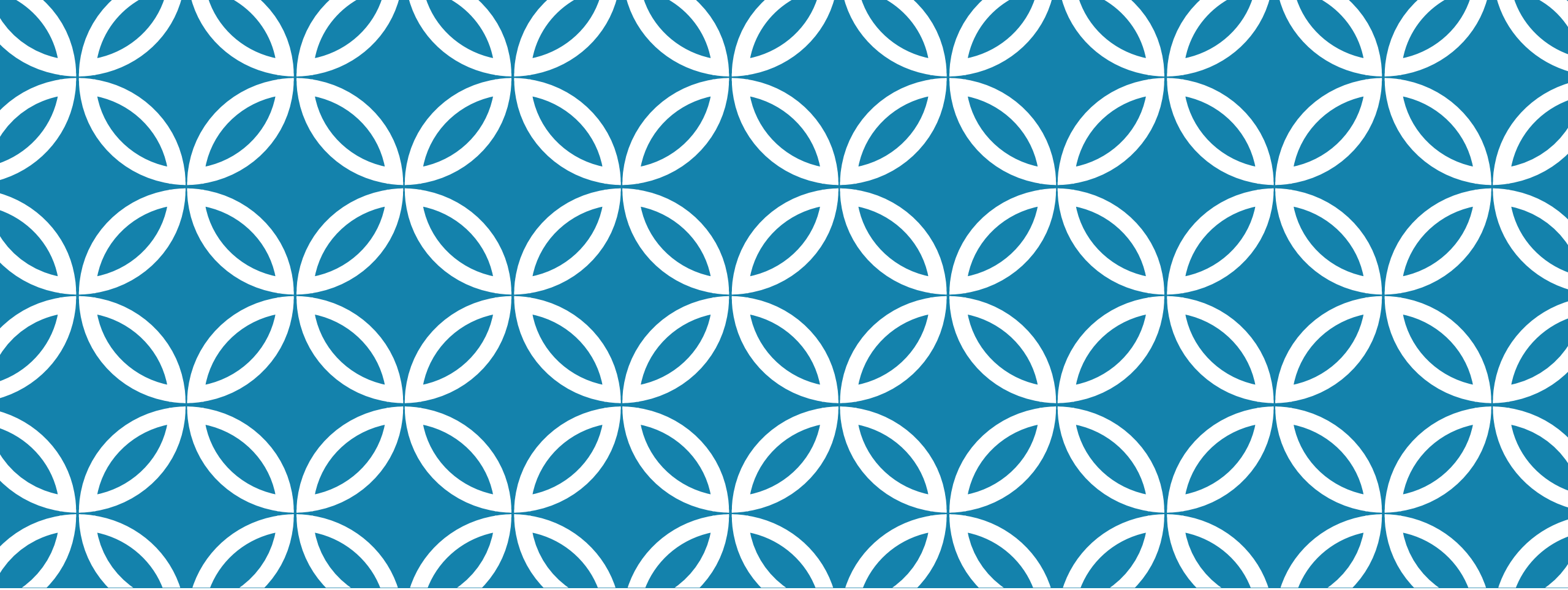


OBJECT ORIENTED PROGRAMMING

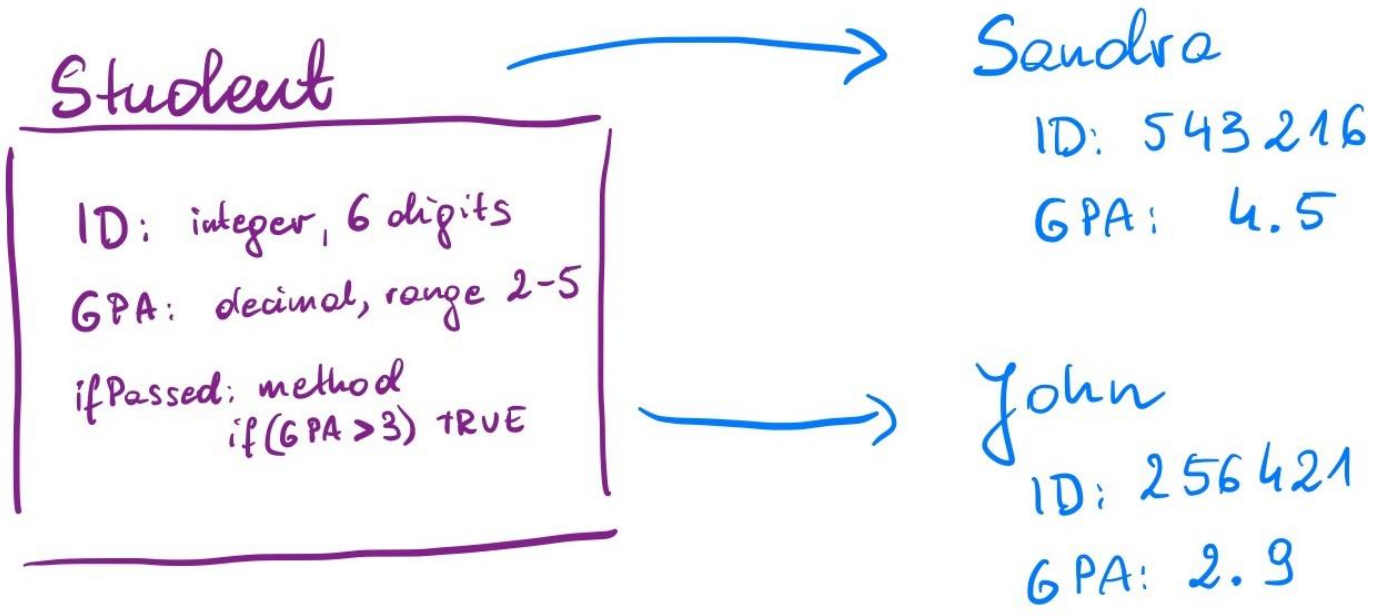
dr Maria Kubara, WNE UW

OBJECT ORIENTED PROGRAMMING (OOP)

- OOP is a backbone of many programming languages
- Organizing your code into classes, their methods and objects improves structure of your code and allows for better control of your projects
- OOP allows for utilizing similarities across programming tasks and organizing them in a synthetic way
- R and Python are not object-oriented languages (like Java or C#). This means that problems in R and Python can be solved using functions and procedures, and the code can be executed without creating classes
- Both R and Python support OOP paradigm, as it is a very elegant and useful way to deal with specific programmistic issues

OBJECT ORIENTED PROGRAMMING (OOP)

Centered around the idea of “classes” which create structure for their “objects” or “class instances”



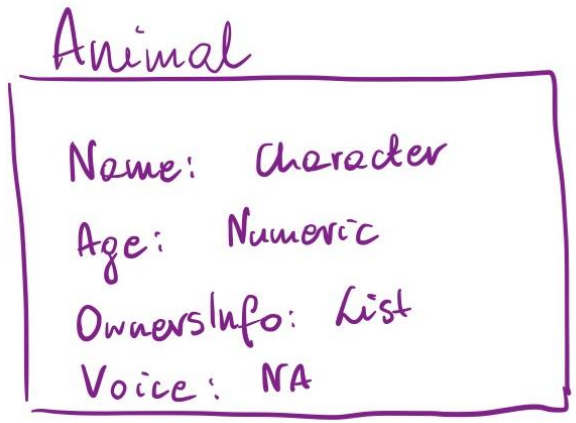
Class – student
Common structure for all students in the program/database

Two students: Sandra and John who follow the same structure as the class. Individual values differ, but the structure stays the same

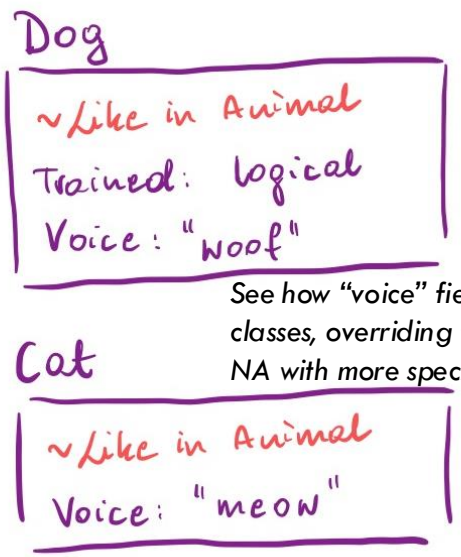
For both Sandra and John, the method ifPassed() will work, as it uses fields that are common for all students

INHERITANCE

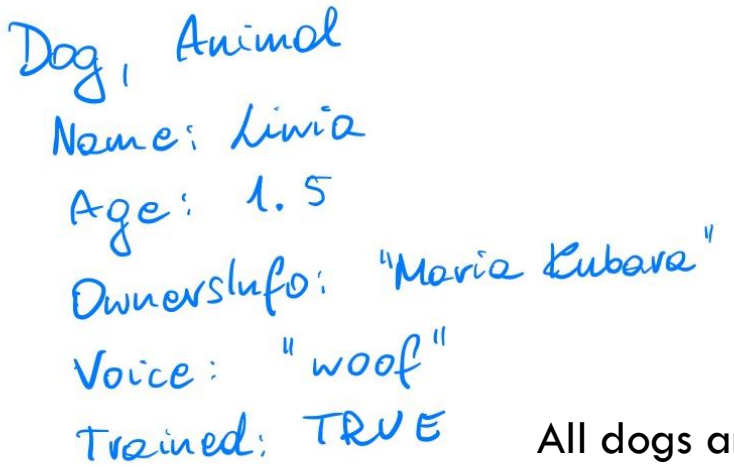
We can group common features into "parent" class (more general class)

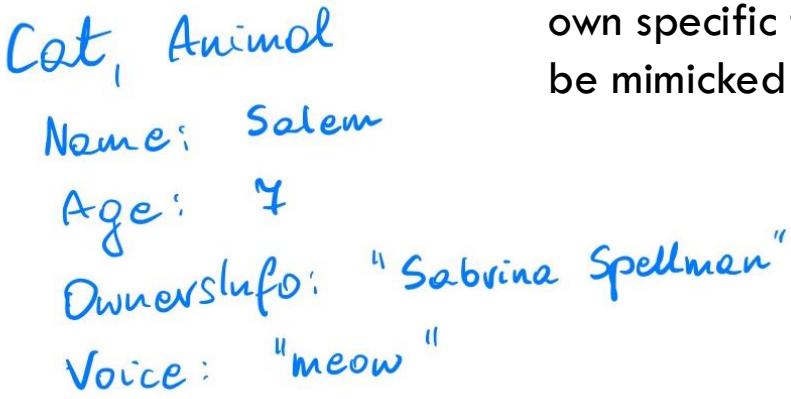


And create "child" classes which store more specific fields



See how "voice" field changes in child classes, overriding the general value NA with more specific content





All dogs and cats are animals – they have some common fields like name or age. However, they also have their own specific features, which should be mimicked in the class structure

INHERITANCE

Animal

Name: Character
Age: Numeric
OwnersInfo: List
Voice: NA

Dog

~Like in Animal
Trained: logical
Voice: "woof"

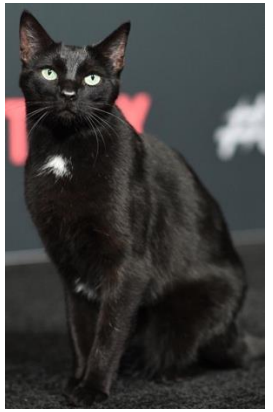
Cat

~Like in Animal
Voice: "meow"

Dog, Animal
Name: Livia
Age: 1.5
OwnersInfo: "Maria Kubara"
Voice: "woof"
Trained: TRUE

Cat, Animal
Name: Salem
Age: 7
OwnersInfo: "Sabrina Spellman"
Voice: "meow"

These two beautiful creatures need their own specific classes

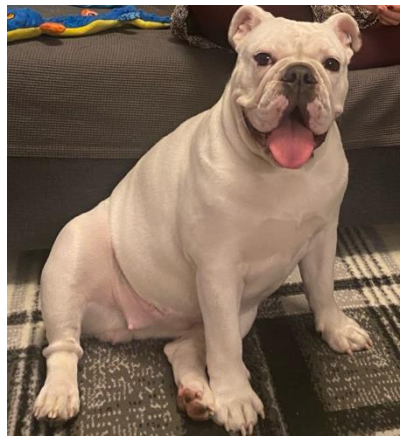


POLYMORPHISM

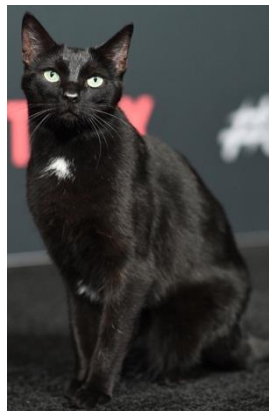
We can create a function “sayHello” which will work on Animal objects. See that the behavior will be different depending on the (child) class:

Class: Animal
sayHello: “I don’t have a voice”

Class: Dog (Animal)
sayHello: “I’m a Dog. Woof!”



Class: Cat (Animal)
sayHello: “I’m a Cat. Meow!”



Function sayHello() is polymorphic – its output has different shape depending on the class

WHAT DO CLASSES GIVE US?

- **Elegant code –** it gives structure and clarity about types of the elements within a project
- **Solution for "laziness" –** less typing is needed, as we can smartly group common fields into categories – it diminishes redundancy of repeated codes
- **Organization –** the only solution for growing projects, which allows to create structured objects with paired functions, which work correspondingly to the content of the class

KEY CONCEPTS

- **Class** – what an object is
- **Method** – what an object can do (function paired with a class)
- **Object** – instance of a class
- **Fields** – features within a class, characteristics of an object
- **Inheritance** – creation of more specific classes, which take from the general class and extend its fields further, if a field of a method is not defined within the child class, value from parent class will be used (child inherits the behavior and content)
- **Polymorphism** – possibility to create different function behaviors for specific classes

CONNECTION OF METHODS AND CLASSES

In R we have a set of initial objects which do not follow any OOP structures. They serve as building blocks for all the remaining extensions of R language – e.g. numeric, character, list
Their attribute "class" is empty.

```
> attr(1:10, "class")
NULL
> attr(iris, "class")
[1] "data.frame"
```

Encapsulated OOP

Methods belong to objects or classes

Method calls typically look like object.method(arg1, arg2)

Object encapsulates both data (with fields) and behavior (with methods)

Popular approach, used in most popular languages

R6 S4

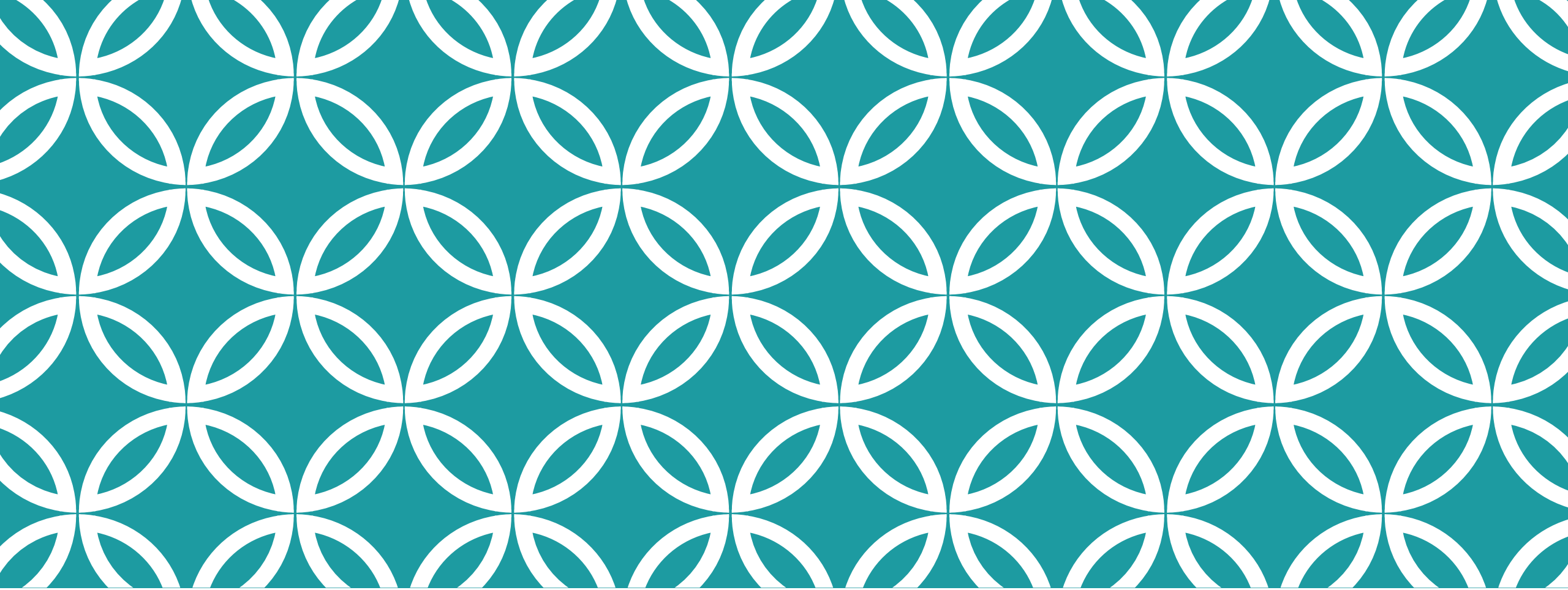
Functional OOP

Methods belong to generic functions

Method calls look like ordinary function calls: generic(object, arg2, arg3)

This is called functional because from the outside it looks like a regular function call – specific version of generic function is created to work with a particular class

S3



S3

Minimalistic OOP

S3

- S3 is the simplest system for OOP in R
- It does not impose many restrictions by itself
- It is very minimalistic – it only has the necessary elements for implementing OOP
- Because of its simplicity it is widely used in R – e.g. factors and data.frames are implemented in S3

CLASS DEFINITION IN S3

There are no formal class definitions in S3!

Classes in S3 are created "ad hoc" by adding a new "class" attribute to a list

As elements in lists can be named, we can create intriguing objects with defined structure under specific class definition

Important: It is our responsibility to ensure the proper class structure. We are imposing restrictions, not S3.

```
> myList <- list("I am a list")
> myList
[[1]]
[1] "I am a list"

> attr(myList, "class")
NULL
> class(myList) <- "newClass"
> attr(myList, "class")
[1] "newClass"
> myList
[[1]]
[1] "I am a list"

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

New class is created here

We have the first object of class "newClass"

You can do the same with `structure(myList, class = "newClass")`

HOW TO ENSURE STRUCTURE IN S3

Constructor function! Good practice is to create a function named exactly as your class which will ensure proper class structure when an object is created

```
student <- function(name, ID, GPA){
   stopifnot(is.character(name))
   stopifnot(is.integer(ID))
   stopifnot(is.numeric(GPA))

   studentObject <- list(name = name, ID = ID, GPA = GPA)
   studentObject <- structure(studentObject, class = "student")
                                              #adding class attribute
   return(studentObject)
}
```

stopifnot() is a simple defensive function, which stops the execution and throws an error when a condition is not met. It does not, however, return user-friendly pop-ups. For better error handling use if() statements paired with stop("Your own error message") inside

METHODS IN S3

Methods in S3 do not belong to class definition. They belong to generic functions.

First a generic function must exist, and then we can write its specific behavior which will work for our new class.

```
newGenericFunction <- function(x) {
  UseMethod("newGenericFunction")
}
newGenericFunction.default <- function(x) print("Basic behavior")
newGenericFunction.myClass <- function(x) print("Behavior for class myClass")
```

UseMethod is a heart of any generic function
It direct the behavior to the specific method defined for a given class. If no specific method is found the default method is used

INHERITANCE IN S3

- As may be expected, there are no clear rules regarding inheritance in S3
- The usual practice utilizes the inheritance of methods based on the content of the class vector
- It is automatically done in S3 by applying methods defined for the 2nd, 3rd … element of the class attribute vector is a method for the 1st element was not found – using a parent's method for the child object
- Using NextMethod() we can use the definition of the parent's method in the child's method definition to extend its possibilities. This is a nice practice to limit redundant co
- Different types of inheritance structures can be implemented in S3 by hand. It is the user's choice.