R S3 Classes

R S3 Classes

In this article, you will learn to work with S3 classes (one of the three class systems in R programming).

S3 class is the most popular and prevalent class in R programming language.

Most of the classes that come predefined in R are of this type. The fact that it is simple and easy to implement is the reason behind this.

How to define S3 class and create S3 objects?

S3 class has no formal, predefined definition.

Basically, a list with its class attribute set to some class name, is an S3 object. The components of the list become the member variables of the object.

Following is a simple example of how an S3 object of class student can be created.

```
> # create a list with required components
> s <- list(name = "John", age = 21, GPA = 3.5)
> # name the class appropriately
> class(s) <- "student"
> # That's it! we now have an object of class "student"
> s
$name
[1] "John"
$age
[1] 21
$GPA
[1] 3.5
attr(,"class")
[1] "student"
```

This might look awkward for programmers coming from C++, Python etc. where there are formal class definitions and objects have properly defined attributes and methods.

In R S3 system, it's pretty ad hoc. You can convert an object's class according to your will with objects of the same class looking completely different. It's all up to you.

How to use constructors to create objects?

It is a good practice to use a function with the same name as class (not a necessity) to create objects.

This will bring some uniformity in the creation of objects and make them look similar.

We can also add some integrity check on the member attributes. Here is an example. Note that in this example we use the `attr()` function to set the class attribute of the object.

Here is a sample run where we create objects using this constructor.

```
> s <- student("Paul", 26, 3.7)
> s
$name
[1] "Paul"
$age
[1] 26
$GPA
[1] 3.7
attr(,"class")
[1] "student"
> class(s)
[1] "student"
> s <- student("Paul", 26, 5)
Error in student("Paul", 26, 5) : GPA must be between 0 and 4
> # these integrity check only work while creating the object using constructor
> s <- student("Paul", 26, 2.5)
> # it's up to us to maintain it or not
> s$GPA <- 5
```

Methods and Generic Functions

In the above example, when we simply write the name of the object, its internals get printed.

In interactive mode, writing the name alone will print it using the `print()` function.

```
> s
$name
[1] "Paul"
$age
[1] 26
$GPA
[1] 3.7
attr(,"class")
[1] "student"
```

Furthermore, we can use `print()` with vectors, matrix, data frames, factors etc. and they get printed differently according to the class they belong to.

How does `print()` know how to print these variety of dissimilar looking object?

The answer is, `print()` is a generic function. Actually, it has a collection of a number of methods. You can check all these methods with `methods(print)`.

```
> methods(print)
[1] print.acf*
[2] print.anova*
...
[181] print.xngettext*
[182] print.xtabs*
Non-visible functions are asterisked
```

We can see methods like `print.data.frame` and `print.factor` in the above list.

When we call `print()` on a data frame, it is dispatched to `print.data.frame()`.

If we had done the same with a factor, the call would dispatch to `print.factor()`. Here, we can observe that the method names are in the form `generic_name.class_name()`. This is how R is able to figure out which method to call depending on the class.

Printing our object of class "`student`" looks for a method of the form `print.student()`, but there is no method of this form.

So, which method did our object of class "`student`" call?

It called `print.default()`. This is the fallback method which is called if no other match is found. Generic functions have a default method.

There are plenty of generic functions like `print()`. You can list them all with `methods(class="default")`.

```
> methods(class="default")
[1] add1.default*        aggregate.default*
[3] AIC.default*         all.equal.default
...
```

How to write your own method?

Now let us implement a method `print.student()` ourself.

```
print.student <- function(obj) {
cat(obj$name, "\n")
cat(obj$age, "years old\n")
cat("GPA:", obj$GPA, "\n")
}
```

Now this method will be called whenever we `print()` an object of class "`student`".

In S3 system, methods do not belong to object or class, they belong to generic functions. This will work as long as the class of the object is set.

```
> # our above implemented method is called
> s
Paul
26 years old
GPA: 3.7
> # removing the class attribute will restore as previous
> unclass(s)
$name
[1] "Paul"
$age
[1] 26
$GPA
[1] 3.7
```

Writing Your Own Generic Function

It is possible to make our own generic function like `print()` or `plot()`. Let us first look at how these functions are implemented.

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x0674e230>
<environment: namespace:base>
> plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x04fe6574>
<environment: namespace:graphics>
```

We can see that they have a single call to `UseMethod()` with the name of the generic function passed to it. This is the dispatcher function which will handle all the background details. It is this simple to implement a generic function.

For the sake of example, we make a new generic function called `grade`.

```
grade <- function(obj) {
UseMethod("grade")
}
```

A generic function is useless without any method. Let us implement the default method.

```
grade.default <- function(obj) {
cat("This is a generic function\n")
}
```

Now let us make method for our class "`student`".

```
grade.student <- function(obj) {
cat("Your grade is", obj$GPA, "\n")
}
```

A sample run.

```
> grade(s)
Your grade is 3.7
```

In this way, we implemented a generic function called `grade` and later a method for our class.