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 fot 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
```

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 good 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 (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"
```

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')
    cat("sides: ")
    print(x@sides)
    cat("prob: ")
    print(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: [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"
##     (inherited from: object="ANY")
```

Constructor functions

The way we have defined the class `"coin"` is not entirely correct.

```
# 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 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
```

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))
```

Generic Methods

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

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

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

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

```
setMethod(
  "flip",
  signature = "coin",
  function(object, times = 1) {
    sample(object@sides, size = times, replace = TRUE, prob = object@prob)
  }
)
```

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

Let's try `flip()`

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

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