Chapter 9

S3 vs. S4 Objects

References:

- Freidrich Leisch’s lecture on S4 classes and methods

phylobase and ouch are written in the newer S4 class system (the one which you’ve been using and learning is the S3 class system). The main difference between these two systems is in the degree to which they follow the object-oriented programming model.

9.1 What is an object?

R works on objects:

- Objects are ways of bundling parts of programs into small, manageable pieces.
- Objects are simply a definition for a type of data to be stored
e.g., data vector, matrix, array, data frame, list, function
- An object is a component of a program that knows how to perform certain actions and to interact with other pieces of the program.
- Functions can be described as ”black boxes” that take an input and spit out an output. Objects can be thought of as ”smart” black boxes. That is, objects can know how to do more than one specific task (method or behavior), and they can store their own set of data.
CHAPTER 9.  S3 VS. S4 OBJECTS

Table 9.1: Attributes of Hero characters.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health</td>
<td>16</td>
</tr>
<tr>
<td>Strength</td>
<td>12</td>
</tr>
<tr>
<td>Agility</td>
<td>14</td>
</tr>
<tr>
<td>WeaponType</td>
<td>&quot;mace&quot;</td>
</tr>
<tr>
<td>ArmorType</td>
<td>&quot;leather&quot;</td>
</tr>
</tbody>
</table>

Table 9.2: Behaviors or methods of Hero characters.

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>move through the maze</td>
</tr>
<tr>
<td>attack monsters</td>
</tr>
<tr>
<td>pick up treasure</td>
</tr>
</tbody>
</table>

- It is an abstraction: Objects are something that have attributes (values) and behaviors (actions). These are sometimes called states and methods. These are formally defined in the object definition.

9.2 Object example: A Medieval Video Game (remember Dungeons and dragons?)

We have two types of players: Monsters and Heros. For the hero character, we need to store the values of certain attributes (Table ??):.

Heros must also be able to certain behaviors (which we will call methods):

These attributes and behaviors completely define the Hero. Modules may be written that know how to interpret (interact with) heros.

9.3 S3 Classes

Similarly, S3 classes have attributes and methods. If we create a data.frame, we can ask R what its attributes and methods available are:

```r
> flies <- data.frame(species=c("melanogaster", "silvestris", "heteroneura"), + headW=c(3.5, 5, 12))
> attributes(flies)

$names
[1] "species" "headW"
```
$row.names
[1] 1 2 3

class
[1] "data.frame"

> methods(class="data.frame")

1 $<-.data.frame Math.data.frame Ops.data.frame
4 Summary.data.frame [.data.frame [<-.data.frame
7 [[.data.frame ]<[-.data.frame aggregate.data.frame
10 as.data.frame.data.frame as.list.data.frame as.matrix.data.frame
13 by.data.frame cbind.data.frame dim.data.frame
16 dimnames.data.frame dimnames<-.data.frame duplicated.data.frame
19 edit.data.frame* format.data.frame formula.data.frame*
22 get Covariate.data.frame* getGroups.data.frame* getResponse.data.frame*
25 head.data.frame* is.na.data.frame mean.data.frame
28 merge.data.frame na.exclude.data.frame* na.omit.data.frame*
31 plot.data.frame* print.data.frame prompt.data.frame*
34 rbind.data.frame row.names.data.frame row.names<-.data.frame
37 rowsum.data.frame split.data.frame split<-.data.frame
40 stack.data.frame* str.data.frame* subset.data.frame
43 summary.data.frame t.data.frame tail.data.frame*
46 transform.data.frame unique.data.frame unstack.data.frame*
49 within.data.frame

Non-visible functions are asterisked

Programmers can write methods specifically for the classes that they define. For example, let’s see what methods are available for class phylo objects from the ape package:

> require(ape)

> methods(class="phylo")

1 AIC.phylo all.equal.phylo
3 as.hclust.phylo as.matching.phylo
5 coalescent.intervals.phylo cophenetic.phylo
7 deviance.phylo identify.phylo
9 logLik.phylo makeLabel.phylo
11 plot.phylo print.phylo
13 reorder.phylo skyline.phylo
15 summary.phylo
CHAPTER 9. S3 VS. S4 OBJECTS

9.3.1 No Validation

However, the S3 system also lacks features that object oriented languages often have, mostly related to a less structured language definition. For users, problems can arise when they accidentally make bad objects and crash their code. Under the S3 system, the class of an object is simply assigned via the class attribute.

```r
> tree <- "This is not a phylogenetic tree"
> class(tree) <- "phylo"
> attributes(tree)

$class
[1] "phylo"
```

There is no validation or checking that the objects have appropriate contents. R just tries to do what it can with the crazy object. Worse things will happen if you try to plot this fake tree.

9.3.2 Methods dispatch

One of the beauties of methods is that the user doesn’t have to worry about remembering the package-specific name of the function they want to use. They just use familiar generics such as `plot`, `summary`, and so on. The reason why this works is because of **methods dispatch**. The function looks at the class of the object that is given as its argument, and then it calls the correct function.

With S3 methods dispatch, however, it is rather clunky for programmers to write methods for every variation of parameters. As a consequence, sometimes you get the wrong method called. For example, here is something you may have experienced yourself. Take Fisher’s famous iris dataset. You may have wanted a boxplot of some continuous character by species, but instead gotten a scatterplot:

```r
> data(iris)
> op <- par(no.readonly = TRUE)
> par(mfrow=c(1,2))
> with(iris, plot(Sepal.Length, Species))
> with(iris, plot(Species, Sepal.Length))
> par(op)
```
The plot method dispatched a scatter plot when the first argument was a continuous variable, but dispatched a boxplot when the first argument was a factor. It is because R is trying to figure out what you want based on the supplied arguments. In this case, it is the first argument that determines the method, not the two arguments together.

9.4 S4 Classes

The S4 class system is considered to follow the principles of object oriented programming because:

1. New objects of any class must be specifically created using the constructor function `new()`. Programmers write a function to create new objects so that users never have to make calls to `new()`. Part of this process is validation, a series of checks that programmers write to make sure that object supplied by users are actually valid for that class. For users, this is a very nice feature that will prevent errors down the line which can be very confusing and difficult to trace.
2. Classes and methods can be inherited. This is mostly of benefit to programmers, but it is of benefit to users as well because there is a good chance that methods for one class of object can be used for related classes.

3. Programmers using S4 classes must write more consistent code, which makes it easier to extend packages and build a more functional family of packages. This consistency and predictability in the code makes it much easier for users to learn new packages.

Currently, *ouch* and *phylobase* are written in S4, and *ade4* will move to S4 in the next revision. More packages will soon follow.

### 9.4.1 What are the differences for users?

There are three main differences for users. We will illustrate these with examples in the next chapter on *phylobase*.

1. Accessing help
2. Creating objects
3. Accessing internal elements of objects