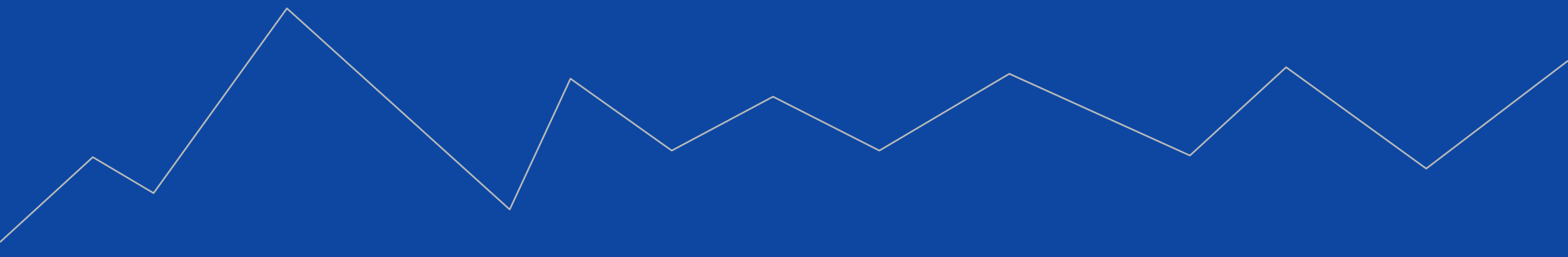
Object-Oriented Programming in R

Anthony Shook, PhD
Jellyvision

Things we'll plan to cover (Not necessarily in this order)

- Quick intro to Object-Oriented Programming
  - Like… what the heck it is
- The main benefits to OOP and how R represents them
- Implementations in R
  - S3 -- The popular kid
  - S4 -- The stuffy one
  - Relative Classes (RC) -- The one nobody talks about (and we won't either)
  - R6 -- The 'new' kid with the cool motorcycle jacket who does things *a little differently…*

So… What is OOP anyway?



This may seem silly but...

- OOP is a programming paradigm focused on **objects and their interactions**
- This is contrast to **Procedural Programming**, where you have data that you pass from function to function in order to achieve a result
- The goal in OOP isn't to provide ordered instructions but to define classes of objects with attributes and behaviors
  - And, how those attributes and behaviors change over time, or influence *other* objects!
  - Philosophically, this means classes and how they act are not separable

Let’s look at this more in depth...

OOP: Classes and Methods

- A **class** is a blueprint that defines attributes and behaviors of objects
  - ‘objects’ are instances of ‘classes’ -- they are unique, but defined by the class blueprint
  - Methods and attributes are shared by objects of the class -- they apply universally*

Class: **Cats**

Attributes:
- *breed*
- *color*
- *pattern*
- *fur_length*

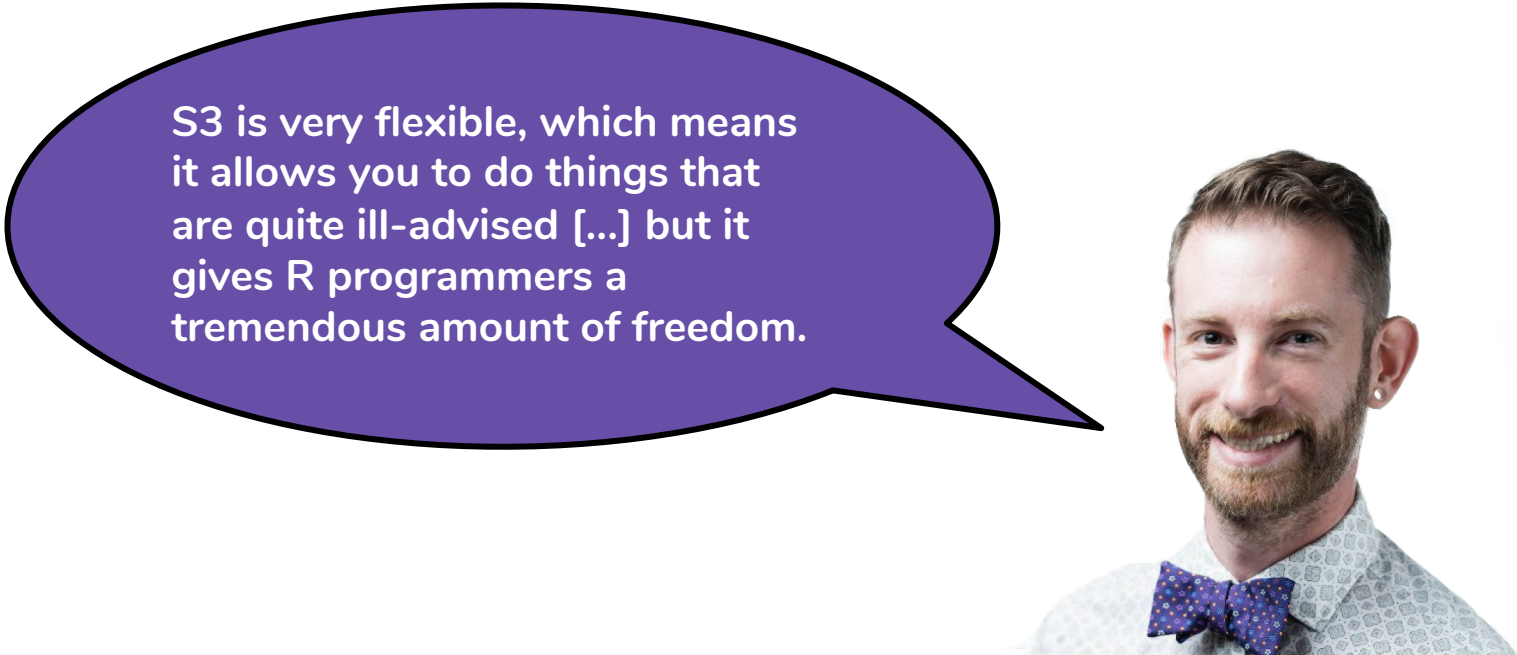
Methods:
- *pet()*
- *feed()*
- *fill_phone_with_photos_of()*

Objects

Let’s use the Construction of a Class as an introduction to R’s various OOP Systems

Introducing S3

- Here is R's most common, and least strict, OO paradigm: **S3**



Making a Class in S3

```
> ### Here I've made an object
> X <- list(name='Anthony', age = 36)
>
> ### Here I give it class
> class(X) <- 'student'
> X
$`name`
[1] "Anthony"

$age
[1] 36

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

- That's it. You just… tell R your object is a new class, and R is totally cool with it!
- If you're programming in **production environments**, this should probably terrify you.
    - You can arbitrarily give *any* object of *any* type *any* class.
    - Obviously, don't do this. It's a good way to completely break your understanding of what R is doing at any given moment.

Introducing S4

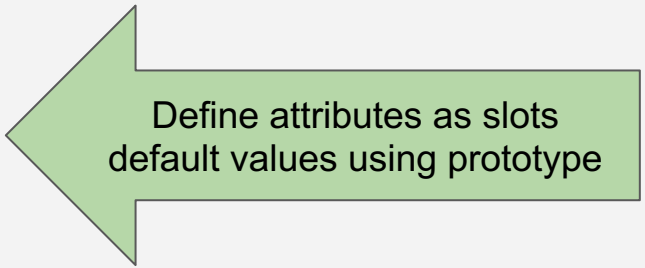
S4 is the more rigorous implementation of S3 -- it cares deeply about rules

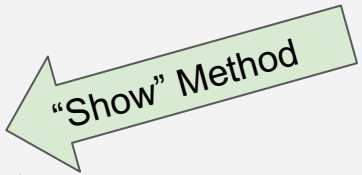
- In S3 you can create an object and just assign it a class
- S4 requires you to define your class and methods upfront!

- S4 uses a new element, called `slots`, to hold data
  - If you've ever tried to look at the internals to an object and had to use `@`, then you were almost certainly dealing with an S4 object
  - Check out packages like `Matrix`, `lme4`, or anything on Bioconductor

Making a Class in S4

```
# Student class
setClass(Class = 'student',
         slots = list(
           name = 'character',
           age  = 'numeric',
           area = 'character'
         ),
         prototype = list(
           name = 'Anthony',
           age = 36,
           area = 'Data Science'
         )
)

setMethod('show',
          signature = 'student',
          definition = function(object){
            cat(paste0(object@name, ", age ", object@age ,", is a student of ", object@area))
          })

> new('student')
Anthony, age 36, is a student of Data Science

> new('student', name = 'Rachel', age = 22, area = 'Chemistry')
Rachel, age 22, is a student of Chemistry
```





Introducing R6:
The least R-like OO Paradigm

- R6 is distinct from S3/S4 in several ways (which we'll get into more later):
  - It is a separate package that does not come loaded with R
    - There are benefits to this!
  - Methods belong to the class
    - They, in fact, exist in a separate NAMESPACE
  - R6 classes are **mutable**, meaning they have modify-in-place capabilities
    - Uncommon for R, which relies typically on modify-on-copy semantics
  - The code it produces does not look like typical R code

Classes and Methods in R6

- Class and Method definition typically happen in the same `R6Class` call


```
User <- R6Class("User",
                public = list(
                  name = NA,
                  creds = NA,
                  initialize = function(name, creds) {
                    self$creds <- creds
                    self$greet()
                  },
                  set_creds = function(val) {
                    self$creds <- val
                  },
                  greet = function() {
                    cat(paste0("Hello, my name is ", self$name,
                               ", and my user-level is: ", self$creds,".\n"))
                  }
                ))

User$new(name = 'Anthony')
Hello, my name is Anthony, and my user-level is: Basic.

User$new(name = 'Jim', creds = 'Admin')
Hello, my name is Jim, and my user-level is: Admin.
```

Handling Encapsulation: Keeping it Valid

- This brings us to our first important feature of OOP -- **encapsulation**

- Keeping some of the internal data / methods out of the hands of the user, often for security reasons
  - Often this means the user can't *directly* change the internal state of a class (e.g., alter the value of it's fields), but can only indirectly do so through methods
  - Sometimes those methods can only be called by the class itself!

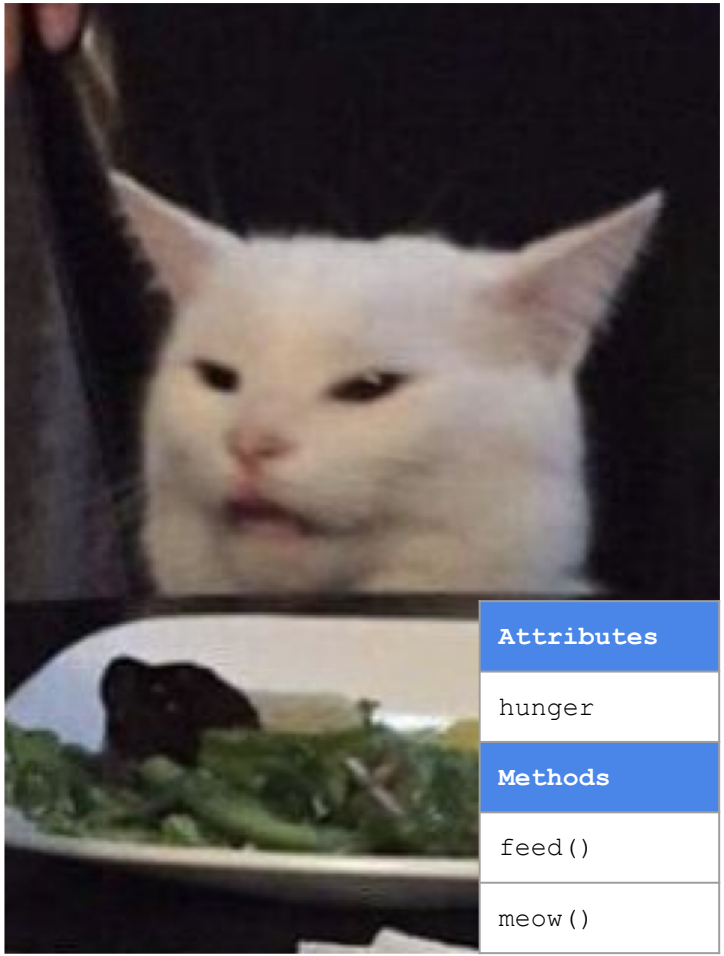
- It also sometimes refers simply to the practice of **encapsulating** data and behaviors (i.e., classes and methods) together into a single unit

The More You Know:
I'm using a sort of combined definition of **encapsulation** and **abstraction** (which we'll talk about shortly), but like everything else in programming, there isn't 100% agreement on what's quite right. #twospaces #tabsRforsuckerz

Cat Example #2

Here's a cat. His name is Smudge. He's hungry.

- He has one attribute, `hunger`, but you can't **directly** affect it
- All you can do is call the method `feed()`, which will have some effect on the attribute (though you don't know exactly how much)
  - `feed()` is a **public method**
- He also has a method that only he can use, `meow()`
  - This is **private method**
  - *You can ask him to meow, but he definitely won't do it just because you asked*



How does R handle it?

- Spoiler alert: depends on the paradigm!
- For example, in **S3** you really can't encapsulate anything very easily. In fact, you can do all sorts of strange things quite naturally.
- Fields aren't 'protected', either in value or existence.
  - The entirety of the class can generally be altered at a whim

```
# Here's my S3 class 'student'
print(X)
$`name`
[1] "Anthony"
$age
[1] 36
attr(,"class")
[1] "student"

# I can add a slot
X$GPA <- 4.0

# I can take one away
X$age <- NULL

# I can change the type of a slot
X$GPA <- as.character(X$GPA)

print(X)
$`name`
[1] "Anthony"
$GPA
[1] "4"
attr(,"class")
 [1] "student"
```

Encapsulation in S4 -- a little better

- You can’t just add or remove a slot from an existing class
  - You’d have to *extend it* to make alterations to the underlying class structure

- You can still access any slot publicly simply via `class@slot`
  - There is no way to make them private, save from trying to hide them in a slots like `@pleaseDoNotLookAtMe` or `@definitelyContainsHungryTigers`

- But S4 at least has formal validity-checking processes
  - S3 validity is checked by convention within a given method or constructor function

Explicit validation in S4

- Built in validation of type

```
> A <- new('student')
> A@name <- 10
Error in (function (cl, name, valueClass)  :
  assignment of an object of class “numeric” is not valid for @‘name’ in an object of class “student”;
is(value, "character") is not TRUE
```

- Additional Validity checks can be added with setValidity

```
setValidity('student', method = function(object){
  if (object@age >= 150 || object@age <= 0) {
    message('Supplied an `age` outside reasonable expectations')
    return(FALSE)
  } else {
    return(TRUE)
  }
})

> A <- new('student', name = 'Methuselah', age = 2000)
Supplied an `age` outside reasonable expectations
Error in validObject(.Object) : invalid class “student” object: FALSE
```

Accessor Functions in S4 (getter / setter)

- Using *Accessor functions* helps encapsulation by giving your users an alternate way to access or alter data
  - Show them how to use the door so they don't break the window
- It also allows you more control over object **validity**
  - Validity checks on S4 objects are only automatically run during creation. So you can make a valid object, then make it invalid

```
> A <- new('student', name = 'Methusela', age = 2000)
Supplied an `age` outside reasonable expectations
Error in validObject(.Object) : invalid class "student" object: FALSE

> A <- new('student', name = 'Methusela', age = 20)

> A@age <- 2000  # NO ERROR!
```

Accessor Functions

```
setGeneric('age', function(object) standardGeneric('age'))
setMethod('age', signature = 'student', definition = function(object) object@age)

# Setter
setGeneric('age<-', function(object, value) standardGeneric('age<-'))
setMethod('age<-', signature = 'student',
          definition = function(object, value) {
            object@age <- value
            validObject(object)
            return(object)
          })

> A <- new('student', name = 'Anthony', age = 22)
> age(A)
[1] 22
> age(A) <- 2000
Supplied an `age` outside reasonable expectations
Error in validObject(object) : invalid class "student" object: FALSE
```

R6 is the only fully encapsulated OO Paradigm in R*

Well, so is RC, but I already told you…* ***we don't talk about RC*

- Private Fields and Methods
  - Only accessible and alterable by *internal methods*
- Locked Classes
  - Slots cannot be added or removed (unless unlocked, of course)
- Methods belong to the namespace of the class object, NOT to the global namespace (unlike S3 and S4)

R6: Private Fields, Encapsulated Methods

```
Person <- R6Class(
 classname = "Person",
 private = list(
     name = NULL,
     age = NA
   ),
 public = list(
   initialize = function(name, age = NA) {
     stopifnot(is.character(private$age))
     private$name <- name
     private$age <- age
   }
 )
)
> A <- Person$new(name = "Anthony", age = 36)
> A$name
 NULL
```

```
Accumulator <- R6Class("Accumulator",
  list(
    sum = 0,
    add = function(x = 1) {
      self$sum <- self$sum + x
      invisible(self)
  })
)

x <- Accumulator$new()

# Calling the Method
x$add(10)

# Can't call method apart from class
x <- add(x, 10)
Error in add(x, 10) : could not find function "add"
```

Note that validity checking in R6 is not as formal as S4; type-safety is typically added manually by the developer through use of `stopifnot()` [see example on the left above]

Why do we care about encapsulation?

- Encapsulation, in and of itself, isn't so important when you don't care about hiding data, or enforcing some kind of validity
- Most people in aren't digging around in classes and methods they don't own
  - And if they are, they really only have themselves to blame when things go sideways
- Encapsulation naturally leads to another benefit: **Abstraction**

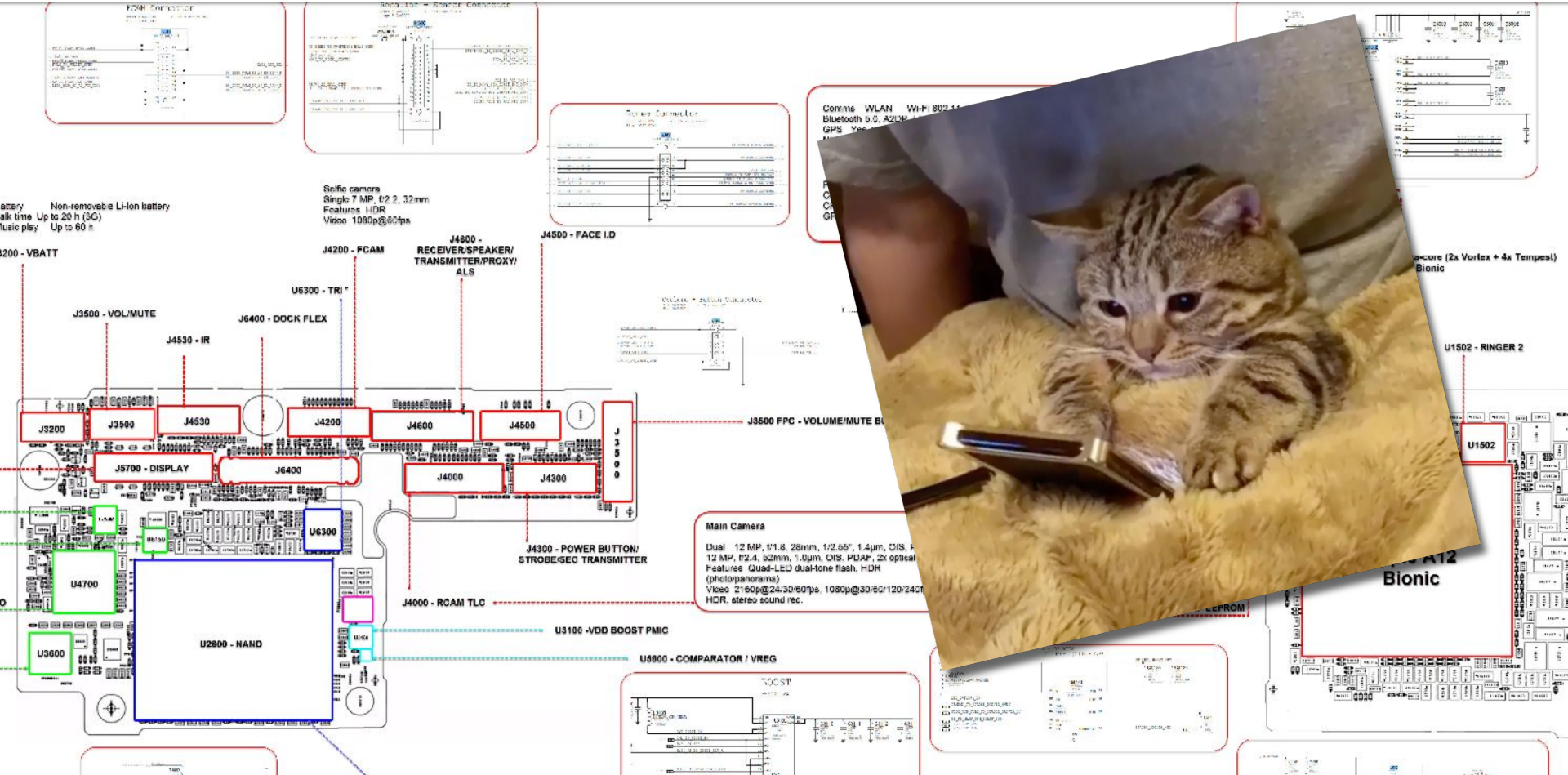
Abstraction

- Code-bases, even moderately sized, can be fairly complex
  - Lot’s of moving parts and components that interact

- Maintaining large code-bases can be difficult, especially when you’re working with teams focused on different sections of the pipeline

- **Abstraction** is a way of managing this complexity by hiding away the really complex stuff that the average user doesn’t need to know about, in favor of some *high level way of interacting with the code*

Abstraction Example: iPhone XS Circuit



Abstraction is very beneficial to both the user and the developer

- **Users** are given only what they need to make the code work
- They don't need to worry about *how* the thing is happening, just that it **is** happening

- **Developers** have flexibility to change internals as long as they don't break the high-level API
  - Switch from `dplyr` to `data.table` and users will interact with the classes in exactly the same way they used to
- Can make testing easier
  - Testing of high-level behaviors still apply even when you change the implementation

Good news, everyone!

All of the OOP Paradigms in R handle abstraction well!

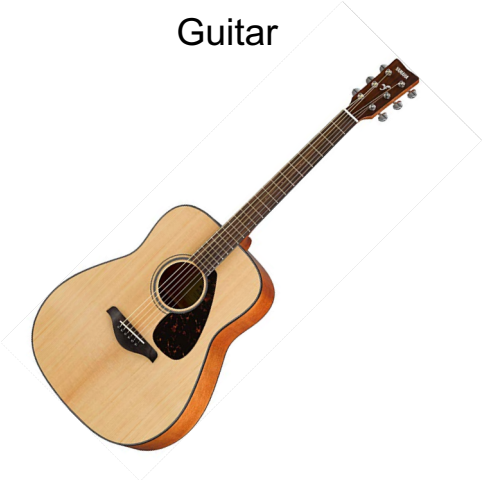
How? Classes & Methods!

- Classes and methods *create* abstraction
- Classes typically do so via **encapsulation**
- Methods typically rely on another pillar of OOP: **Polymorphism**

Polymorphism

- This is probably the single biggest reason to use OOP, esp. in R
- **Polymorphism** is a way of giving a single instruction that reflects *desired output* while abstracting the class-specific implementation

Guitar



Piano

Badgermin
(Badger + Theremin)

Method: `play()`

Polymorphism is great. Full stop.

- **Polymorphism** is what allows you to run the same function on lots of different object types in R, and still get a result

Example - `summary()`

- R knows *what* you want is a ‘reasonable’ summary of an arbitrary object, and that you *don’t really care *how** R got there.

```
> summary(iris$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900


> summary(iris$Species)
    setosa versicolor  virginica
        50         50         50


> summary(lm(Petal.Width ~ Sepal.Width, data = iris))

Call:
lm(formula = Petal.Width ~ Sepal.Width, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-1.38424 -0.60889 -0.03208  0.52691  1.64812

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   3.1569     0.4131   7.642 2.47e-12 ***
Sepal.Width  -0.6403     0.1338  -4.786 4.07e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.7117 on 148 degrees of freedom
Multiple R-squared:  0.134,         Adjusted R-squared:  0.1282
F-statistic: 22.91 on 1 and 148 DF,  p-value: 4.073e-06
```

Polymorphism in S3, S4, and R6

- R's OOP methods actually exhibit two forms of polymorphism
- S3 and S4 rely on *parametric polymorphism*: methods are implemented through the use of generic functions
- R6 methods, as we've noted, belong to the class itself (sometimes called *Internal Polymorphism*)
- Both rely on a process called **Method Dispatch**

Method Dispatch

Take, as an example, *`print()`*

What's the functional definition of print?

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x0000023dc30d3858>
<environment: namespace:base>
```

Method Dispatch

- When you call `print(...)`, you're not calling the actual instructions of *what to do with the input* -- you're running a **generic** that seeks out the proper method for that class

- How the **generic** knows what method to pull up and apply to the input is via Method Dispatch

Example from S3 with sloop::dispatch

```
library(sloop)

##' Make some objects of different types
x_numeric    <- 1
x_data_frame <- iris
x_dates      <- Sys.Date()

> ##' Examine the dispatch
> s3_dispatch(print(x_numeric))
   print.double
   print.numeric
=> print.default
> s3_dispatch(print(x_data_frame))
=> print.data.frame
 * print.default
> s3_dispatch(print(x_dates))
=> print.Date
 * print.default
>
```

s3_dispatch shows what method was called by the **generic** function 'print'

- X_data_frame: `` `print.data.frame` ``
- x_dates: `` `print.Date` ``

Writing Methods in S3

- So, S3's dispatch is easy to understand – it's looking for functions with a specific syntactic shape

```
generic.yourClass <- function(...)
```

- One wonderful thing about S3 is that once you have a new class, implementing methods with existing generics is very simple

Example: print method for class ‘student’

```
# a constructor function for a new class "student"
student <- function(name, major, graduation_year, gpa) {
  x <- list(name = name, major = major, graduation_year = graduation_year, gpa = gpa)
  attr(x, 'class') <- 'student'
  x
}

> MEIRL <- student(name = 'Anthony Shook', major = 'Cat Studies', graduation_year =
2014, gpa = 3.7)
> print(MEIRL)
$`name`
[1] "Anthony Shook"
$major
[1] "Cat Studies"
$graduation_year
[1] 2014
$gpa
[1] 3.7
attr(,"class")
[1] "student"
```

Example: print method for class ‘student’

```
# Here’s the original dispatch, which indicates that no method for class student
existed, so it used the default print method.
> s3_dispatch(print(MEIRL))
   print.student
=> print.default


# Now we write a new Method and rely upon S3’s generic.class() convention to find it
print.student <- function(obj) {
  paste0(obj$name, ' (', obj$graduation_year, ') majored in ',
         obj$major, '. They graduated with a ', obj$gpa, ' GPA.')
}

> print(MEIRL)
[1] "Anthony Shook (2014) majored in Cat Studies. They graduated with a 3.7 GPA."

# Check the new dispatch!
> s3_dispatch(print(MEIRL))
=> print.student
 * print.default
```

So, again: **Polymorphism** is pretty great

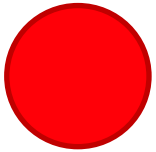
- Ability to create methods that match already existing **generics** for S3/S4 lowers the barrier of entry substantially
  - Users can use your new class and the things they know and love, like `print` and `summary`, will just work

- In R6, any class can have a `class$plot()` or `class$print()` method written directly into the class
  - Since it lives *within* the class, it's knows exactly what to do with objects of that class

- You can also leverage it to just make your code cleaner and clearer

Code Clarity via custom generics

- Let’s start with some example classes



```r
# Triangle
triangle <- function(base, height) {
  tr <- list(base = base, height = height)
  attr(tr, 'class') <- 'triangle'
  return(tr)
}

# Rectangle
rectangle <- function(base, height) {
  rt <- list(base = base, height = height)
  attr(rt, 'class') <- 'rectangle'
  return(rt)
}

# Circle
circle <- function(diameter) {
  cr <- list(diameter = diameter)
  attr(cr, 'class') <- 'circle'
  return(cr)
}
```

Conditional Function vs. Generic Function

```r
## Conditional function
get_area <- function(obj) {
  if (is(obj, 'triangle')) {
    area <- (.5 * obj$base) * obj$height
  } else if (is(obj, 'rectangle')) {
    area <- (obj$base * obj$height)
  } else if (is(obj, 'circle')) {
    area <- (pi/4) * (obj$diameter^2)
  } else {
    stop("No function recognized for class ", class(obj))
  }

  return(area)
}
```

```r
## Generic Function
get_area <- function(obj) {
  UseMethod("get_area")
}

get_area.triangle <- function(obj) {
   (.5 * obj$base) * obj$height
}
get_area.rectangle <- function(obj) {
  (obj$base * obj$height)
}
get_area.circle <- function(obj) {
  (pi/4) * (obj$diameter^2)
}
```

Conditional

- Just a big if-else statement
- Increases in size/complexity every time you add a new class or subclass
- Developers can't extend it (because they don't "own" the function internals)

Generic

- Relies on Method Dispatch
- Better complexity management by separating functional behavior
- Allows extension
  - I can add class `rhombus`, and method `get_area.rhombus` without needing access to source code

And more good news!

Those generic S3 functions work for S4 as well!

```
# S4 Class ‘myClass’ containing the iris data set
setClass('myClass', slots = list(data = "data.frame"))
A <- new('myClass', data = iris)

# Define a Generic around that class
summary.myClass <- function(object) {summary(object@data)}

> summary(A)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Defining New Methods in S4: The formal way

```
# Make an 'employee' class
setClass("employee", slots = list(name = 'character', salary = 'numeric'))

# Register a GENERIC function (this is our Parametric Polymorphism)
setGeneric('give_raise', function(object, percentage) standardGeneric('give_raise'))

# Set the method for this class, with that generic
setMethod('give_raise',
          signature = 'employee',
          definition = function(object, percentage){
            object@salary <- object@salary * (1 + percentage)
            return(object)
          } )
```

Using the S4 Method

```
# Define a new object of the employee class
> emp <- new('employee', name = 'Anthony', salary = 50000)
> emp
An object of class "employee"
Slot "name":
[1] "Anthony"

Slot "salary":
[1] 50000

# Use the Method
> give_raise(emp, percentage = .10)
An object of class "employee"
Slot "name":
[1] "Anthony"

Slot "salary":
[1] 55000
```

Defining Methods in R6

- For the 10 millionth time, these are typically defined within the class

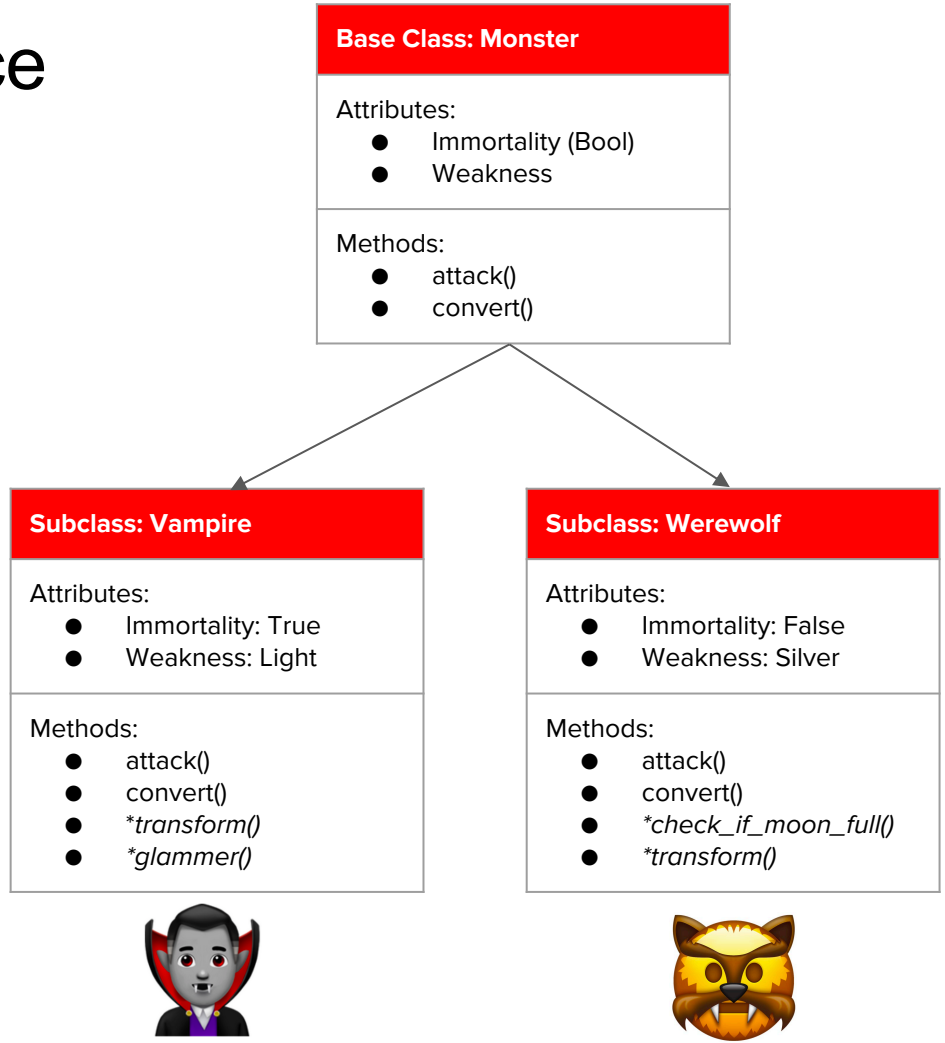
- They can be added later with `$set` as long as the class isn't locked

```r
Accumulator <- R6Class("Accumulator",
  list(
    sum = 0,
    add = function(x = 1) {
      self$sum <- self$sum + x
      invisible(self)
  })
)

x <- Accumulator$new()
x$add(10)

# Unlock
Accumulator$unlock()

# Set new method
Accumulator$set(which = 'public',
                name = 'subtract',
                value =  function(x) {
                  self$sum <- self$sum - x
                  invisible(self)
                })
```

The final pillar: Inheritance

- So you've defined your class, you've written some (abstracted) methods and you realize -- you want to share the wealth

- **Inheritance** is the process by which subclasses can obtain characteristics of (and *extend*) a base class
    - All its attributes *and* methods



Inheritance

- When you have many interrelated classes, there’s a chance they will share attributes, and sometimes even specific implementations of methods
- Inheritance lets you re-use code for related classes
- It also represents hierarchical relationships directly in the code

Inheritance in R

- Inheritance in S3 is superficial, and in R6 is quite straightforward
  - The function `R6Class()` has an optional `inherit` argument, which can accept another `R6ClassGenerator` object
- Let’s look at S4’s inheritance just as an example
  - We won’t show it here, but S4 also allows *Multiple Inheritance* -- so a child-class can have multiple parents

S4: Straightforward Inheritance

```
# Here's a gameCharacter class (our BASE class)
setClass(Class = 'gameCharacter',
         slots = list(name = 'character', hit_points = 'numeric'),
         contains = 'VIRTUAL'  # This means the class can't be called by itself, but can be extended
        )

# Set show method for the Base class
setMethod('show',
          signature = 'gameCharacter',
          function(object) print(paste0('Character: ', object@name, '; Hit points: ', object@hit_points)))


### Extend class into a Wizard!  (´°‿°)つ━☆・*。
setClass(Class = 'wizard',
         slots = list(magic_points = 'numeric'),
         contains = 'gameCharacter') # This is how you inherit from another class

# the Wizard class inherits the 'show' method from the Base class
harry <- new('wizard', name = 'Harry', hit_points = 1000, magic_points = 200)

> harry
[1] "Character: Harry; Hit points: 1000"
> slotNames(harry)
[1] "magic_points" "name"         "hit_points"
```

ENCAPSULATION

ABSTRACTION

INHERITANCE

POLYMORPHISM

A few notes about R6

Because R6 is different and those differences are worth talking about

R6 has some quirks

- Some of what makes R6 interesting also make it very different from how most of R behaves
- One of those things is **mutability**
- S3 and S4 are **immutable** -- to change their values, they require re-assignment, and that the method be passed the object itself

```
s3_or_s4_Class <- method(s4Class, arguments)
```

Mutability

In R6, however, objects are edited in place -- no need to assign with **<-**

```
> x <- Accumulator$new()
> x$sum
[1] 0
> x$add(10)
> x$sum
[1] 10
> x$add(87)
> x$sum
[1] 97
```

Method Chaining

- In our Accumulator example, the method `add` is used for its **side-effect** -- updating the field `sum` in the class object
- Because the method invisibly returns the object itself, you can chain method calls (similar to piping in the tidyverse with `%>%`)

```
> x <- Accumulator$new()
> x$sum
[1] 0
> x$add(10)$add(7)$add(4)$add(3)
> x$sum
[1] 24
```

Trickiest Part: Reference Semantics

- If you're new to OOP, and R6 specifically, this is something that you're going to have to learn to live with
  - Sidenote: If you're a regular user of `data.table`, this might be a concern you're familiar with!

- R6 stores a reference to an object in memory -- it does NOT silently copy objects the way most R functions do.
  - So you can get what seems like weird behavior, like this:

```
y1 <- Accumulator$new()
y2 <- y1

# Add to Y1 and Y2 is updated as well!
y1$add(10)
c(y1$sum, y2$sum)
[1] 10 10

# Add to Y2 and Y1 gets updated as well!
y2$add(5)
c(y1$sum, y2$sum)
[1] 15 15
```

Reference Semantics

- If you want to make a copy of something, you need to use object$clone()
  - A lot like using `data.table::copy()`

```
y1 <- Accumulator$new()
y2 <- y1$clone()

# Add to Y1 and check values
y1$add(10)
c(y1$sum, y2$sum)
[1] 10  0

# Add to Y2 and check values
y2$add(5)
c(y1$sum, y2$sum)
[1] 10  5
```

Okay, we're done with
the ginormous blocks of code:
Take a breath!

Final thoughts -- Why use OOP in R?

- **Polymorphism** and **Inheritance** make code less complex
  - Coherent, reusable code blocks
- **Encapsulation** and **Abstraction** helps you hide code and data, helping to separate the user-needs from the literal interpretation
- Easy adoption of known R methods**,** like `print`, `summary`, etc.
  - This makes it much easier for someone to pick up your code and understand how to use it
- IMO, OOP helps you reason about your data / process better
  - Forces you to think about *what* you want to do before thinking about *how* to do it
  - You have to think abstractly about what different forms an object instance might take

When to use OOP

- If you’re running a single script to perform an analysis, or writing just a few functions to work with existing classes (e.g., data.frames), OOP is going to be overkill
- If you’re developing a package in R, OOP is going to be extremely useful!

Which paradigm to use?

- Here’s my unsatisfying answer -- whatever you like!
- But I do have some general thoughts...

Thoughts on S3

Use it when:

- You’re programming for yourself, not with other developers
- You don’t have a lot of classes that interact, or complex parent-child hierarchies to capture

Thoughts on S4

Use it when:

- You’re going to be using the code in a production environment, or with other developers
  - IMO, S4 is easier to reason about, because classes and methods have formal definitions
- You have complex parent-child hierarchies, or interrelated classes

Opinion Alert

- I prefer S4 to S3, precisely because it is more rigorous
  - Built-in type-strictness for individual slots
  - Formal design principles mean less community-driven convention to learn (or unlearn)
  - Formal functions for making classes and methods
    - setClass, setGeneric, and setMethod
- I *personally* find that S4 requires me to think about what I'm doing before I do it in a way that S3 does not
  - S4's rigidity means changing class relationships or structures mid-stream is more difficult
  - Choosing S3 or S4 is a balancing act between flexibility and front-loading your design

Thoughts on R6

Use it when:

- You’re have to solve “threading state” problems (the `pop()` problem)
- You’re not terrified by difficult to debug reference semantics errors
- Developing behind-the-scenes
  - The code is non-idiomatic, but performant, so it’s great for managing internal processes

Thank you!