Object Oriented Programming

Gaston Sanchez

October 10, 2016

Programming in R: Coin Tossing

To illustrate the concepts behind object-oriented programming in R, we are going to implement code that simulates tossing a fair coin one or more times.

To toss a coin using R, we first need an object that plays the role of a coin. So let's start by creating a `coin` object using a character vector with two elements: `"heads"` and `"tails"`:

```
# coin object
coin <- c("heads", "tails")
```

Tossing a coin is a random experiment: you either get heads or tails. To get a random output in R we can use the function `sample()` which takes a random sample of a given vector. Here's how to simulate a coin toss using `sample()` to take a random sample of size 1 from `coin`:

```
# one toss
sample(coin, size = 1)
```

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

We can also use `sample()` to take samples of sizes different than one, and also to sample with replacement. To simulate multiple tosses, we can change the value of the `size` argument, and set `replace = TRUE`:

```
# 3 tosses
sample(coin, size = 3, replace = TRUE)
```

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

```
# 6 tosses
sample(coin, size = 6, replace = TRUE)
```

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

To make our code reusable, it's better to create a function that lets us toss a coin multiple times:

```
toss <- function(coin, times = 1) {
  sample(coin, size = times, replace = TRUE)
}

toss(coin, times = 1)
```

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

```
toss(coin, times = 4)
```

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

Typical probability problems that have to do with coin tossing, require to compute the total proportion of `"heads"` and `"tails"`:

```
# five tosses
five <- toss(coin, times = 5)

# proportion of heads and tails
sum(five == "heads") / 5
```

```
## [1] 0.6
```

```
sum(five == "tails") / 5
```

```
## [1] 0.4
```

It is also customary to compute the relative frequencies of `"heads"` and `"tails"` in a series of tosses:

```
# relative frequencies of heads
cumsum(five == "heads") / 1:length(five)
```

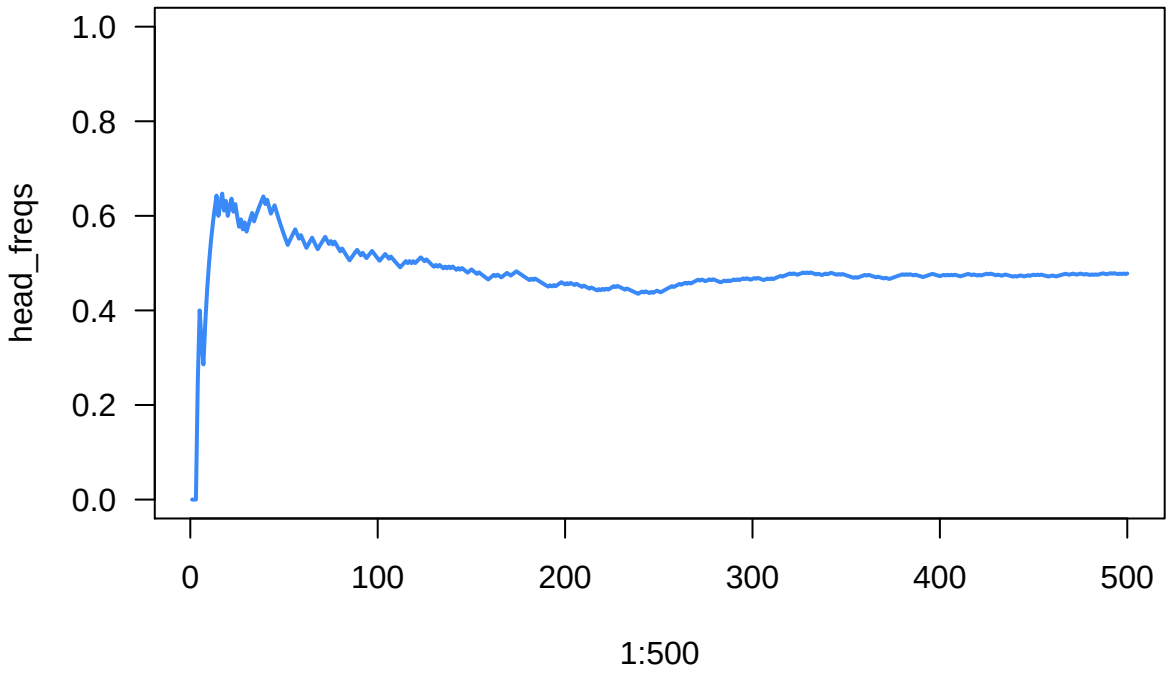
```
## [1] 1.0000000 1.0000000 0.6666667 0.7500000 0.6000000
```

```
# relative frequencies of tails
cumsum(five == "tails") / 1:length(five)
```

```
## [1] 0.0000000 0.0000000 0.3333333 0.2500000 0.4000000
```

Likewise, it is common to look at how the relative frequencies of heads or tails change over a series of tosses:

```
set.seed(5938)
hundreds <- toss(coin, times = 500)
head_freqs = cumsum(hundreds == "heads") / 1:500

plot(1:500, head_freqs, type = "l", ylim = c(0, 1), las = 1,
     col = "#3989f8", lwd = 2)
```



So far we have written code in R that simulates tossing a coin one or more times. We have included commands to compute proportion of heads and tails, as well the relative frequencies of heads (or tails) in a series of tosses. In addition, we have produced a plot of the relative frequencies and see how, as the number of tosses increases, the frequency of heads (and tails) approach 0.5.

In the following sections, we are going to see how to implement various functions and methods in R to make our coin tossing code more reliable, more structured, and more useful.

Object-Oriented Programming

Popular languages that use OOP include C++, Java, and Python. Different languages implement OOP in different ways. R also provides OOP capabilities, but compared to other languages, R's OOP options are less formal.

The idea of OOP is that all operations are built around objects, which have a **class**, and **methods** that operate on objects in the class. Classes are constructed to build on (inherit from) each other, so that one class may be a specialized form of another class, extending the components and methods of the simpler class (e.g. `"lm"`, and `"glm"` objects).

Note that in more formal OOP languages, all functions are associated with a class, while in R, only some are.

Often when you get to the point of developing OOP code in R, you're doing some serious programming, and you're going to be acting as a software engineer. It's a good idea to think carefully in advance about the design of the classes and methods.

Object-Oriented Programming in R

R has two (plus one) object oriented systems, so it can be a bit intimidating when you read and learn about them for the first time. The goal of this unit is not to make you an expert in all R's

OOP systems, but to help you become familiar with the so-called S3 and S4 classes.

R's three OO systems differ in how classes and methods are defined:

- **S3** implements a style of OO programming called generic-function OO. S3 uses a special type of function called a *generic* function that decides which method to call, e.g., `table(iris$Species)`. S3 is a very casual system. It has no formal definition of classes.
- **S4** works similarly to S3, but is more formal. There are two major differences to S3. S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods. S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.

Classic Programming

If you have no previous experience with object-oriented programming, it can be a bit challenging. You may be tempted to think that OOP does not provide any evident advantages: you need to think in advance before writing code, brainstorm, choose the right objects, their types, what their relationships will be, ... "so many things to consider". So why bother? Why care about objects?

Let me show you a simple example of why OOP is not a bad idea. Taking the code we've written for tossing a coin, we can generate two series of tosses. The first experiment involves tossing a coin five times, and then computing the proportion of heads:

```
# random seed
set.seed(534)

# five tosses
five <- toss(coin, times = 5)

# prop of heads in five
sum(five == "heads") / length(five)
```

```
## [1] 0.6
```

The second experiment involves tossing a coin six times and computing the proportion of heads:

```
# six tosses
six <- toss(coin, times = 6)

# prop of heads in six
sum(six == "heads") / length(five)
```

```
## [1] 0.8
```

The code works ... except that there is an error; the number of heads in `six` is being divided by 5 instead of 6. R hasn't detected this error: it doesn't know that the division has to be done using `length(six)`.

Wouldn't it be prefarable to have some mechanism that prevented this type of errors from happening? Bugs will always be part of any programming activity, but it is better to minimize certain types of errors like the one above.

S3 Classes and Objects

S3 classes are widely-used, in particular for statistical models in the `"stats"` package. S3 classes are very informal in that there is not a formal definition for an S3 class.

S3 objects are usually built on top of lists, or atomic vectors with attributes. You can also turn functions into S3 objects.

To make an object an instance of a class, you just take an existing base object and set the `"class"` attribute. You can do that during creation of the object with `structure()`, or after the object has been createad with `class<-()`.

```
# object coin
coin1 <- structure(c("heads", "tails"), class = "coin")

# object coin
coin2 <- c("heads", "tails")
class(coin2) <- "coin"
```

You can determine the class of any object using `class(x)`

```
class(coin1)
```

```
## [1] "coin"
```

You can also determine if an object inherits from a specific class using `inherits()`

```
inherits(coin2, "coin")
```

```
## [1] TRUE
```

Generic and Specific Methods

A coin could have a function `flip()`:

```
flip <- function(coin, times = 1) {
  sample(coin, size = times, replace = TRUE)
}

flip(coin1)
```

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

The issue with the way `flip()` is defined, is that you can pass it any type of vector (not necessarily of class `"coin"`), and it will still work:

```
flip(c('tic', 'tac', 'toe'))
```

```
## [1] "tic"
```

To create a function `flip()` that only works for objects of class `"coin"`, we could add a `stop()` condition that checks if the argument `coin` is of the right class:

```
flip <- function(coin, times = 1) {
  if (class(coin) != "coin") {
    stop("\nflip() requires an object 'coin'")
  }
  sample(coin, size = times, replace = TRUE)
}

# ok
flip(coin1)
```

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

```
# bad coin
flip(c('tic', 'tac', 'toe'))
```

```
## Error in flip(c("tic", "tac", "toe")):
## flip() requires an object 'coin'
```

A more formal strategy, and one that follows OOP principles, is to create a flip **method**. In R, many functions are actually methods: e.g. `print()`, `summary()`, `plot()`, `str()`, etc.

```
# print method
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x7fb87a9ff920>
## <environment: namespace:base>
```

```
# plot method
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fb87adfc5c8>
## <environment: namespace:graphics>
```

These types of functions are not really one unique function, they typically comprise a collection or family of functions for printing objects, computing summaries, plotting, etc. Depending on the class of the object, a generic method will look for a specific function for that class:

```
# methods for objects "matrix"
methods(class = "matrix")
```

```
##  [1] anyDuplicated as.data.frame as.raster     boxplot       coerce
##  [6] determinant   duplicated    edit          head          initialize
## [11] isSymmetric   Math          Math2         Ops           relist
## [16] subset        summary       tail          unique
## see '?methods' for accessing help and source code
```

`flip` method

When implementing new methods, you begin by creating a **generic** method with the function `UseMethod()`:

```
flip <- function(x, ...) UseMethod("flip")
```

The function `UseMethod()` allows you to declare the name of a method. In this example we are telling R that the function `flip()` is now a generic `"flip"` method. Note the use of `"..."` in the function definition, this will allow you to include more arguments when you define specific methods based on `"flip"`.

A generic method alone is not very useful. You need to create specific cases for the generic. In our example, we only have one class `"coin"`, so that is the only class we will allow `flip` to be applied on. The way to do this is by defining `flip.coin()`:

```
flip.coin <- function(x, times = 1) {
  sample(x, size = times, replace = TRUE)
}
```

The name of the method, `"flip"`, comes first, followed by a dot `"."`, followed by the name of the class, `"coin"`. To use the `flip()` method on a `"coin"` object, you don't really have to call `flip.coin()`; calling `flip()` is enough:

```
flip(coin1)
```

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

How does `flip()` work? Becasue `flip()` is now a generic method, everytime you use it, R will look at the class of the input, and see if there is an associated `"flip"` method. In the previous example, `coin1` is an object of class `"coin"`, for which there is a specific `flip.coin()` method. Thus using `flip()` on a `"coin"` object works fine.

Now let's try `flip()` on the character vector `c('tic', 'tac', 'toe')`:

```
# no flip() method for regular vectors
flip(c('tic', 'tac', 'toe'))
```

```
## Error in UseMethod("flip"): no applicable method for 'flip' applied to an object of class "
```

When you try to use `flip()` on an object that is not of class `"coin"`, you get a nice error message.

A more robust `"coin"` class

Let's review our class `"coin"`. The way we defined a coin object was like this:

```
# object coin
coin1 <- c("heads", "tails")
class(coin1) <- "coin"
```

While this definition is good to illustrate the concept of an object, its class, and how to define generic methods, it is a very loose-defined class. One could create a `"coin"` out of `c('tic', 'tac', 'toe')`, and then use `flip()` on it:

```
ttt <- c('tic', 'tac', 'toe')
class(ttt) <- "coin"

flip(ttt)
```

```
## [1] "tic"
```

I would like to have a more formal definition of a coin. For instance, it makes more sense to require that a coin should only have two sides. In this way, the vector `ttt` would not be a valid coin.

For convenience purposes, we can define a class **constructor** function to initialize a `"coin"` object:

```
coin <- function(object = c("heads", "tails")) {
  class(object) <- "coin"
  object
}

# default coin
coin()
```

```
## [1] "heads" "tails"
## attr(,"class")
## [1] "coin"
```

```
# another coin
coin(c("h", "t"))
```

```
## [1] "h" "t"
## attr(,"class")
## [1] "coin"
```

To implement the requirement that a coin must have two sides, we can check for the length of the provided vector:

```
coin <- function(object = c("heads", "tails")) {
  if (length(object) != 2) {
    stop("\n'object' must be of length 2")
  }
  class(object) <- "coin"
  object
}

# US penny
penny <- coin(c("lincoln", "shield"))
penny
```

```
## [1] "lincoln" "shield"
## attr(,"class")
## [1] "coin"
```

```
# invlaid coin
coin(ttt)
```

```
## Error in coin(ttt):
## 'object' must be of length 2
```

Because the `flip()` function simulates flips using `sample()`, we can take advantage of the argument `prob` to specify probabilities for each side of the coin. In this way, we can create *loaded* coins.

We can add a `prob` argument to the constructor function. This argument takes a vector of probabilities for each element in `object`, and we pass this vector as an attribute of the coin object. Furthermore, we can set a default `prob = c(0.5, 0.5)`, that is, a *fair* coin by default:

```r
coin <- function(object = c("heads", "tails"), prob = c(0.5, 0.5)) {
  if (length(object) != 2) {
    stop("\n'object' must be of length 2")
  }
  attr(object, "prob") <- prob
  class(object) <- "coin"
  object
}

coin()
```

```
## [1] "heads" "tails"
## attr(,"prob")
## [1] 0.5 0.5
## attr(,"class")
## [1] "coin"
```

Once again, we need to check for the validity of `prob`. Here is one possible function to check several aspects around `prob`: must be of length 2, probability values must be between 0 and 1, and the sum of these values must add up to 1:

```r
check_prob <- function(prob) {
  # if (!is.numeric(prob)) {
  #   stop("\n'prob' must be a numeric vector")
  # }
  if (length(prob) != 2 | !is.numeric(prob)) {
    stop("\n'prob' must be a numeric vector of length 2")
  }
  if (any(prob < 0) | any(prob > 1)) {
    stop("\n'prob' values must be between 0 and 1")
  }
  if (sum(prob) != 1) {
    stop("\nelements in 'prob' must add up to 1")
  }
  TRUE
}
```

Note that I'm adding a `TRUE` statement at the end of the function. This is just an auxiliary value to know if the function returns a valid `prob`. Now let's test it:

```r
# good prob
check_prob(c(0.5, 0.5))
check_prob(c(0.1, 0.9))
check_prob(c(1/3, 2/3))
check_prob(c(1/3, 6/9))
```

```r
# bad length
check_prob(1)
```

```
## Error in check_prob(1):
## 'prob' must be a numeric vector of length 2
```

```r
# bad length
check_prob(c(0.1, 0.2, 0.3))
```

```
## Error in check_prob(c(0.1, 0.2, 0.3)):
## 'prob' must be a numeric vector of length 2
```

```r
# negative probability
check_prob(c(-0.2, 0.8))
```

```
## Error in check_prob(c(-0.2, 0.8)):
## 'prob' values must be between 0 and 1
```

```r
# what should we do in this case?
check_prob(c(0.33, 0.66))
```

```
## Error in check_prob(c(0.33, 0.66)):
## elements in 'prob' must add up to 1
```

Here's the improved constructor function `coin()`:

```r
coin <- function(object = c("heads", "tails"), prob = c(0.5, 0.5)) {
  if (length(object) != 2) {
    stop("\n'object' must be of length 2")
  }
  check_prob(prob)
  attr(object, "prob") <- prob
  class(object) <- "coin"
  object
}

coin1 <- coin()
```

And the new definition of `flip.coin()`:

```r
flip.coin <- function(x, times = 1) {
  sample(x, size = times, replace = TRUE, prob = attr(x, 'prob'))
}
```

Let's flip a loaded coin:

```r
set.seed(2341)
load_coin <- coin(c('HEADS', 'tails'), prob = c(0.75, 0.25))
flip(load_coin, times = 6)
```

```
## [1] "HEADS" "HEADS" "HEADS" "HEADS" "HEADS" "tails"
```

Extending classes

We can extend the class `"coin"` and create a derived class for special types of coins. For instance, say we want to create a class `"quarter"`. One side of the coin refers to George Washington, while the other side refers to John Brown's Fort:

```r
quarter1 <- coin(c("washington", "fort"))
class(quarter1) <- c("quarter", "coin")
quarter1
```

```
## [1] "washington" "fort"
## attr(,"prob")
## [1] 0.5 0.5
## attr(,"class")
## [1] "quarter" "coin"
```

Our coin `quarter1` inherits from `"coin"`:

```r
inherits(quarter1, "coin")
```

```
## [1] TRUE
```

Likewise, we can create a class for a slightly unbalanced `"dime"`:

```r
dime1 <- coin(c("roosevelt", "torch"), prob = c(0.48, 0.52))
class(dime1) <- c("dime", "coin")
dime1
```

```
## [1] "roosevelt" "torch"
## attr(,"prob")
## [1] 0.48 0.52
## attr(,"class")
## [1] "dime" "coin"
```

Object "toss"

Because we are not only interested in tossing a coin, but also in keeping track of such tosses, it would be good to have another object for this purpose. How do you know that you need this new object class? Well, this is precisely an example that illustrates the process of programming in general, and OOP in particular. This kind of decisions require some (or a lot of) thinking, and brainstorming time. The more you understand a problem (i.e. phenomenon, process), the better you will be prepared to design what objects and methods you need to program.

While I was writing this material, I decided that the object "toss" should have the following information:

- all the outcomes from the series of tosses
- the total number of tosses
- the total number of heads
- the total number of tails

The most flexible type of data structure in R to store other data structures is a list. Having a vector of tosses, we can use a list to keep all the desired information:

```
flips <- flip(coin1, times = 6)

a <- list(
  tosses = flips,
  total = length(flips),
  heads = sum(flips == "heads"),
  tails = sum(flips == "tails")
)

a
```

```
## $tosses
## [1] "tails" "tails" "tails" "tails" "heads" "heads"
##
## $total
## [1] 6
##
## $heads
## [1] 2
##
## $tails
## [1] 4
```

For convenience purposes, we can write a **constructor** function, which I will call make_toss(). This function takes an input vector (i.e. a character vector with "heads" and "tails" elements), and it returns an object of class "toss":

```
make_toss <- function(coin, flips) {
  res <- list(
    coin = coin,
    tosses = flips,
    total = length(flips),
    heads = sum(flips == coin[1]),
    tails = sum(flips == coin[2]))
  class(res) <- "toss"
  res
}
```

Assuming that we have a `"coin"` object and a vector of flips, we can pass them to `make_toss()` and produce an object of class `"toss"`:

```
many_flips <- flip(coin1, times = 50)
a <- make_toss(coin1, many_flips)
class(a)
```

```
## [1] "toss"
```

Main Function `toss()`

Now that we have the constructor function, we can encapsulate it in a *master* function `toss()`:

```
toss <- function(coin, times = 1) {
  flips <- flip(coin, times)
  make_toss(coin, flips)
}
```

You may ask why we need a function `make_toss()`, and another function `toss()`. These two functions give the impression that we are just duplicating code, and following a convoluted path. Can't we just write a single function `suppertoss()` that does everything at once?:

```
supertoss <- function(coin, times = 1) {
  flips <- flip(coin, times = times)
  res <- list(
    coin = coin,
    tosses = flips,
    total = length(flips),
    heads = sum(flips == coin[1]),
    tails = sum(flips == coin[2]))
  class(res) <- "toss"
  res
}
```

The short answer is: yes, you can. And probably this is what most beginners tend to do. The reason why I decided to break things down into simpler and smaller functions is because I went already through a couple of implementations, and realized that it was better to have the auxiliary function `make_toss()`.

So here's a brief recap of the main functions we have so far:

- `coin()` is a constructor function to create objects of class `"coin"`.
- `flip()` is a generic `"flip"` method.
- `flip.coin()` is the specific `"flip"` method to be used on `"coin"` objects.
- `make_toss()` is an auxiliary function that takes a `"coin"` and a vector of flips, and which produces an object `"toss"`.
- `toss()` is just the wrapper (this is the function designed for the user)

```
toss(quarter1, times = 4)
```

```
## $coin
## [1] "washington" "fort"
## attr(,"prob")
## [1] 0.5 0.5
## attr(,"class")
## [1] "quarter" "coin"
##
## $tosses
## [1] "washington" "fort"       "fort"       "washington"
##
## $total
## [1] 4
##
## $heads
## [1] 2
##
## $tails
## [1] 2
##
## attr(,"class")
## [1] "toss"
```

`toss()` works ok, and you can try it with different values for `times`. The only issue is that a distracted user could pass an unexpected value for the argument `times`:

```
toss(quarter1, times = -4)
```

```
## Error in sample.int(length(x), size, replace, prob): invalid 'size' argument
```

R produces an error when `times = -4`, but it's an error that may not be very helpful for the user. The error message clearly says that `'size'` is an invalid argument, but `toss()` just has one argument: `times`.

To be more user friendly, among other reasons, it would be better to check whether `times` has a valid value. One way to do that is to include a conditional statement:

```
toss <- function(coin, times = 1) {
  if (times <= 0) {
    stop("\nargument 'times' must be a positive integer")
  }
  flips <- flip(coin, times = times)
  make_toss(coin, flips)
}

# this works ok
toss(quarter1, 5)
```

```
## $coin
## [1] "washington" "fort"
## attr(,"prob")
## [1] 0.5 0.5
## attr(,"class")
## [1] "quarter" "coin"
##
## $tosses
## [1] "washington" "washington" "fort"       "fort"       "washington"
##
## $total
## [1] 5
##
## $heads
## [1] 3
##
## $tails
## [1] 2
##
## attr(,"class")
## [1] "toss"
```

```
# this doesn't work, but the error message is clear
toss(quarter1, -4)
```

```
## Error in toss(quarter1, -4):
## argument 'times' must be a positive integer
```

Once again, it is good practice to write short functions that preferably do one thing. In this case, we could define a checking function for `times`:

```
check_times <- function(times) {
  if (times <= 0 | !is.numeric(times)) {
    stop("\nargument 'times' must be a positive integer")
  } else {
    TRUE
  }
}
```

and then include `check_times()` inside `toss()`:

```
toss <- function(coin, times = 1) {
  check_times(times)
  flips <- flip(coin, times = times)
  make_toss(coin, flips)
}

toss(quarter1, 5)
```

```
## $coin
## [1] "washington" "fort"
## attr(,"prob")
## [1] 0.5 0.5
## attr(,"class")
## [1] "quarter" "coin"
##
## $tosses
## [1] "washington" "washington" "fort"       "fort"       "washington"
##
## $total
## [1] 5
##
## $heads
## [1] 3
##
## $tails
## [1] 2
##
## attr(,"class")
## [1] "toss"
```

Print Method

Typically, most classes in R have a dedicated printing method. To create such a method we use the generic function `print()` like so:

```r
# print method for object of class "toss"
print.toss <- function(x, ...) {
  cat('object "toss"\n')
  cat(sprintf('coin: "%s", "%s"', x$coin[1], x$coin[2]), "\n")
  cat("total tosses:", x$total, "\n")
  cat(sprintf("num of %s:", x$coin[1]), x$heads, "\n")
  cat(sprintf("num of %s:", x$coin[2]), x$tails, "\n")
  invisible(x)
}
```

By convention, `print` methods return the value of their principal argument invisibly. The `invisible` function turns off automatic printing, thus preventing an infinite recursion when printing is done implicitly at the session level.

After a `print` method has been defined for an object `"toss"`, everytime you type an object of such class, R will earch for the corresponding method and display the output accordingly:

```r
# testing print method
a <- make_toss(coin1, many_flips)
a
```

```
## object "toss"
## coin: "heads", "tails"
## total tosses: 50
## num of heads: 25
## num of tails: 25
```

Summary Method

For most purposes the standard `pint` method will be sufficient output, but some times a more extensive display is required. This can be done with a `summary`. To define this type of method we use the function `summary()`.

```r
summary.toss <- function(object) {
  structure(object, class = c("summary.toss", class(object)))
}

print.summary.toss <- function(x, ...) {
  cat('summary "toss"\n\n')
  cat(sprintf('coin: "%s", "%s"', x$coin[1], x$coin[2]), "\n")
  cat("total tosses:", x$total, "\n\n")
  cat(sprintf("num of %s:", x$coin[1]), x$heads, "\n")
  cat(sprintf("prop of %s:", x$coin[1]), x$heads/x$total, "\n\n")
  cat(sprintf("num of %s:", x$coin[2]), x$tails, "\n")
  cat(sprintf("prop of %s:", x$coin[2]), x$tails/x$total, "\n")
  invisible(x)
}
```

Let's test it:

```
summary(a)
```

```
## summary "toss"
##
## coin: "heads", "tails"
## total tosses: 50
##
## num of heads: 25
## prop of heads: 0.5
##
## num of tails: 25
## prop of tails: 0.5
```

Plot Method

We can also define a `plot` method for objects of class `"toss"`:

What we want to plot of an object `"toss"` is the series of realtive frequencies (of either `"heads"` ot `"tails"`). This means we need to create a couple of auxiliary functions:

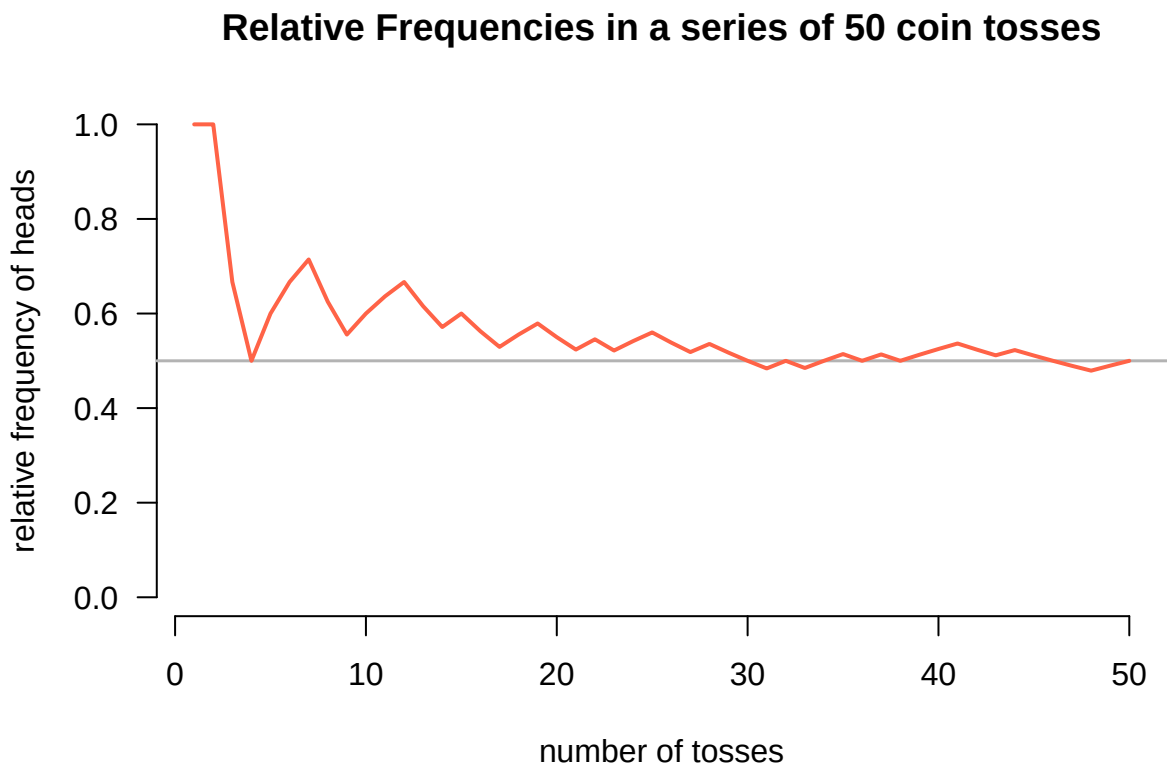
```
head_freqs <- function(x) {
  cumsum(x$tosses == x$coin[1]) / 1:x$total
}


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))
  }
}
```

```
plot.toss <- function(x, side = 1, ...) {
  freqs <- frequencies(x, side = side)
  plot(1:x$total, freqs, type = "n", ylim = c(0, 1), las = 1,
       xlab = "number of tosses", bty = "n",
       ylab = sprintf("relative frequency of %s", x$coin[side]))
  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))
}
```

Let's test our `plot` method:

```
plot(a)
```



Replacement Method

Replacement functions are those calls like `x[1] <- 3`. The function behind this expression is the replacement `"[<-"()` function. We can also create a replacement function for a given class using the notation `"[<-.class"`, where `class` is the name of the class:

```
"[<-.toss" <- function(x, i, value) {
  if (value != x$coin[1] & value != x$coin[2]) {
    stop(sprintf('\nreplacing value must be %s or %s', x$coin[1], x$coin[2]))
  }
  x$tosses[i] <- value
  make_toss(x$coin, x$tosses)
}
```

Test it:

```r
set.seed(3752)
b <- toss(dime1, times = 5)
b$tosses
```

```
## [1] "roosevelt" "roosevelt" "roosevelt" "torch"     "torch"
```

```r
# replacement
b[1] <- "torch"
b$tosses
```

```
## [1] "torch"     "roosevelt" "roosevelt" "torch"     "torch"
```

Replacing out of original range:

```r
set.seed(3752)
b <- toss(dime1, times = 5)
b$tosses
```

```
## [1] "roosevelt" "roosevelt" "roosevelt" "torch"     "torch"
```

```r
# replacement
b[6] <- "torch"
b
```

```
## object "toss"
## coin: "roosevelt", "torch"
## total tosses: 6
## num of roosevelt: 3
## num of torch: 3
```

What about this?

```r
set.seed(3752)
b <- toss(dime1, times = 5)
b$tosses
```

```
## [1] "roosevelt" "roosevelt" "roosevelt" "torch"     "torch"
```

```r
# replacement
b[10] <- "torch"
b
```

```
## object "toss"
## coin: "roosevelt", "torch"
## total tosses: 10
## num of roosevelt: NA
## num of torch: NA
```

```
"[<-.toss" <- function(x, i, value) {
  if (value != x$coin[1] & value != x$coin[2]) {
    stop(sprintf('\nreplacing value must be %s or %s', x$coin[1], x$coin[2]))
  }
  if (i > x$total) {
    stop("\nindex out of bounds")
  }
  x$tosses[i] <- value
  make_toss(x$coin, x$tosses)
}
```

Now we cannot replace if index is out of the original length:

```
set.seed(3752)
b <- toss(dime1, times = 5)
b$tosses
```

```
## [1] "roosevelt" "roosevelt" "roosevelt" "torch"     "torch"
```

```
# replacement
b[10] <- "torch"
```

```
## Error in `[<-.toss`(`*tmp*`, 10, value = "torch"):
## index out of bounds
```

Extraction Method

What if you want to know what is the value of toss in position 3? You could type something like this:

```
b$tosses[3]
```

```
## [1] "roosevelt"
```

Or you could create an extraction method that allows you to type `x[1]`. The function behind this expression is the extraction `"["()` function. We can also create a extraction function for a given class.

```
"[.toss" <- function(x, i) {
  x$tosses[i]
}
```

Test it:

```
set.seed(3752)
b <- toss(dime1, times = 5)
b$tosses
```

```
## [1] "roosevelt" "roosevelt" "roosevelt" "torch"     "torch"
```

```
b[1]
```

```
## [1] "roosevelt"
```

Is "toss"

Another common type of function for an object of a given class is `is.class()`-like functions: e.g. `is.list()`, `is.numeric()`, `is.matrix()`.

```
is.toss <- function(x) {
  inherits(x, "toss")
}

is.toss(a)
```

```
## [1] TRUE
```

```
is.toss(c("heads", "tails"))
```

```
## [1] FALSE
```

Addition Method

R comes with generic Math methods (see `?Math`). Among these generic methods we can find the `"+"` operator. This means that we can define our own *plus* method for objects of class `"toss"`. The idea is to be able to call a command like this:

```
# toss object
b <- toss(dime1, times = 5)

# add 5 more flips
b + 5
```

Here's one implementation of `"+.toss()" in which the first argument is an object of class`"toss"', and the second argument is a single positive number that will play the role of additional flips:

```
"+.toss" <- function(obj, incr) {
  if (length(incr) != 1 | incr <= 0) {
    stop("\ninvalid increament")
  }
  more_flips <- flip(obj$coin, times = incr)
  make_toss(obj$coin, c(obj$tosses, more_flips))
}
```

Remember that `"+"` is a binary operator, which means that writing a `"+"` method requires a function with two arguments. Let's try it:

```
# add four more tosses
mycoin <- coin()
seven <- toss(mycoin, times = 7)

# two more flips
seven + 2
```

```
## object "toss"
## coin: "heads", "tails"
## total tosses: 9
## num of heads: 4
## num of tails: 5
```

```
# three more flips
seven + 3
```

```
## object "toss"
## coin: "heads", "tails"
## total tosses: 10
## num of heads: 6
## num of tails: 4
```

Your turn: Minus Method

Try writing a subtraction method for objects of class `"toss"` using the minus operator `"="`.

```
# your "-" method
```

Coercion methods

To make things more intersting and flexible, we can try to design a coercion methods that allows us to take a vector of 1's and 0's and convert it into "heads" and "tails".

Say 1 = heads, and 0 = tails, it would be nice to do something like this:

```
as.toss(c(0, 1, 1, 0, 1, 0, 1, 0, 1))
```

So here's one solution:

```
as.toss <- function(x) {
  x_coin <- coin(unique(x))
  make_toss(x_coin, x)
}
```

Let's test it:

```
g <- as.toss(c(0, 1, 1, 0, 1, 0, 1, 0, 1))
g
```

```
## object "toss"
## coin: "0", "1"
## total tosses: 9
## num of 0: 4
## num of 1: 5
```

Method "tails()"

Function to get number of tails

```
# declaring the method
tails <- function(x) UseMethod("tails")

# tails for object "toss"
tails.toss <- function(x) {
  x$tails
}

tails(b)
```

```
## [1] 2
```

```
# try it on a different object
tails(1:4)
```

```
## Error in UseMethod("tails"): no applicable method for 'tails' applied to an object of class
```

Function to get number of heads

```
# declaring the method
heads <- function(x) UseMethod("heads")

# tails for object "toss"
heads.toss <- function(x) {
  x$heads
}

heads(b)
```

```
## [1] 3
```

```
# try it on a different object
heads(1:4)
```

```
## Error in UseMethod("heads"): no applicable method for 'heads' applied to an object of class
```