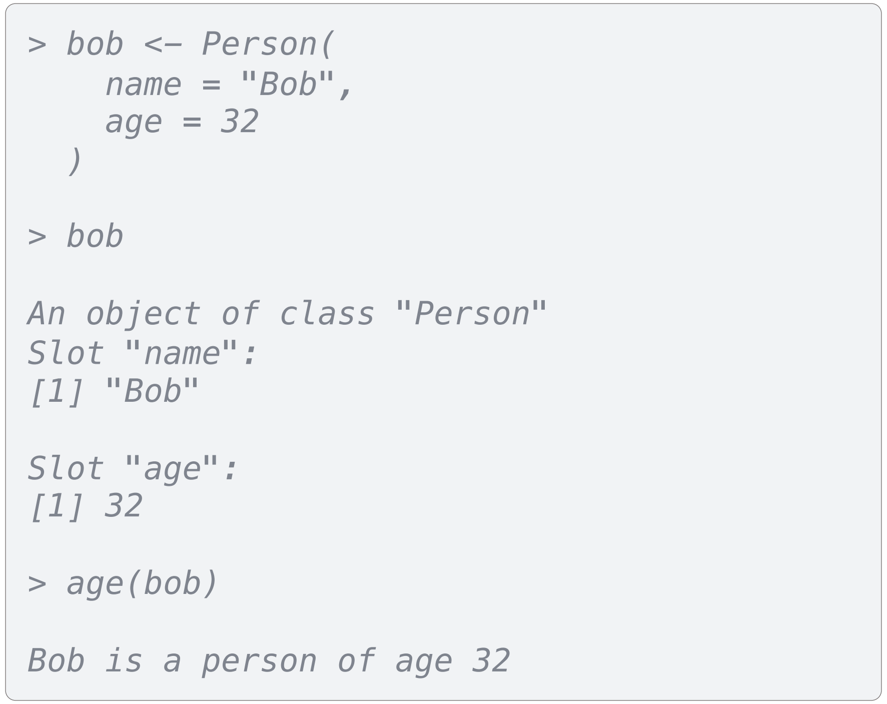
2. There is no need to set a default method! The generic will be used if no matching class-specific method is found.

3. The **setMethod** function is used to create a method for a specific class (instead of <generic>.<class> in S3).

# Creating S4 methods



```
setClass(
      Class = "Person",
      slots = list(
      name = "character",
      age = "numeric"
   )
)

Person <- function(name, age) {
   methods::new(
      "Person",
      name = name,
      age = age
   )
}

#1. Generic method
setGeneric("age", function(object)
   "Unknown class")

#2. Class-specific method
setMethod(
   function = "age",
   signature="Person",
   definition = function(object){
      cat(object@name, "is a person of
      age", object@age, "\n")
   }
)
```



```
> bob <- Person(
    name = "Bob",
    age = 32
  )

> bob

An object of class "Person"
Slot "name":
[1] "Bob"

Slot "age":
[1] 32

> age(bob)

Bob is a person of age 32
```

# Fundamental concepts of R classes

- Constructor: A function which generates objects of a S3/S4 class.

- Polymorphism: Same function call leads to different operations for objects of different classes.

- Generic Function: A function which behaves differently depending on the class of the argument.

- Dispatch: The process of determining the correct method to call based on the argument type

# Further reading on S4 classes & methods

https://bioconductor.org/help/course-materials/2017/Zurich/S4-classes-and-methods.html