Programming with S4 Classes

Gaston Sanchez

October 9, 2016

S4 Classes

Another type of OOP system in R is the so-called S4 classes. This system is more formal and rigorous than S3 classes.

To define a new class, you use the `setClass()` function. For example, here's how to define a class `"coin"`:

```
# class "coin"
setClass(
  Class = "coin",
  representation = representation(
    sides = "character",
    prob = "numeric"
  )
)
```

The argument `Class` is used to specify the name of the class. The argument `representation` allows you specify the attributes of the objects. Compared to S3 classes, S4 classes allows you to be more explicit about the exact type of objects for the attributes. In the `coin` example, the `sides` of the coin are set to a character vector; likewise the `prob` (probabilities) of each side are set to a numeric vector.

You initialize a `"coin"` object with `new()`

```
coin1 <- new(Class = "coin",
             sides = c("heads", "tails"),
             prob = c(0.5, 0.5))
coin1
```

```
## An object of class "coin"
## Slot "sides":
## [1] "heads" "tails"
##
## Slot "prob":
## [1] 0.5 0.5
```

If you try to create a new `coin` with the wrong type of `sides` and `prob`, you will get an error message like this:

```
coin2 <- new(Class = "coin",
             sides = c(0, 1),
             prob = c(TRUE, FALSE))
```

```
## Error in validObject(.Object): invalid class "coin" object: 1: invalid object for slot "sides" in cl
## invalid class "coin" object: 2: invalid object for slot "prob" in class "coin": got class "logical",
```

Let's create another coin:

```
quarter1 <- new(Class = "coin",
                sides = c("washington", "fort"),
                prob = c(0.5, 0.5))
quarter1
```

```
## An object of class "coin"
## Slot "sides":
## [1] "washington" "fort"
##
## Slot "prob":
## [1] 0.5 0.5
```

You access the attributes with the slot operator `@`:

```
coin1@sides
```

```
## [1] "heads" "tails"
```

```
coin1@prob
```

```
## [1] 0.5 0.5
```

Prototype

When defining a class, often it's useful to include a `prototype`, that is, a *default* instance for an object:

```
# class "coin"
setClass(
  Class = "coin",
  representation = representation(
    sides = "character",
    prob = "numeric"
  ),
  prototype = prototype(
    sides = c('heads', 'tails'),
    prob = c(0.5, 0.5)
  )
)
```

Notice that, by default, creating a new `"coin"` will have `sides` attributes `"heads"` and `"tails"`, and probabilities `prob` 0.5 and 0.5 (i.e. a fair coin).

Let's re-initialize `coin1` with the default `prototype`:

```
coin1 <- new(Class = "coin")
coin1
```

```
## An object of class "coin"
## Slot "sides":
## [1] "heads" "tails"
##
## Slot "prob":
## [1] 0.5 0.5
```

To inspect the attributes of an object of class S4, you can use `slotNames()` and `getSlots()`

```
slotNames("coin")
```

```
## [1] "sides" "prob"
```

```
getSlots("coin")
```

```
##       sides        prob
## "character"   "numeric"
```

Print method

Like the `print` method with S3 classes, you can define a `print` method for S4 classes. To do so, use the function `setMethod()`. When declaring a specific `"print"` method you use the argument `signature = "coin"` to indicate that there will be a new `print()` method for objects `"coin"`.

```
setMethod(
  "print",
  signature = "coin",
  function(x, ...) {
    cat('object "coin"\n\n')
    cat("sides: ", x@sides, "\n")
    cat("prob : ", x@prob)
  }
)
```

```
## Creating a generic function for 'print' from package 'base' in the global environment
```

```
## [1] "print"
```

Now, when you `print()` an object of class `"coin"`, the specified method is applied to `"coin"`:

```
print(coin1)
```

```
## object "coin"
##
## sides:  heads tails
## prob :  0.5 0.5
```

Note that the `print` method only works when you explicitly call `print()`. If you just simplyt type the name of the object, the displayed values are different:

```
coin1
```

```
## An object of class "coin"
## Slot "sides":
## [1] "heads" "tails"
##
## Slot "prob":
## [1] 0.5 0.5
```

Show method

With S4 class objects, in addition to `print` methods, it is also common to define a `show` method:

```
setMethod("show",
          signature(object = "coin"),
          function(object) {
            cat("sides:", "\n")
            print(object@sides)
            cat("\nprob:", "\n")
            print(object@prob)
          })
```

```
## [1] "show"
```

The `show` method is the actual function that is called everytime you type the name of the object:

```
coin1
```

```
## sides:
## [1] "heads" "tails"
##
## prob:
## [1] 0.5 0.5
```

To see the defined methods on a given class, use `showMethods()`:

```
showMethods(class = "coin")
```

```
##
## Function ".DollarNames":
##  <not an S4 generic function>
##
## Function "complete":
##  <not an S4 generic function>
##
## Function "formals<-":
##  <not an S4 generic function>
##
## Function "functions":
##  <not an S4 generic function>
## Function: initialize (package methods)
## .Object="coin"
##     (inherited from: .Object="ANY")
##
## Function: print (package base)
## x="coin"
##
##
## Function "prompt":
##  <not an S4 generic function>
## Function: show (package methods)
## object="coin"
```

Validiting Attributes

The way we have set-up the class `"coin"` is still loosely defined. You could create a coin with more than two `sides` and `prob` with incorrect probabilities:

```
# weird coin
weird <- new("coin",
             sides = c('tic', 'tac', 'toe'),
             prob = c(1))
```

Even though we are requiring `sides` to be `character`, and `prob` to be `numeric`, we didn't specified anything else about the length, or their possible content.

To have a better ensuring mechanism, S4 provides a `validity` argument:

```
# class "coin"
setClass(
  Class = "coin",
  representation = representation(
    sides = "character",
    prob = "numeric"
  ),
  validity = function(object) {
    if (length(object@sides) != 2) {
      stop("'sides' must be of length 2")
    }
    if (length(object@prob) != 2) {
      stop("'prob' must be of length 2")
    }
  },
  prototype = prototype(
    sides = c('heads', 'tails'),
    prob = c(0.5, 0.5)
  )
)
```

Now, it is less likely to have weird coins:

```
weird <- new("coin",
             sides = c('tic', 'tac', 'toe'),
             prob = c(1))
```

```
## Error in validityMethod(object): 'sides' must be of length 2
```

To be have a more complete validity function, you can create an external auxiliary function, e.g. `validate_prob()`, that checks both `sides` and `prob` of a potential `"coin"` object:

```
validate_prob <- function(object) {
  if (length(object@sides) != 2) {
    stop("'sides' must be of length 2")
  }
  if (length(object@prob) != 2 | !is.numeric(object@prob)) {
    stop("\n'prob' must be a numeric vector of length 2")
```

```
  }
  if (any(object@prob < 0) | any(object@prob > 1)) {
    stop("\n'prob' values must be between 0 and 1")
  }
  if (sum(object@prob) != 1) {
    stop("\nelements in 'prob' must add up to 1")
  }
  TRUE
}
```

And then, include `validate_prob()` as the value of the `validity` argument, inside the `setClass()`:

```
# class "coin"
setClass(
  Class = "coin",
  representation = representation(
    sides = "character",
    prob = "numeric"
  ),
  validity = validate_prob,
  prototype = prototype(
    sides = c('heads', 'tails'),
    prob = c(0.5, 0.5)
  )
)
```

Public Constructor Function

Initializing an object with `new()` is not very user friendly. Instead, you typically create a user-intended **public constructor** function:

```
coin <- function(sides, prob) {
  new(Class = "coin",
      sides = sides,
      prob = prob)
}
```

Using the public constructor function is like

```
loaded <- coin(sides = c('h', 't'), prob = c(0.3, 0.7))

loaded
```

```
## sides:
## [1] "h" "t"
##
## prob:
## [1] 0.3 0.7
```

New Generic Methods

In addition to existing methods in R, you can also declare a new generic method. Use `setGeneric()`:

```r
setGeneric(
  "flip",
  function(object, ...) standardGeneric("flip")
)
```

```
## [1] "flip"
```

Once the method has been declared, you use `setMethod()` for defining specific methods:

```r
setMethod(
  "flip",
  signature = "coin",
  function(object, times = 1) {
    if (!is.numeric(times) | times <= 0) {
      stop("\n'times' must be a positive integer")
    }
    sample(object@sides, size = times, replace = TRUE, prob = object@prob)
  }
)
```

```
## [1] "flip"
```

Let's try `flip()`

```r
flip(coin1, times = 5)
```

```
## [1] "tails" "heads" "heads" "tails" "tails"
```

A `"toss"` object

Like we did with S3 classes, we are going to create a `"toss"` object using S4 classes. This object will have the following attributes:

- the vector of tosses
- the `sides` of the coin
- the `prob` of each side
- the `total` number of tosses
- the number of `heads`
- the number of `tails`

```r
# class "toss"
setClass(
  Class = "toss",
  representation = representation(
    tosses = "character",
    sides = "character",
    prob = "numeric",
    total = "integer",
    heads = "integer",
    tails = "integer"
  )
)
```

Instead of using `new()` we are going to create a public constructor function `toss()`:

```
toss <- function(coin, times) {
  tosses <- flip(coin, times = times)
  new(Class = "toss",
      tosses = tosses,
      sides = coin@sides,
      prob = coin@prob,
      total = length(tosses),
      heads = sum(tosses == coin@sides[1]),
      tails = sum(tosses == coin@sides[2]))
}
```

Tossing a coin 10 times:

```
toss(coin1, 10)
```

```
## An object of class "toss"
## Slot "tosses":
##  [1] "heads" "tails" "tails" "tails" "tails" "tails" "tails" "heads"
##  [9] "heads" "tails"
##
## Slot "sides":
## [1] "heads" "tails"
##
## Slot "prob":
## [1] 0.5 0.5
##
## Slot "total":
## [1] 10
##
## Slot "heads":
## [1] 3
##
## Slot "tails":
## [1] 7
```

Plot Method

Auxiliary functions:

```
head_freqs <- function(x) {
  cumsum(x$tosses == x$coin[1]) / 1:x$total
}


tail_freqs <- function(x) {
  cumsum(x$tosses == x$coin[2]) / 1:x$total
}


frequencies <- function(x, side = 1) {
```

```
  if (side == 1) {
    return(head_freqs(x))
  } else {
    return(tail_freqs(x))
  }
}
```

Finally, let's implement the `plot` method for objects `"toss"`

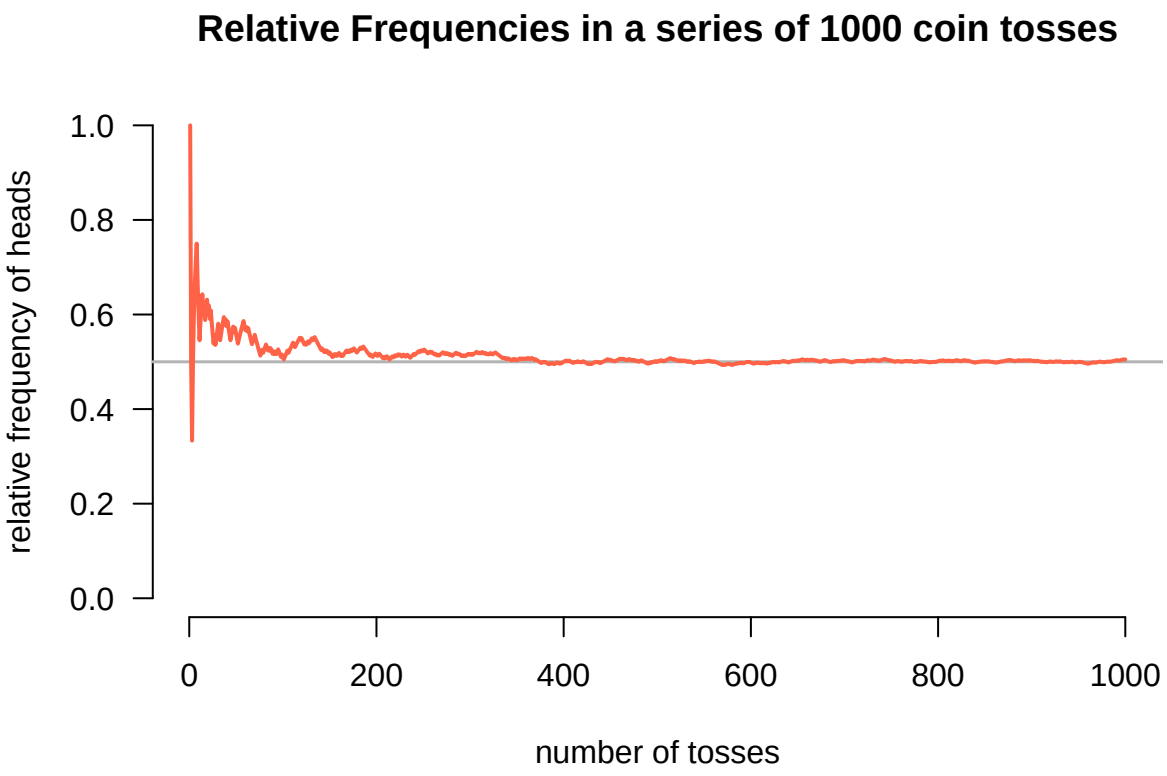
```
setMethod(
  "plot",
  signature = "toss",
  function(x, ...) {
    freqs <- cumsum(x@tosses == x@sides[1]) / 1:x@total
    plot(1:x@total, freqs, type = "n", ylim = c(0, 1), las = 1,
       xlab = "number of tosses", bty = "n",
       ylab = sprintf("relative frequency of %s", x@sides[1]), ...)
    abline(h = 0.5, col = "gray70", lwd = 1.5)
    lines(1:x@total, freqs, col = "tomato", lwd = 2)
    title(sprintf("Relative Frequencies in a series of %s coin tosses",
                  x@total))
  }
)
```

```
## Creating a generic function for 'plot' from package 'graphics' in the global environment
```

```
## [1] "plot"
```

Let's test our `plot` method:

```
set.seed(78943)
toss1 <- toss(coin1, 1000)
plot(toss1)
```


Relative Frequencies in a series of 1000 coin tosses
relative frequency of heads
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
number of tosses