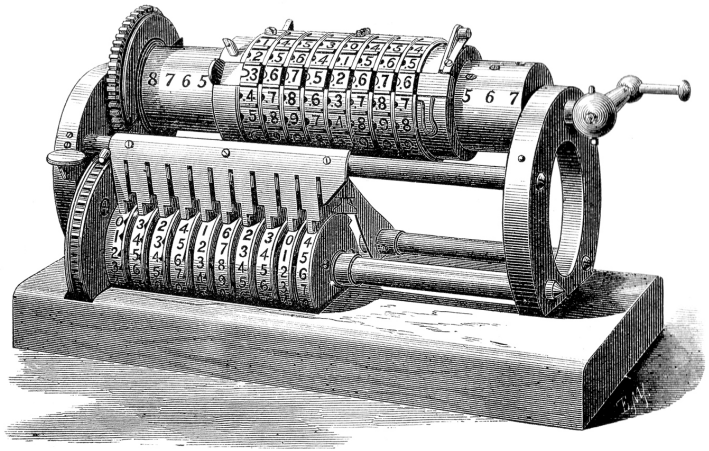
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

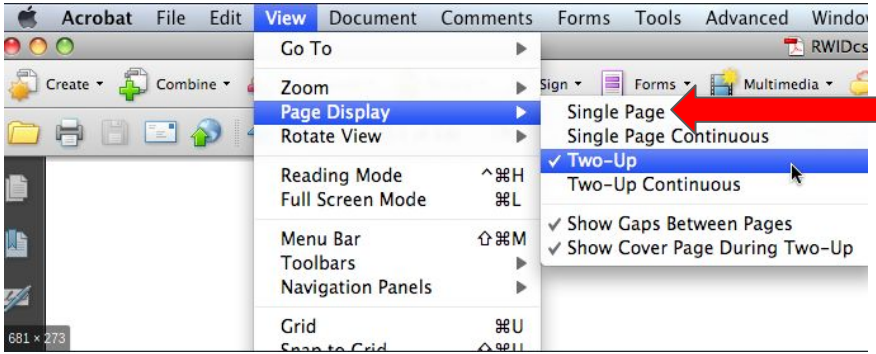
Programming in R

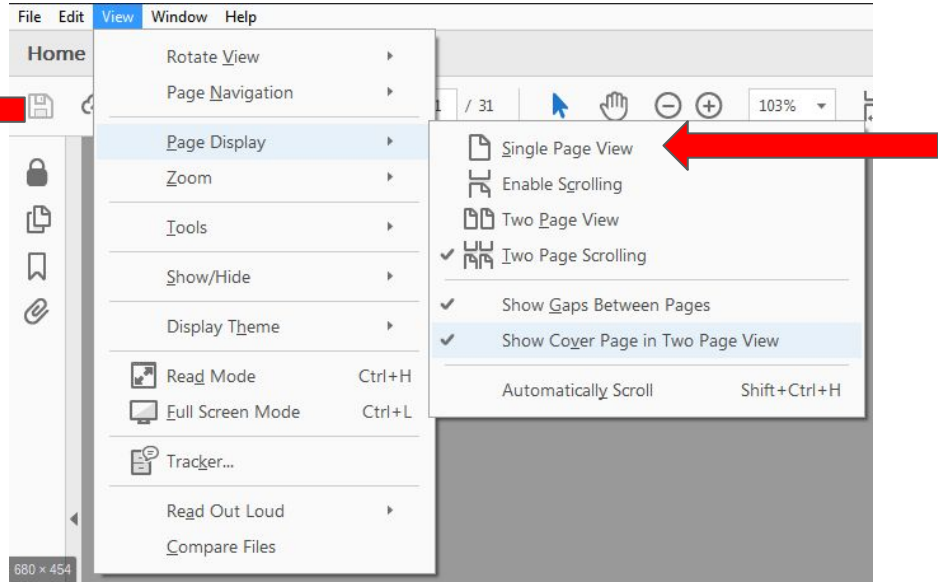


Unit 10: Object Oriented Programming in R (I)

About these Slides

The best way to view these slides (and to get the most out of the animations) is to view it in "presentation mode" or "single page view". See the images below where to set up single page view in your system.





Object Oriented Programming in R

Remember, this course has multiple goals:

- Learn things about the R language: "R"
- Get to know nice tools to use: "Tools"
- Learn things about software development in general: "Dev"

This unit:

- "Dev" Track: Object-Oriented Programming
- "Tools" Track: R6 Objects
- "R" Track: S3 Objects
- "Dev" Track: Software Design Patterns

Dev Track
Object-Oriented Programming

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. linear model

```
> model <- lm(speed ~ dist, cars)
> print(model)

Call:
lm(formula = speed ~ dist, data = cars)

Coefficients:
(Intercept)         dist
     8.2839       0.1656
```

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

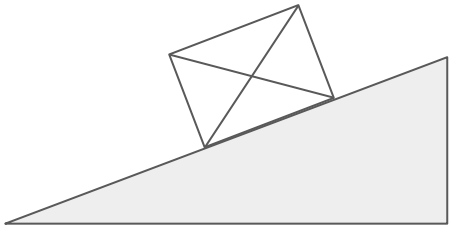
- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data

```
> df <- data.frame(a = c(1, 2, 3), b = c(-1, -2, -3))
> df
  a  b
1 1 -1
2 2 -2
3 3 -3
```

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data
  - a block in a physics simulation



Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data
  - a block in a physics simulation
- You can "do things" with that "something" (without having to know how it works internally)
  - e.g. make predictions with the model

```
> predict(model, cars[1, ])
       1
8.615041
```

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

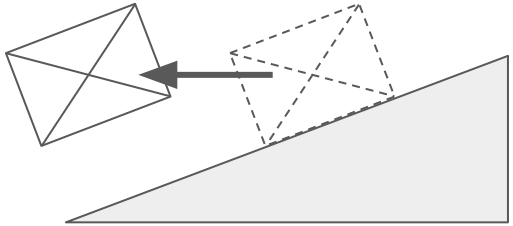
- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data
  - a block in a physics simulation
- You can "do things" with that "something" (without having to know how it works internally)
  - e.g. make predictions with the model
  - add another column to data.frame

```
> cbind(df, c = 99)
  a  b  c
1 1 -1 99
2 2 -2 99
3 3 -3 99
```

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data
  - a block in a physics simulation
- You can "do things" with that "something" (without having to know how it works internally)
  - e.g. make predictions with the model
  - add another column to data.frame
  - change the placement coordinates of a simulated block



Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. linear model
  - data.frame of some data
  - a block in a physics simulation
- You can "do things" with that "something" (without having to know how it works internally)
  - e.g. make predictions with the model
  - add another column to data.frame
  - change the placement coordinates of a simulated block
- Often "doing things" involves calling a function
  - Many programming languages: `obj.doStuff(parameter, parameter)`
  - R: `doStuff(obj, parameter, parameter)`

Object-Oriented Programming

Before we go into history, definitions and all that nerdy stuff, what is OOP "like"?

- you have "something" representing a state or collection of data
  - e.g. line
  - data.fra ta
  - a block in a simulation
- You can "do things" with that "something" (without having to know how it works in
  - e.g ons with the model
  - add a umn to data.frame
  - change the placement coordinates of a simulated block
- Often "doing things" involves calling a function
  - Many programming languages: `obj.doStuff(parameter, parameter)`
  - R: `doStuff(obj, parameter, parameter)`

Objects

Methods

Object-Oriented Programming

Object-oriented programming

From Wikipedia, the free encyclopedia

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or *properties*), and the code is in the form of procedures (often known as *methods*).

From Wikipedia, the Free Encyclopedia (link).

* The Cool People pronounce it like the "oop"-part in "loop" or "oops", by the way.

Object-Oriented Programming

Object-oriented programming

From Wikipedia, the free encyclopedia

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or *properties*), and the code is in the form of procedures (often known as *methods*).

From Wikipedia, the Free Encyclopedia (link).

Stands, to some degree, in contrast with other "*programming paradigms*", most prominently:

- **Procedural programming**: Organizing code into functions ("procedures", "subroutines"). Remember the "Modular Programming" slides. OOP *also* organizes code into functions, but groups these functions together with the data they are primarily meant to work with / manipulate.
- **Functional programming**: Treating functions as variables themselves (a.k.a. "first class functions") and expecting functions to not have "side effects": They are "expressions" and should "return a value" (always the same value for the same arguments) they are not "commands" that "do something". Using "`print()`" (has side-effect of creating output) or "`readline()`" (value depends on user input) would not be considered "functional programming", even though they involve functions.

Object-Oriented Programming

Object-oriented programming

From Wikipedia, the free encyclopedia

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or *properties*), and the code is in the form of procedures (often known as *methods*).

From Wikipedia, the Free Encyclopedia (link).

Stands, to some degree, in contrast with other "*programming paradigms*", most prominently:

- **Procedural programming**
- **Functional programming**

These paradigms ("Object-oriented", "Functional" etc.) mean different things in different contexts:

- How code is written to solve a specific problem (e.g. whether it uses "objects", whether it uses functions as variables)
- Whether, and to what degree, a language supports or even enforces a certain way of solving a problem. One would consider R an Object-oriented, a functional, and a procedural language since it supports all of these to a large degree.

Paradigms are not mutually exclusive and can come in varying degrees of strictness. We are not going to go crazy about definitions here.

OOP in R

R supports object oriented programming in multiple ways:

- S3 objects (named after programming language S, the predecessor of R, version 3)
- S4 objects (named after S version 4, where they were introduced)
- Reference classes, rarely also called "R5" to keep up with the naming scheme
- R6, an external R package similar to reference classes, named to keep up with the naming scheme
- R7, a system currently (2023) under development, named to keep up with the naming scheme*

* "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

OOP in R

R supports object oriented programming in multiple ways:

- S3 objects (named after programming language S, the predecessor of R, version 3)
- S4 objects (named after S version 4, where they were introduced)
- Reference classes, rarely also called "R5" to keep up with the naming scheme
- R6, an external R package similar to reference classes, named to keep up with the naming scheme
- R7, a system currently (2023) under development, named to keep up with the naming scheme*

These mostly differ on two axes:

- Whether they have copy semantics (like most of base R) vs. reference semantics (e.g. like `data.table`'s "`:=`" operation)
- Whether they are relatively formal, explicit and strict with class structure and type checking

* "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

OOP in R

R supports object oriented programming in multiple ways:

- S3 objects (named after programming language S, the predecessor of R, version 3)
- S4 objects (named after S version 4, where they were introduced)
- Reference classes, rarely also called "R5" to keep up with the naming scheme
- R6, an external R package similar to reference classes, named to keep up with the naming scheme
- R7, a system currently (2023) under development, named to keep up with the naming scheme*

These mostly differ on two axes:

- Whether they have copy semantics (like most of base R) vs. reference semantics (e.g. like `data.table`'s "`:=`" operation)
- Whether they are relatively formal, explicit and strict with class structure and type checking

	Copy Semantics	**Reference Semantics**
Very Informal	S3	
Formal class structure, no type checking		R6
Formal class structure + type checking	S4, R7	Reference Classes

* "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

OOP in R

R supports object oriented programming in multiple ways:

- S3 objects (named after programming language S, the predecessor of R, version 3)
- S4 objects (named after S version 4, where they were introduced)
- Reference classes, rarely also called "R5" to keep up with the naming scheme
- R6, an external R package similar to reference classes, named to keep up with the naming scheme
- R7, a system currently (2023) under development, named to keep up with the naming scheme*

These mostly differ on two axes:

- Whether they have copy semantics (like most of base R) vs. reference semantics (e.g. like `data.table`'s "`:=`" operation)
- Whether they are relatively formal, explicit and strict with class structure and type checking

	Copy Semantics	**Reference Semantics**
Very Informal	S3	
Formal class structure, no type checking		R6
Formal class structure + type checking	S4, R7	Reference Classes

S4 also allows "multiple inheritance". S4 and R7 support "multiple dispatch". We will not talk about this more here.

* "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

OOP in R

R supports object oriented programming in multiple ways:

- S3 objects (named after programming language S, the predecessor of R, version 3)
- S4 objects (named after S version 4, where they were introduced)
- Reference classes, rarely also called "R5" to keep up with the naming scheme
- R6, an external R package similar to reference classes, named to keep up with the naming scheme
- R7, a system currently (2023) under development, named to keep up with the naming scheme*

These mostly differ on two axes:

- Whether they have copy semantics (like most of base R) vs. reference semantics (e.g. like `data.table`'s "`:=`" operation)
- Whether they are relatively formal, explicit and strict with class structure and type checking

	Copy Semantics	**Reference Semantics**
Very Informal, implicit structure	S3	
Formal / explicit class structure, no type checking		R6
Formal / explicit class structure + type checking	S4, R7	Reference Classes

S4 also allows "multiple inheritance". S4 and R7 support "multiple dispatch". We will not talk about this more here.

➔ Whether you use *copy semantics* or *reference semantics* depends on what kind of object you want to represent / what problem you are solving. E.g. there is no reason not to use both S3 and R6 in the same project at the same time for different purposes.

➔ Whether you use R6 vs. Reference Classes, and whether you use S3 vs. S4 vs. R7, depends on your preferences. It would be unusual to mix R6 with reference classes or to mix S4 with R7 in the same project.

➔ R6 is more popular than reference classes. S4 is popular in some spaces (bioconductor) but S3 is more widely spread in general. R7 is too new to judge (as of 2023).

* "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

Tools Track

R6 Objects

R6 -- Intro

- R6 objects' behaviour is more similar to other widespread OOP frameworks (e.g. Python), so we will treat it first here.
- R6 is *not* part of R itself, it is a library that you would usually install and load

```
> install.packages("R6")
* installing *source* package ‘R6’ ...
# [...]
* DONE (R6)

The downloaded source packages are in
        ‘/tmp/RtmpuNN3qV/downloaded_packages’
> library("R6")
```

- The introduction vignette is relatively short and quite good: https://r6.r-lib.org/articles/Introduction.html
- The following slides are a short overview, read these and the introduction and you will know everything about R6 that you need.

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.

R6Class() is part of the R6 library. You won't need this in your homework.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)
```

R6 -- Class

The "Cat" variable here is the constructor for the class. By convention, it is capitalized (and written with CamelCase, if it contains multiple words).

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)
```

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.

The "Cat" variable here is the constructor for the class. By convention, it is capitalized (and written with CamelCase, if it contains multiple words).

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unname
  )
)
```

The name of the class. This should be the same as the variable name where the constructor is stored.

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.

```r
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)
```

Notice how this is a normal function call, which produces a return value. "public" is a named argument of the "R6Class" function.

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)
```

Notice how this is a normal function call, which produces a return value. "public" is a named argument of the "R6Class" function.

We use = for assignment here, and not <-, since this part of the code is defining a named list!

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.
- The constructor is used to create the **object**, using `<constructor>$new()`.
  - The resulting object is called an **instance** of the class.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)

Minka <- Cat$new()
```

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.
- The constructor is used to create the **object**, using `<constructor>$new()`.
  - The resulting object is called an **instance** of the class.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)

Minka <- Cat$new()
```

The object

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.
- The constructor is used to create the **object**, using `<constructor>$new()`.
  - The resulting object is called an **instance** of the class.
- The object behaves like a named list in R.

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)

Minka <- Cat$new()
```

The object

```
> Minka$name
[1] "unnamed"
> Minka$name <- "Minka"
> Minka$name
[1] "Minka"
```

R6 -- Class

- At first, we create a "**class**", which is a specification for what **objects** of that class should look like.
  - resulting objects will be "instances" of that class
  - Classes are created by calling the "R6Class()" function in the R6 library.
- The R6Class()-function returns a "**Class Generator**" a.k.a. "**Constructor**"
  - The thing returned by R6Class is the "*constructor*" / "*class generator*".
  - So what is "the class"? It is more of an abstract construct here. We have "a constructor for *the class*", the resulting object is an "instance of *the class*", but there is no unique thing we can point to here and say "this is the class". The schema according to which the object is created is the class, but it lives in the platonic realm, not in your R session.
- The constructor is used to create the **object**, using `<constructor>$new()`.
  - The resulting object is called an **instance** of the class.
- The object behaves like a named list in R.
  - Except for the fact that it has *reference semantics*, but we will come to that!

```
library("R6")

Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)

Minka <- Cat$new()
```

The object

```
> Minka$name
[1] "unnamed"
> Minka$name <- "Minka"
> Minka$name
[1] "Minka"
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed"
  )
)
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Bella <- Cat$new()
> Bella$name <- "Bella"
>
> Minka$name
[1] "Minka"
> Bella$name
[1] "Bella"
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    }
  )
)
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    }
  )
)
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.
  - They can also call other methods of the same object.

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    },
    think = function() {
      cat(".oO(")
      self$speak()
    }
  )
)
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$think()
.oO(Meow, I am Minka ^._.^
```

1

2

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.
  - They can also call other methods of the same object.
- Methods can change fields of a class by modifying the "self" variable

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    },
    rename = function(name) {
      self$name <- name
    }
  )
)
```

```
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka$rename("Minka the Cat")
> Minka$speak()
Meow, I am Minka the Cat ^._.^
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.
  - They can also call other methods of the same object.
- Methods can change fields of a class by modifying the "self" variable

!!!

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    },
    rename = function(name) {
      self$name <- name
    }
  )
)
```

```
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka$rename("Minka the Cat")
> Minka$speak()
Meow, I am Minka the Cat ^._.^
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.
  - They can also call other methods of the same object.
- Methods can change fields of a class by modifying the "self" variable

!!!

Usually you can not change a variable *outside* a function by assigning to a variable *inside* that function, but R6 is different!

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    },
    rename = function(name) {
      self$name <- name
    }
  )
)
```

```
> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka$rename("Minka the Cat")
> Minka$speak()
Meow, I am Minka the Cat ^._.^
```

R6 -- Fields and Methods

- In this example, "name" is a **field** of the "Cat" class.
  - Fields are sometimes also called "properties" or "attributes" in other OOP frameworks. However, "attributes" means something completely different in R, so you should avoid the latter term here.
  - Each instance of a class has its own value for the fields
  - The totality of the values of the fields of an object represent the **state** of that object. Two objects with the same state are equal.
- Besides *fields*, a class can also have **methods**: functions associated with each object.
  - On the surface, methods behave like functions saved in a named list
  - However, methods can access the class object itself through the **self** object.
  - They can also call other methods of the same object.
- Methods can change fields of a class by modifying the "self" variable

!!!

Usually you can not change a variable *outside* a function by assigning to a variable *inside* that function, but R6 is different!

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    },
    rename = function(name) {
      self$name <- name
    }
  )
)

> Minka <- Cat$new()
> Minka$name <- "Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka$rename("Minka the Cat")
> Minka$speak()
Meow, I am Minka the Cat ^._.^
```

As an aside, it is okay for the name of the function argument be the same as the name of the field that it pertains to here. The field is always accessed through "self$name", so there is no way to get confused here.

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
```

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.
- Modifying self$<field> inside a method violates this

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

```
> Minka$name <- "Minka"
> Bella$name <- "Bella"
> f <- function(kitten) {
+   Minka$name <- "Minka the Cat"
+   kitten$name <- paste(kitten$name, "the Kitten")
+ }
> f(Bella)
```

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.
- Modifying self$<field> inside a method violates this

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

```
> Minka$name <- "Minka"
> Bella$name <- "Bella"
> f <- function(kitten) {
+   Minka$name <- "Minka the Cat"
+   kitten$name <- paste(kitten$name, "the Kitten")
+ }
> f(Bella)
```

This changes the (one existing) Minka object

If f is called with an R6 object for 'kitten', then this changes that object.

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.
- Modifying self$<field> inside a method violates this

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

```
> Minka$name <- "Minka"
> Bella$name <- "Bella"
> f <- function(kitten) {
+   Minka$name <- "Minka the Cat"
+   kitten$name <- paste(kitten$name, "the Kitten")
+ }
> f(Bella)
> Minka$name
Minka$name
[1] "Minka the Cat"
> Bella$name
[1] "Bella the Kitten"
```

This changes the (one existing) Minka object

If f is called with an R6 object for 'kitten', then this changes that object.

Both changed: one directly (Minka), one because it was passed for the 'kitten' argument.

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.
- Modifying self$<field> inside a method violates this
- This is because all fields of an R6 object have "**Reference Semantics**"
  - If a variable refers to an R6 object, that object exists only once in R's memory
  - Changing that object changes all copies of it

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

```
> Minka$name <- "Minka"
> Bella$name <- "Bella"
> f <- function(kitten) {
+   Minka$name <- "Minka the Cat"
+   kitten$name <- paste(kitten$name, "the Kitten")
+ }
> f(Bella)
> Minka$name
Minka$name
[1] "Minka the Cat"
> Bella$name
[1] "Bella the Kitten"
```

This changes the (one existing) Minka object

If f is called with an R6 object for 'kitten', then this changes that object.

Both changed: one directly (Minka), one because it was passed for the 'kitten' argument.

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

R6 -- Reference Semantics

- Usually assigning to a value inside a function only changes the value inside the function, not outside of it
  - "Copy Semantics": values are copied before being changed.
  - Exception we have seen so far: some data.table operations.
- Modifying self$<field> inside a method violates this
- This is because all fields of an R6 object have "**Reference Semantics**"
  - If a variable refers to an R6 object, that object exists only once in R's memory
  - Changing that object changes all copies of it

```
> x <- list(name = "Minka")
> y <- list(name = "Bella")
> f <- function(z) {
+   x$name <- "Minka the Cat"
+   z$name <- paste(z$name, "the Kitten")
+ }
> f(y)
> x$name
[1] "Minka"
> y$name
[1] "Bella"
```

does not change the value of 'x' outside of f()!

If z is not an R6 object, then this does not change the variable passed as z!

Both unchanged

```
> Minka$name <- "Minka"
> Bella$name <- "Bella"
> f <- function(kitten) {
+   Minka$name <- "Minka the Cat"
+   kitten$name <- paste(kitten$name, "the Kitten")
+ }
> f(Bella)
> Minka$name
Minka$name
[1] "Minka the Cat"
> Bella$name
[1] "Bella the Kitten"
```

This changes the (one existing) Minka object

If f is called with an R6 object for 'kitten', then this changes that object.

Both changed: one directly (Minka), one because it was passed for the 'kitten' argument.

```
> Minka.copy <- Mink
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

Minka.copy refers to the *same* object as Minka

This changes Minka, since Minka and Minka.copy are the same.

R6 -- Reference Semantics and Cloning

- Sometimes you actually want a copy of an object that you can modify independently.

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

R6 -- Reference Semantics and Cloning

- Sometimes you actually want a copy of an object that you can modify independently.
- -> Use the "`$clone()`"-method.
  - $clone() creates a "shallow copy": if the object contains another R6 object in one of its fields, it will *not* be cloned itself.
  - $clone(deep = TRUE) creates a "deep copy": if a field contains R6 objects, then these will be (deeply) cloned as well.
  - -> You should usually use $clone(deep = TRUE) unless you (1) have good reason not to and (2) know the object does not require it, and that it won't require it in the future.
  - The fact that $clone(deep = TRUE) is not the default is bad design on the part of R6.

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

```
> Minka.clone <- Minka$clone(deep = TRUE)
> Minka$name <- "Minka"
> Minka.clone$name <- "the other Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka.clone$speak()
Meow, I am the other Minka ^._.^
```

R6 -- Reference Semantics and Cloning

- Sometimes you actually want a copy of an object that you can modify independently.
- -> Use the "`$clone()`"-method.
  - $clone() creates a "shallow copy": if the object contains another R6 object in one of its fields, it will *not* be cloned itself.
  - $clone(deep = TRUE) creates a "deep copy": if a field contains R6 objects, then these will be (deeply) cloned as well.
  - -> You should usually use $clone(deep = TRUE) unless you (1) have good reason not to and (2) know the object does not require it, and that it won't require it in the future.
  - The fact that $clone(deep = TRUE) is not the default is bad design on the part of R6.

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

```
> Minka.clone <- Minka$clone(deep = TRUE)
> Minka$name <- "Minka"
> Minka.clone$name <- "the other Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka.clone$speak()
Meow, I am the other Minka ^._.^
```

WIKIPEDIA

CC (cat)

CC, for "CopyCat" *or* "Carbon Copy"[1] (December 22, 2001 – March 3, 2020), was a brown tabby and white domestic shorthair and the first cloned pet.[2]

CC

CC the first cloned cat, age 2, with her owner, Shirley Kraemer, in College Station, Texas

Breed	Domestic shorthair
Born	December 22, 2001 College Station, Texas
Known for	First cloned pet

R6 -- Reference Semantics and Cloning

- Sometimes you actually want a copy of an object that you can modify independently.
- -> Use the "`$clone()`"-method.
  - $clone() creates a "shallow copy": if the object contains another R6 object in one of its fields, it will *not* be cloned itself.
  - $clone(deep = TRUE) creates a "deep copy": if a field contains R6 objects, then these will be (deeply) cloned as well.
  - -> You should usually use $clone(deep = TRUE) unless you (1) have good reason not to and (2) know the object does not require it, and that it won't require it in the future.
  - The fact that $clone(deep = TRUE) is not the default is bad design on the part of R6.
- Check if two variables refer to the same R6-object using identical()
  - (Note that equal R objects with copy semantics are always 'identical()')

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

```
> Minka.clone <- Minka$clone(deep = TRUE)
> Minka$name <- "Minka"
> Minka.clone$name <- "the other Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka.clone$speak()
Meow, I am the other Minka ^._.^
```

```
> identical(Minka, Minka.copy)
[1] TRUE
> identical(Minka, Minka$clone(deep = TRUE))
[1] FALSE
> identical(list(name = "Minka"), list(name = "Minka"))
[1] TRUE
```

WIKIPEDIA

CC (cat)

CC, for "CopyCat" or "Carbon Copy"[1] (December 22, 2001 – March 3, 2020), was a brown tabby and white domestic shorthair and the first cloned pet.[2]

CC

CC the first cloned cat, age 2, with her owner, Shirley Kraemer, in College Station, Texas

Breed	Domestic shorthair
Born	December 22, 2001 College Station, Texas
Known for	First cloned pet

R6 -- Reference Semantics and Cloning

- Sometimes you actually want a copy of an object that you can modify independently.
- -> Use the "`$clone()`"-method.
  - $clone() creates a "shallow copy": if the object contains another R6 object in one of its fields, it will *not* be cloned itself.
  - $clone(deep = TRUE) creates a "deep copy": if a field contains R6 objects, then these will be (deeply) cloned as well.
  - -> You should usually use $clone(deep = TRUE) unless you (1) have good reason not to and (2) know the object does not require it, and that it won't require it in the future.
  - The fact that $clone(deep = TRUE) is not the default is bad design on the part of R6.
- Check if two variables refer to the same R6-object using identical()
  - (Note that equal R objects with copy semantics are always 'identical()')
  - You can check if objects are *equal* using all.equal(), but remember that it does *not* return FALSE when objects differ.

```
> Minka.copy <- Minka
> Minka$name <- "Minka"
> Minka.copy$name <- "Minka, too"
> Minka$speak()
Meow, I am Minka, too ^._.^
> Minka.copy$speak()
Meow, I am Minka, too ^._.^
```

```
> Minka.clone <- Minka$clone(deep = TRUE)
> Minka$name <- "Minka"
> Minka.clone$name <- "the other Minka"
> Minka$speak()
Meow, I am Minka ^._.^
> Minka.clone$speak()
Meow, I am the other Minka ^._.^
```

```
> identical(Minka, Minka.copy)
[1] TRUE
> identical(Minka, Minka$clone(deep = TRUE))
[1] FALSE
> identical(list(name = "Minka"), list(name = "Minka"))
[1] TRUE
```

```
> all.equal(Minka, Minka.copy)
[1] TRUE
> all.equal(Minka, Minka$clone(deep = TRUE))
[1] TRUE
> all.equal(list(name = "Minka"), list(name = "Minka"))
[1] TRUE
```

WIKIPEDIA

CC (cat)

CC, for "CopyCat" *or* "Carbon Copy"[1] (December 22, 2001 – March 3, 2020), was a brown tabby and white domestic shorthair and the first cloned pet.[2]

CC

CC the first cloned cat, age 2, with her owner, Shirley Kraemer, in College Station, Texas

Breed	Domestic shorthair
Born	December 22, 2001 College Station, Texas
Known for	First cloned pet

R6 -- Constructors

Special **`$initialize`** method: gets executed on construction when **`$new()`** is called.

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    initialize = function(name) {
      assertString(name)
      self$name = name
    },
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    }
  )
)
```

R6 -- Constructors

Special `$initialize` method: gets executed on construction when `$new()` is called.

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    initialize = function(name) {
      assertString(name)
      self$name = name
    },
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    }
  )
)
```

```
> Minka <- Cat$new(name = "Minka")
> Minka$speak()
Meow, I am Minka ^._.^
```

R6 -- Constructors

Special `$initialize` method: gets executed on construction when `$new()` is called.

- Arguments are passed along

```
Cat <- R6Class("Cat",
  public = list(
    name = "unnamed",
    initialize = function(name) {
      assertString(name)
      self$name = name
    },
    speak = function() {
      cat(sprintf("Meow, I am %s ^._.^\n", self$name))
    }
  )
)
```

```
> Minka <- Cat$new(name = "Minka")
> Minka$speak()
Meow, I am Minka ^._.^
```

R6 -- Other Topics

- Deep Cloning
- Inheritance
- Private Fields & Methods
- Active Bindings
- Finalizers

What We Expect You to Know

- **Classes** specify the structure of **Objects** which have **fields** and **methods**
- OOP in R:
  - S3: copy-semantics, implicit structure
  - R6: reference-semantics, explicit structure
  - some others

R6

- Define class:
  `R6Class(<name>, public = list(..), active = list(..), private = list(..))`
  - public: accessible from outside
  - private: accessible through special variable "`private`"
  - active: "active binding" function, looks like a field from outside
- Instantiate: `<Object Generator>$new()`
- Inheritance:
  - `R6Class(..., inherit = <superclass>, ...)`
  - Methods and fields from superclass if not overwritten
- Special variables inside methods: `self`, `private`, `super`.
- Special methods: `initialize()`, `deep_clone()`.
- Deep copy: `<Object>$clone(deep = TRUE)` , calls `deep_clone()` for all fields & methods

What We Expect You to Know

S3

- **attributes**: Additional information hanging on to objects in R
  - `"names"`: names of lists / vectors, `"dim"`: dimension of matrix / array
  - `"class"`: S3 class
  - Access through `attr(<obj>, <name>)` or `attributes(<obj>)$<name>`
  - Set conveniently with `<-` or `structure(<obj>, <name> = <value>, ...)`
- Create S3-object by setting `"class"` attribute
- Define class: Constructor function
  `<ClsName> <- function(..) { .. structure(list(..), class = "<ClsName>" ) }`
- **Generic** function: `<fname> <- function(...) UseMethod("<fname>")`
- **S3 Method**: `<fname>.<ClsName> <- function(...) { .. }`
  - Should have compatible signature (i.e. arguments) with generic
- Special method `print.<ClsName> <- function(x, ...)`
  - called automatically when object is displayed.
  - should have "`x`" and "`...`" arguments and must return `invisible(x)`.
- Inheritance through multiple entries in `"class"` attribute vector. Subclass first, superclass next.
- `NextMethod()` : call to superclass method
- assert through `assertClass(<obj>, "<ClsName>")` and assume internal structure is valid