Advanced Statistical Programming in R

Object-Oriented Programming in R

Thomas Mailund

Table of Contents

Introduction

Welcome to the *Advanced Statistical Programming in R* series and this book, *Object Oriented Programming in R*. I wrote the series, and this book, to have teaching material beyond the typical introductory level most textbooks on R have. It covers some more advanced techniques used in R programming such as fully exploiting functional programming, writing meta-programs (programming the actual language structures), and writing domain specific languages to embed in R.

About the series

The *Advanced Statistical Programming in R* series is intended to consist of short single-topic books where each book can be used alone for teaching or learning R. That said, there will, of course, be some dependencies in topics, if not in content, among the books. For instance, functional programming is essential to understand for any serious R programming and the first book in the series covers that. Reading the other books without understanding functional programming will not be fruitful. However, if you are already familiar with functional programming, then you can safely skip the first book.

For each book, I will make clear what I think the prerequisites for reading the book are, but for the entire series I will expect you to be already familiar with programming and the basic R you will see in any introductory book. None of the books will give you a tutorial introduction to the R programming language.

If you have used R before for data analysis and are familiar with writing expressions and functions, and want to take it further and write more advanced R code, then the series is for you.

About this book

This book is intended to give an introduction to objects and classes in R, and how object-oriented programming is done in R. Object-oriented programming is based on the concept of *objects* and on designing programs in terms of operations, one can do with objects and how objects communicate with other objects.

This is often thought of in terms of objects with states, where operations on objects change the object state. In R, data is immutable, so you don't write code where you change an object's state. Rather, you work with objects as values and operations on objects create new objects when you need new "state".

Objects and classes in R are more like abstract data structures. You have values and associated operations you can do on these values. Such abstract data structures are implemented differently in different programming languages. Most object-oriented languages implement them using classes and class hierarchies while many functional languages define them using some kind of type specifications that define which functions can be applied to objects.

R is dynamically typed, so you do not specify abstract data types through a type specification. The operations you can do on objects is simply determined by which functions you can call on the objects. You can still think of this as specifications of abstract data structures. However, they are just implicitly defined.

Abstract data structures can be implemented in different ways, that is what makes them *abstract*, and the way to separate implementation from an interface is through *polymorphic* or *generic* functions, a construction founded on object oriented programming. Generic functions are implemented through a *class* mechanism, also derived from object oriented programming. The functions implemented by a class determines the interface of objects in the class, and by constructing hierarchies of classes, you can share the implementation of common functions between classes.

Object oriented programming was not built into the R language initially but was added later, and unfortunately, more than one object-oriented system was added. There are actually three different ways to implement object oriented constructions in R, each with different pros and cons, and these three systems do not operate well together. I will cover all three in this book, but put most emphasis on the S3 system which is the basis of the so-called "tidy verse", the

packages such as `tidyr`, `dplyr`, `ggplot2`, etc. that forms the basis of most data analysis pipelines these days.

When developing your own software, I will strongly recommend that you stick to one object oriented system instead of mixing them, but which is a matter of taste and which other packages your code is intended to work with.

Most books I have read on object-oriented programming, and the classes I have taken on object-oriented programming, have centred on object-oriented modelling and software design. There, the focus is on how object-orientation can be used to structure how you think about your software and how the software can reflect physical or conceptual aspects of the world that you try to model in your software. If for instance, you implement software for dealing with accounting you would model accounts as objects with operations for inserting and withdraw money. You would try to, as much as possible, mapping concepts from the problem domain to software as directly as possible. This is a powerful approach to designing your software, but there are always aspects of software that does not readily fit into such modelling. Especially when it comes to algorithmic programming and design of data structures. Search trees and sorting algorithms, for instance, are usually not reflecting anything concrete in a problem domain.

Object-oriented programming, however, is also a very powerful tool to use when designing algorithms and data structures. The way I was taught programming, algorithms and data structures were covered in separate classes from where I was taught object-orientation. Combining object-orientation and algorithmic programming were something I had to teach myself by writing software. I think this was a pity since the two really fit together well.

In this book, I will try to cover object-orientation both as a modelling technique for designing software but also as a tool for developing reusable algorithmic software. Polymorphism, a cornerstone of object-oriented programming, lends itself readily to developing flexible algorithms and to combining different concrete implementations of abstract data types to tailor abstract algorithms to concrete problems. A main use of R is machine learning and data science where efficient, and flexible, algorithms are more important than modelling a problem domain, so much of the book will focus on those aspects of object-oriented programming.

To read this book, you need to know the fundamentals of R programming: how to manipulate data and how to write functions. We will not see particular complex R programming, so you do not need a fundamental knowledge of how to do functional programming in R, but should you want to learn how to, I

suggest reading the first book in this series which is exactly on that. You should be able to follow the book without having read it, though.

Classes and generic functions

R's approach to object-oriented programming is through *generic functions* and *classes*. As mentioned in the introduction, R actually has three systems for implementing this, called S3, S4, and RC. In this chapter, I will only describe the S3 system, which is the simplest of the three, but I will return to the other two systems in later chapters.

Generic functions

The term *generic functions* refers to functions that can be used on more than one data type. Since R is dynamically typed, which means that there is no check of type consistency before you run your programs, type checking is really only a question of whether you can manipulate data in the way your functions attempts to. This is also called "duck typing" from the phrase "if it walks like a duck. . . ". If you can do the operations you want to do on a data object, then it has the right type. Where generic functions come into play is when you want to do the same semantic operation on objects of different types, but where the implementation of how that operation is done depends on the concrete types. Generic functions are functions that work differently on different types of objects. They are therefore also known as *polymorphic functions*.

To take this down from the abstract discussion to something more concrete, let us consider an abstract data type, say a stack. A stack is defined by the operations we can do on it:

- get the top element
- pop the first element off a stack
- push a new element to the top of the stack

To have a base case for stacks we typically also want a way to

- create an empty stack
- check if a stack is empty

These five operations define what a stack *is*, but we can implement a stack in many different ways. Defining a stack by the operations, that we can do on stacks, makes it an abstract data type. To implement a stack we need a concrete implementation.

In a statically typed programming language, we would define the type of a stack by these operations. How this would be done depends on the type of programming language and the concrete language, but generally in statically typed functional languages you would define a *signature* for a stack—the functions and their type for the five operations—while in an object oriented language you would define an abstract superclass.

In R, the types are implicitly defined, but for a stack, we would also define the five functions. These functions would be generic and not actually have any implementation in them; the implementation goes into the concrete implementation of stacks.

Of the five functions defining a stack, one is special. Creating an empty stack does not work as a generic function. When we create a stack, we always need a concrete implementation. But the other four can be defined as generic functions. Defining a generic function is done using the `UseMethod` function, and the four functions can be defined as thus:

```
top <- function(stack) UseMethod("top")
pop <- function(stack) UseMethod("pop")
push <- function(stack, element) UseMethod("push")
is_empty <- function(stack) UseMethod("is_empty")
```

What `UseMethod` does here is dispatching to different concrete implementations of functions in the S3 object-oriented programming system. When it is called, it will look for an implementation of a function and call it with the parameters the generic function was called with. We will see how this lookup works shortly.

When defining generic functions, you can specify "default" functions as well. These are called when `UseMethod` cannot find a concrete implementation. These are mostly useful when it is possible actually to have some default behaviour

that works in most cases, so not all concrete classes need to implement them, but it is a good idea always to implement them, even if all they do is inform you that an actual implementation wasn't found.

```
top.default <- function(stack) .NotYetImplemented()
pop.default <- function(stack) .NotYetImplemented()
push.default <- function(stack, element) .NotYetImplemented()
is_empty.default <- function(stack) .NotYetImplemented()
```

Classes

To make concrete implementations of abstract data types we need to use *classes*. In the S3 system, you create a class, and assign a class to an object, just by setting an attribute on the object. The name of the class is all that defines it, so there is no real type checking involved. Any object can have an attribute called "class" and any string can be the name of a class.

We can make a concrete implementation of a stack using a vector. To define the class we just need to pick a name for it. We can use `vector_stack`. We create such a stack using a function for creating an empty stack, and in this function, we set the attribute "class" using the `class<-` modification function.

```
empty_vector_stack <- function() {
  stack <- vector("numeric")
  class(stack) <- "vector_stack"
  stack
}

stack <- empty_vector_stack()
stack
```

```
## numeric(0)
## attr(,"class")
## [1] "vector_stack"
```

```
attributes(stack)
```

```
## $class
## [1] "vector_stack"
```

```
class(stack)
```

```
## [1] "vector_stack"
```

The empty stack is a numeric vector, just because we need some type to give the empty vector, but pushing other values onto it will just force a type conversion, so we can also put other basic types into it. It is limited to basic data types since vectors cannot contain complex data; for that, we would need a list. If we need complex data, we could easily change the implementation to use a list instead of a vector.

We will push elements by putting them at the front of the vector, pop elements by getting everything except the first element of the vector, and of course get the top of a vector by just indexing the first element. Such an implementation can look like this:

```
top.vector_stack <- function(stack) stack[1]
pop.vector_stack <- function(stack) {
  new_stack <- stack[-1]
  class(new_stack) <- "vector_stack"
  new_stack
}
push.vector_stack <- function(element, stack) {
  new_stack <- c(element, stack)
  class(new_stack) <- "vector_stack"
  new_stack
}
is_empty.vector_stack <- function(stack) length(stack) == 0
```

You will notice that the names of the functions are composed of two parts. Before the "." you have the names of the generic functions that define a stack, and after the "." you have the class name. This name format has semantic meaning; it is how generic functions figure out which concrete functions should be called based on the data provided to them.

When the generic functions call `UseMethod`, this function will check if the first value the generic function was called with has an associated class. If so, it will

get the name of that class and see if it can find a function with the name of the generic function (the name parameter given to `UseMethod`, not necessarily the name of the function that calls `UseMethod`) before a "." and the name of the class after the ".". If so, it will call that function. If not, it will look for a `.default` suffix instead and call that function if it exists.

This lookup mechanism gives semantic meaning to function names, and you really shouldn't use "."s in function names unless you want R to interpret the names in this way. The built-in functions in R are not careful about this—R has a long history and is not terribly consistent in how functions are named—but if you don't want to accidentally implement a function that works as a concrete implementation of a generic function, you shouldn't do it.

If we call `push` on a vector stack, it will, therefore, be `push.vector_stack` that will be called instead of `push.default`.

```
stack <- push(stack, 1)
stack <- push(stack, 2)
stack <- push(stack, 3)
stack
```

```
## [1] 1 2 3
## attr(,"class")
## [1] "vector_stack"
```

In the `push.vector_stack` we explicitly set the class of the concatenated new vector. If we didn't do this, we would just be creating a vector—the stack-ness of the second argument to `c` does not carry on to the concatenated vector—and we wouldn't return a stack. By setting the class of the new vector, we make sure that we return a stack.

The class isn't preserved when we remove the first element of the vector either, which is why we also have to set the class in the `pop.vector_stack` function explicitly. Otherwise, we would only have a stack the first time we pop an element, and after that, it would just be a plain vector. By explicitly setting the class we make sure that the function returns a stack that we can use with the generic functions again.

```
while (!is_empty(stack)) {
  stack <- pop(stack)
}
```

The remaining two functions, `top` and `is_empty`, do not return a stack object, and they are not supposed to, so we don't set the class attribute there.

We can avoid having to set the class attribute explicitly whenever we update it—that is, whenever we return a new value; we never actually modify an object here—by wrapping the class creation code in another function. Such a version could look like this:

```
make_vector_stack <- function(elements) {
  structure(elements, class = "vector_stack")
}
empty_vector_stack <- function() {
  make_vector_stack(vector("numeric"))
}
top.vector_stack <- function(stack) stack[1]
pop.vector_stack <- function(stack) make_vector_stack(stack[-1])
push.vector_stack <- function(stack, element) make_vector_stack(c(element, stack))
is_empty.vector_stack <- function(stack) length(stack) == 0
```

We are of course still setting the class attribute when we create an updated stack, we are just doing so implicitly by translating a vector into a stack using `make_vector_stack`. That function uses the `structure` function to set the class attribute, but otherwise just represent the stack as a vector just like before.

Polymorphism in action

The point of having generic functions is, of course, that we can have different implementations of the abstract operations. For the stack, we can try a different representation. The vector version has the drawback that each time we return a modified stack we need to create a new vector, which means copying all the elements in the new vector from the old. This makes the operations linear time in the vector size. Using a linked list, we can make them constant time operations. Such an implementation can look like this:

```
make_list_node <- function(head, tail) {
  list(head = head, tail = tail)
}
```

```
make_list_stack <- function(elements) {
  structure(list(elements = elements), class = "list_stack")
}
empty_list_stack <- function() make_list_stack(NULL)
top.list_stack <- function(stack) stack$elements$head
pop.list_stack <- function(stack) make_list_stack(stack$elements$tail)
push.list_stack <- function(stack, element) {
  make_list_stack(make_list_node(element, stack$elements))
}
is_empty.list_stack <- function(stack) is.null(stack$elements)

stack <- empty_list_stack()
stack <- push(stack, 1)
stack <- push(stack, 2)
stack <- push(stack, 3)
stack

## $elements
## $elements$head
## [1] 3
##
## $elements$tail
## $elements$tail$head
## [1] 2
##
## $elements$tail$tail
## $elements$tail$tail$head
## [1] 1
##
## $elements$tail$tail$tail
## NULL
##
##
##
##
## attr(,"class")
## [1] "list_stack"
```

Generally, when working with lists, we would use `NULL` as the base case to terminate a list. We cannot just wrap a list and use `NULL` this way when we

need to associate a class with the element. You cannot set the class to `NULL`. So instead we wrap the actual list inside another list where we set the class attribute. The real data is in the `elements` of this list, but except for having to use this list element of the object we just work with the list representation as we normally would with linked lists.

We now have two different implementations of the stack interface, but—and this is the whole point of having generic functions—code that uses a stack does not need to know which implementation it is operating on, as long as it only accesses stacks through the generic interface.

We can see this in action in the small function below that reverses a sequence of elements by first pushing them all onto a stack and then popping them off again.

```
stack_reverse <- function(empty, elements) {
  stack <- empty
  for (element in elements) {
    stack <- push(stack, element)
  }
  result <- vector(class(top(stack)), length(elements))
  for (i in seq_along(result)) {
    result[i] <- top(stack)
    stack <- pop(stack)
  }
  result
}
```

```
stack_reverse(empty_vector_stack(), 1:5)
```

```
## [1] 5 4 3 2 1
```

```
stack_reverse(empty_list_stack(), 1:5)
```

```
## [1] 5 4 3 2 1
```

Since the `stack_reverse` function only refers to the concrete stacks through the abstract interface, we can give it either a vector stack or a list stack, and it can do the same operations on both. As long as the concrete data structures

all implement the abstract interface correctly then code that only uses the generic functions will be able to work on any implementation.

One single concrete implementation is rarely superior in all cases, so it makes sense that we are able to combine algorithms working on abstract data types with concrete implementations depending on the particular problem we need to solve. For the two stack implementations they generally work equally well, but as discussed above, the stack implementation has a worst-case quadratic running time while the list implementation has a linear running time. For large stacks, we would thus expect the list implementation to be the best choice, but for small stacks, there is more overhead in manipulating the list implementation the way we do—having to do with looking up variable names and linking lists and such—so for short stacks, the vector implementation is faster.

```
library(microbenchmark)
microbenchmark(stack_reverse(empty_vector_stack(), 1:10),
               stack_reverse(empty_list_stack(), 1:10))
```

```
## Unit: microseconds
##                                       expr
##  stack_reverse(empty_vector_stack(), 1:10)
##    stack_reverse(empty_list_stack(), 1:10)
##      min       lq     mean   median       uq
##  195.787 217.2255 245.5698 229.8335 252.4785
##  235.270 257.0180 283.9289 271.5655 289.5300
##       max neval cld
##   965.714   100  a
##  1174.888   100   b
```

```
microbenchmark(stack_reverse(empty_vector_stack(), 1:1000),
               stack_reverse(empty_list_stack(), 1:1000))
```

```
## Unit: milliseconds
##                                         expr
##  stack_reverse(empty_vector_stack(), 1:1000)
##    stack_reverse(empty_list_stack(), 1:1000)
##       min       lq     mean   median       uq
##  28.22786 32.30359 37.77813 34.25044 36.49355
```

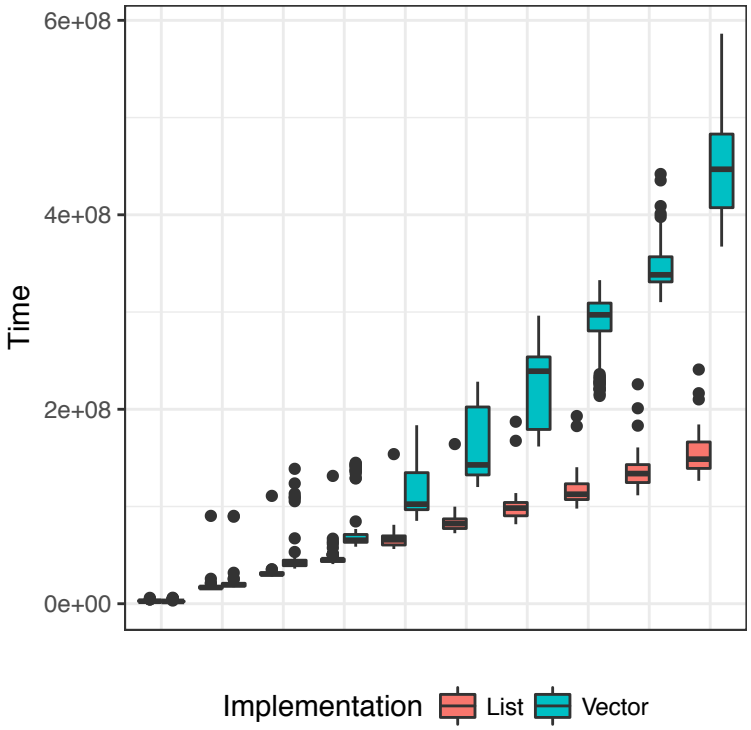

Figure 2.1: Time usage of reversal with two different stacks.

```
##  23.38964 24.40046 26.14922 25.47550 26.65488
##        max neval cld
##  124.57810   100   b
##   61.01179   100  a
```

Plotting the time usage for various length of stacks makes it even more evident that, as the stacks get longer, the list implementation gets relatively faster than the vector implementation.

Only for very short stacks would the vector implementation be preferable—the quadratic versus linear running time kicks in for very small n—but in general, different implementations will be preferred for different usages. By writing code that is polymorphic, we make sure that we can change the implementation of a data structure without having to modify the algorithms using it.

Designing interfaces

It is not just generic functions that are polymorphic. Any function that manipulates data only through generic functions is also polymorphic. The reversal function we implemented using a stack takes the empty stack as an argument, and this empty stack determines which actual stack implementation we use. Nowhere in the function do we refer to any details of a real implementation. If we had, instead, created an empty stack inside this function then, despite otherwise only accessing the implementation through the interface of the generic functions, the function would be bound to a single implementation.

To get the most out of polymorphism, you will want to design your functions to be as polymorphic as possible. This requires two things:

1. Don't refer to concrete implementations unless you really have to.
2. Any time you *do* have to refer to implementation details of a concrete type, do so through a generic function.

The reversal function is polymorphic because it doesn't refer to any concrete implementation. The choice of which concrete stack to use is determined by a parameter, and the operations it performs on the specific stack implementation all goes through generic functions.

It can be very tempting to break these rules in the heat of programming. Using a parameter to determine data structures in an algorithm isn't that difficult to do, but if you are writing an algorithm that uses several different data structures you might not want to have all the different concrete implementations as parameters. You really ought to do it, though. Just write a function that wraps the algorithm and provides implementations if you don't want to remember all the concrete data structures where the algorithm is needed. That way you get the best of both worlds.

More often, you will want to access the details of a concrete implementation. Imagine, for example, that you want to pop elements until you see a specific one, but *only* if that element is on the stack. If we are used to working with the vector implementation of the stack, then it would be natural to write a function like this:

```
pop_until <- function(stack, element) {
  if (element %in% stack) {
```

```
    while (top(stack) != element) stack <- pop(stack)
  }
  stack
}

library(magrittr)
vector_stack <- empty_vector_stack() %>%
  push(1) %>%
  push(2) %>%
  push(3) %T>% print

## [1] 3 2 1
## attr(,"class")
## [1] "vector_stack"

pop_until(vector_stack, 1)

## [1] 1
## attr(,"class")
## [1] "vector_stack"

pop_until(vector_stack, 5)

## [1] 3 2 1
## attr(,"class")
## [1] "vector_stack"
```

Here we use the `%in%` function to test if the element is on the stack (and we use the `magrittr` pipe operator to create a stack for our test). This works fine, as long as the stack is a vector stack, but it will *not* work if the stack is implemented as a list. You won't get an error message, the `%in%` test will just always return `FALSE`, so if you replace the stack implementation you have incorrect code that doesn't even inform you that it isn't working.

Relying on implementation details is the worst you can do to break the interface of polymorphic objects. Not only do you tie yourself into a single implementation, but you also tie yourself into exactly how that concrete data is implemented. If that implementation changes, your algorithm using it will

break. So now you either can't change the implementation, or you will have to change the algorithm that uses when it does. If you are lucky, you might get an error message if you break the interface, but as in the case we just saw (and you can try it yourself if you don't believe me), you won't even get that. The function will just always return the original stack, even when the element you want to pop to is on it.

```
list_stack <- empty_list_stack() %>%
  push(1) %>%
  push(2) %>%
  push(3)
pop_until(list_stack, 1)
```

If you write an algorithm that operates on a polymorphic object, stick to the interface it has if at all possible. For the `pop_until` function we can easily implement it using just the stack interface.

```
pop_until <- function(stack, element) {
  s <- stack
  while (!is_empty(s) && top(s) != element) s <- pop(s)
  if (is_empty(s)) stack else s
}
```

If you cannot achieve what you need using the interface, you should instead extend it. You can always write new generic functions that work on a class.

```
contains <- function(stack, element) UseMethod("contains")
contains.default <- function(stack, element) .NotYetImplemented()
contains.vector_stack <- function(stack, element) element %in% stack
```

You do not need to implement concrete functions for all implementations of an abstract data type to add a generic function. If you have a default implementation that gives you an error—and you have proper unit tests for any code you use—you will get an error if your algorithm attempts to use the function if it isn't implemented yet, and you can add it at that point.

Adding new generic functions is not as ideal as using the original interface in the first place if the abstract data type is from another package. If the

implementation in that package changes at a later point, your new generic function might break—and might break silently. Still, combined with proper unit tests, it is a much better solution than simply accessing the detailed implementation in your other functions.

Designing interfaces is a bit of an art. When you create your own abstract types, you want to think carefully about which operations the type should have. You don't want to have too many operations. That would make it harder for people implementing other versions of type; they would need to implement all the operations, and depending on what those operations are, this could involve a lot of work. On the other hand, you can't have too few operations, because then algorithms using the type will often have to break the interface to get to implementation details, which will break the polymorphism of those algorithms.

The abstract data types you learn about in an algorithms class are good examples of minimal yet powerful interfaces. They define the minimum number of operations necessary to get useful work done, yet still make implementations of concrete stacks, queues, dictionaries, etc. possible with minimal work.

When designing your own types, try to achieve the same kind of minimal interfaces.

The usefulness of polymorphism

Polymorphism isn't only useful for what we would traditionally call abstract data structures. Polymorphism gives you the means to implement abstract data structures, so algorithms work on the abstract interface and never need to know which concrete implementation they are operating on, but generic functions are useful for many cases that we do not traditionally think of as data structures.

In R, you often fit statistical models to data. Such models are not really data structures, but there is an abstract interface to them. You fit a concrete model, for example, a linear model, but once you have a fitted model, there are many common operations that are useful for all models. You might want to predict response variables for new data, or you might want to get the residuals of your fitted values. These operations are the same for all models—although how different models implement the operations will be different—and so they can benefit from being generic. Indeed they are. The functions `predict` and

`residuals`, which implements those two operations, are generic functions, and each model can implement its own version of them.

There is an extensive list of standard functions that are frequently used on fitted models, and all of these are implemented as generics. If you write analysis code that operates on fitted models using only those generic functions, you can change the model at any time and reuse all the code without modifying it.

The same goes for printing and plotting functions. Both `print` and `plot` are generic functions, and they have concrete implementations for different data types (and usually also for different fitted models). It is not something we think much about from day to day, but if we didn't have generic functions like these, we would need to use different functions for displaying vectors and for displaying matrices, for example.

Converting between different data types is also a frequent operation, and again polymorphism is highly useful (and frequently used in R). To translate a data structure into a vector, you use the `as.vector` function—an unfortunate name since it looks like a generic function `as` with a specialisation for `vector`, but actually is a generic function named `as.vector`. To translate a factor into a vector, it is the concrete implementation `as.vector.factor` that gets called.

An algorithm that needs to translate some input data into a vector can use the `as.vector` function and then doesn't have to worry about what the actual data is implemented as. As long as the data type has an implementation of the `as.vector` function.

Polymorphism and algorithmic programming

Polymorphism as a component of designing algorithms, and especially implementing algorithms, is not often covered in classes and textbooks but can be an important aspect of writing reusable software. Take something as simple as a sorting function. For many sorting algorithms, all you need to be able to for sorting elements is to determine whether one element is smaller than another. If you hardwire in an implementation of such an algorithm that the comparison used is interfering or floating point comparison, then you can only sort objects of these types. In general, if you hardwire comparisons, you need a different implementation for each type of elements you want to sort.

Because of this, most languages provide you with a generic sorting function as part of their runtime library where you can provide the comparison functionality

it should use, typically either as a function provided to the function or by allowing you to specify a comparison function for new types. Unfortunately, the `sort` function in R is not of this kind—it does allow you to define sorting for new types, but it wants its input to be in atomic form, so you cannot give it sequences of complex data types. Usually, you can change your data to a matrix or such and sort it this way, but if you actually have a list of complex data, you cannot use it.

We can easily implement our own function for doing this, however, and we can call it `sort_list`—not to be confused with the built-in function `sort.list` that actually does something else than sort lists...

Sorting lists

A straightforward implementation of merge sort could look like this:

```
merge_lists <- function(x, y) {
  if (length(x) == 0) return(y)
  if (length(y) == 0) return(x)

  if (x[[1]] < y[[1]]) {
    c(x[1], merge_lists(x[-1], y))
  } else {
    c(y[1], merge_lists(x, y[-1]))
  }
}

sort_list <- function(x) {
  if (length(x) <= 1) return(x)

  start <- 1
  end <- length(x)
  middle <- end %/% 2

  merge_lists(sort_list(x[start:middle]), sort_list(x[(middle+1):end]))
}
```

It gets the job done, but the merge function is quadratic in running time since it copies lists when it subscripts like `x[-1]` and `y[-1]` and when it combines

the results in the recursive calls. We can make a slightly more complicated function that does the merging in linear time using an iterative approach rather than a recursive:

```
merge_lists <- function(x, y) {
  if (length(x) == 0) return(y)
  if (length(y) == 0) return(x)

  i <- j <- k <- 1
  n <- length(x) + length(y)
  result <- vector("list", length = n)

  while (i <= length(x) && j <= length(y)) {
    if (x[[i]] < y[[j]]) {
      result[[k]] <- x[[i]]
      i <- i + 1
    } else {
      result[[k]] <- y[[j]]
      j <- j + 1
    }
    k <- k + 1
  }

  if (i > length(x)) {
    result[k:n] <- y[j:length(y)]
  } else {
    result[k:n] <- x[i:length(x)]
  }

  result
}
```

We are still copying in the recursive calls of the sorting function, but we are not copying more than we will merge later, so the asymptotic running time is okay at least.

With this function, we can sort lists of elements where `<` can be used to determine if one element is less than another. The built-in `<` function, however, doesn't necessarily work on your own classes.

```
make_tuple <- function(x, y) {
  result <- c(x,y)
  class(result) <- "tuple"
  result
}

x <- list(make_tuple(1,2),
          make_tuple(1,1),
          make_tuple(2,0))
sort_list(x)
```

```
## Warning in if (x[[i]] < y[[j]]) {: the condition
## has length > 1 and only the first element will be
## used

## Warning in if (x[[i]] < y[[j]]) {: the condition
## has length > 1 and only the first element will be
## used

## Warning in if (x[[i]] < y[[j]]) {: the condition
## has length > 1 and only the first element will be
## used
```

```
## [[1]]
## [1] 1 1
## attr(,"class")
## [1] "tuple"
##
## [[2]]
## [1] 1 2
## attr(,"class")
## [1] "tuple"
##
## [[3]]
## [1] 2 0
## attr(,"class")
## [1] "tuple"
```

There are several ways we can fix this. One option is to define a generic function for comparison, we could call it `less`, and then use that in the merge function.

```
merge_lists <- function(x, y) {
  if (length(x) == 0) return(y)
  if (length(y) == 0) return(x)

  i <- j <- k <- 1
  n <- length(x) + length(y)
  result <- vector("list", length = n)

  while (i <= length(x) && j <= length(y)) {
    if (less(x[[i]], y[[j]])) {
      result[[k]] <- x[[i]]
      i <- i + 1
    } else {
      result[[k]] <- y[[j]]
      j <- j + 1
    }
    k <- k + 1
  }

  if (i > length(x)) {
    result[k:n] <- y[j:length(y)]
  } else {
    result[k:n] <- x[i:length(x)]
  }

  result
}

less <- function(x, y) UseMethod("less")
less.numeric <- function(x, y) x < y
less.tuple <- function(x, y) x[1] < y[1] || x[2] < y[2]

sort_list(x)


## [[1]]
```

```
## [1] 1 1
## attr(,"class")
## [1] "tuple"
##
## [[2]]
## [1] 1 2
## attr(,"class")
## [1] "tuple"
##
## [[3]]
## [1] 2 0
## attr(,"class")
## [1] "tuple"
```

We would need to define concrete implementations of `less` for all types we wish to short, though. Alternatively, we can tell R how to handle `` `<` `` for our own types, and we will see how to in a later chapter. With that approach, we will get sorting functionality for all objects that can be compared this way. A third possibility is to make `less` a parameter of the sorting function:

```
merge_lists <- function(x, y, less) {
  # Same function body as before
}


sort_list <- function(x, less = `<`) {

  if (length(x) <= 1) return(x)

  result <- vector("list", length = length(x))

  start <- 1
  end <- length(x)
  middle <- end %/% 2

  merge_lists(sort_list(x[start:middle], less),
              sort_list(x[(middle+1):end], less),
              less)
}
```

```
unlist(sort_list(as.list(sample(1:5))))
```

```
## [1] 1 2 3 4 5
```

```
tuple_less <- function(x, y) x[1] < y[1] || x[2] < y[2]
sort_list(x, tuple_less)
```

```
## [[1]]
## [1] 1 1
## attr(,"class")
## [1] "tuple"
##
## [[2]]
## [1] 1 2
## attr(,"class")
## [1] "tuple"
##
## [[3]]
## [1] 2 0
## attr(,"class")
## [1] "tuple"
```

We make the default `less` function `` `<` `` but can provide another for types where this comparison function doesn't work.

General comments on flexible implementations of algorithms

As a general rule, you want to make your algorithm implementations adaptable by providing handles for polymorphism. Either by providing options for certain functions, like we did with `less` above or by using generic functions for abstract data types.

You might be able to experiment with optimal data structures and implementation of operations when you implement an algorithm for a given use, but by providing handles for modifying your function you make the code more reusable. Even in cases where the algorithm will perform correctly for different applications, you might still want to provide flexibility; the performance of

algorithms often depend on the usage. In an asymptotic analysis we generally prefer implementations that have theoretical better running times, but in practise, we want the fastest code, and that is not necessarily the asymptotically fastest algorithms. We hide away constants when we use "big-O" analysis, but those constants matter, so you want users of your implementations to be able to replace data structures and operations used in your algorithm implementations.

Figuring out how to best provide this flexibility in your implementations often require some experimentation. For abstract data structures, generic functions are usually the best approach. For something like comparison in the sorting example above, all three solutions (generic functions, operator overloading, or providing a function with a good default) are probably equally good. But just like experimentation and some thinking is involved in designing good software interfaces, the same is needed in algorithmic programming.

More on `UseMethod`

The `UseMethod` function is what we use to define a generic function, and it takes care of finding the appropriate concrete implementation using the name lookup we saw earlier. There are some details about `UseMethod` I left out before, though.

First of all, it doesn't actually work as a function normally does. It looks like a function, and to a large degree it is a function, but if you treat it just as any other function you might get effects you didn't expect.

First of all, you can pass local variables along to concrete implementations if you assign them before you call `UseMethod`. Let's consider a simple case:

```
foo <- function(object) UseMethod("foo")
foo.numeric <- function(object) object
foo(4)
```

```
## [1] 4
```

```
bar <- function(object) {
  x <- 2
  UseMethod("bar")
}
```

```
bar.numeric <- function(object) x + object
bar(4)
```

```
## [1] 6
```

Here the `foo` function uses the pattern we saw earlier. It just calls `UseMethod`. We then define a concrete function to be called if `foo` is invoked on a number. Numbers have classes, and that class is `numeric`. (Technically, there is more to numbers than this class, but for now, we don't need to worry about that). Nothing strange is going on with `foo`.

With `bar`, however, we assign a local variable before we invoke `UseMethod`. This variable, `x`, is visible when `bar.numeric` is called. With a normal function call, you have to take steps to get access to the calling scope, so here `UseMethod` does not behave like a normal function.

In the call to `UseMethod`, it doesn't behave like a normal function either. You cannot use `UseMethod` as part of an expression.

```
baz <- function(object) UseMethod("baz") + 2
baz.numeric <- function(object) object
baz(4)
```

```
## [1] 4
```

When `UseMethod` is invoked, the concrete function takes over completely, and the call to `UseMethod` never returns. In this way, it is similar to the `return` function. Any expression you put `UseMethod` in is not evaluated because of this, and any code you might put after the `UseMethod` call is never evaluated.

The `UseMethod` function takes a second argument, besides the name of the generic function. This is the object that is used to dispatch the generic function on—the object whose type determines the concrete function that will be called—and this argument can be used if you do not want to dispatch based on the first argument of the function that calls `UseMethod`. Since dispatching on the type of the first function argument is such a common pattern, using another object in the call to `UseMethod` can cause confusion, and I recommend that you do not do this unless you have very good reasons for it.

Operator overloading

Overloading operators, that is, giving operators such as `+` or `-` new or additional functionality, is not inherently object-oriented, but since it can be thought of as adding polymorphism to functions it fits in naturally here after we have gone through polymorphism through generic functions.

Opinions vary on whether overloading operators is good or bad practise. Some languages allow it, others do not; some languages allow you to make your own infix operators but not change existing once, and some languages are just inconsistent in allowing some operator overloading for built-in objects but not for user-defined. One argument against overloading is the expected behaviour of operators. It is so ingrained in us to expect `+` to mean addition that we cannot handle if it also means string concatenation. This argument, of course, ignores that we have no problem with using `+` for both integer and floating point addition. Another argument, and in my opinion a more valid one, is that it is harder to remember what an operator does than what a function does since the function name at least gives us some hint as to the function's purpose. The truth is that infix operators, when carefully used, can provide us with a more convenient syntax than simple function calls. You probably find `2 * x + 5` easier to read than `plus(times(2, x), 5)`, and most programming languages, with Lisp dialects a noticeable except, prefer the former to the latter. The same goes for user-defined or user-overloaded operators. Using `magrittr`'s pipe operator, `%>%`, makes analysis workflows much easier to read, and `ggplot2`'s overloading of `+` makes plotting code much easier to read.

Since, in R, you can both overload existing operators and create your own, you have to choose which is most appropriate for any given situation. My rule of thumb is to prefer new infix operators unless it feels natural to use an existing one. That is, of course, a terribly subjective evaluating, but some operations just seem like "addition"—I would say combining operations in `ggplot2` can

be justified as such—while others don't—like pipeline operations in `magritte`. It is a judgment call, and you can always experiment with your code to see what you feel is most natural.

In this chapter, we will see how we can overload operators. I will not cover how you create infix operators. If you are interested, I describe those it in my *Functional Programming in R* book.

Functions and operators

Every operation in R involves a function call. Control structures, subscripting, even parenthesis involve functions, and, naturally, operators involve function calls as well. This means that operators can be overridden. You can replace one implementation of `+` with another just by defining a new version of `+`. But you really shouldn't. Replacing the built-in operators with user-defined will affect all your code, slow it down, and is very likely to introduce hard-to-find bugs. You don't have to do this to define operators for your own classes, though. The operators are generic, and you can define specialised versions of them, implementing how any operator should handle new classes you define.

To illustrate how this is done, we define a class for arithmetic modulus some number `n`. Here I assume that `value` is some numeric type—in production code, we would write tests for this, but in the example, we just implicitly assume this—and we use an attribute to store the number `n`. We then compute the value modulus `n`, set the class, and return the result.

```
modulus <- function(value, n) {
  result <- value %% n
  attr(result, "modulus") <- n
  class(result) <- c("modulus", class(value))
  result
}
```

To pretty-print values of the class, we define the `print` method. We want to print values, `x`, using the underlying type but with a line above giving us `n`. If we just call `print(x)` we would recurse back to `print.modulus` since that is the type of `x`, but we can use the function `unclass` to get rid of the type of `x`. It doesn't completely get rid of the type, though. If `x` is a numeric type, `unclass` just reduce `x` to that. When we print primitive values with attributes

the attributes are printed as well, so we get rid of all attributes before we print the reduced `x`. Here we could directly call `print(x)` but `NextMethod()` works as well, so that is what I have used here.

```
print.modulus <- function(x, ...) {
  cat("Modulus", attr(x, "modulus"), "values:\n")
  # remove attributes to get plain numeric printing
  x <- unclass(x)
  attributes(x) <- NULL
  NextMethod()
}
```

```
(x <- modulus(1:6, 3))
```

```
## Modulus 3 values:
## [1] 1 2 0 1 2 0
```

Defining single operators

We can define what addition means for this type by defining the function `` `+.modulus` ``:

```
`+.modulus` <- function(x, y) {
  n <- attr(x, "modulus")
  x <- unclass(x)
  y <- unclass(y)
  modulus(x + y, n)
}
```

We first get `n` from `x`. Then we remove the class information from the two operands so we can call the primitive `+` to calculate their sum. If we didn't do this, we would again recurse when we tried to add `x` to `y`, and we then simply calculate the result and use the `modulus` constructor to return it with the right type.

Dispatching of such operator-functions works a little different from the generic functions we have seen earlier. There, the dispatch is based on the type of the first argument, at least unless you explicitly state otherwise, but for operators,

the dispatch is based on the type of both operands. If both have a class that implements an operator, they must have the same class. If only one of them have a class, and here primitive classes such as `"numeric"` or `"integer"` do not count, then the dispatch is based on that. So if `x` is of type `"modulus"` then both `x + 1:6` and `1:6 + x` will call `` `+.modulus` ``.

```
x + 1:6
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

```
1:6 + x
```

```
## Modulus values:
## numeric(0)
```

The first expression does what we expect, but the second does not. This is not because the dispatch is not working but because we got `n` from the first operand only. Since we are only guaranteed that *one* of the two operands have type `"modulus"` we need to check both. We can do this simply by checking if the attribute `"modulus"` is `NULL` or not:

```
`+.modulus` <- function(x, y) {
  n <- ifelse(!is.null(attr(x, "modulus")),
              attr(x, "modulus"), attr(y, "modulus"))
  x <- unclass(x)
  y <- unclass(y)
  modulus(x + y, n)
}
```

```
x + 1:6
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

```
1:6 + x
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

Depending on which semantic we want addition of these types to have, we might not want to allow addition of types with different `n`. If we add such types now, the `n` is taken from the first operand.

```
y <- modulus(1:6, 2)
x + y
```

```
## Modulus 3 values:
## [1] 2 2 1 1 0 0
```

With a little more check we ensure that the two operands are compatible:

```
`+.modulus` <- function(x, y) {
  nx <- attr(x, "modulus")
  ny <- attr(y, "modulus")
  if (!is.null(nx) && !is.null(ny) && nx != ny)
    stop("Incompatible types")
  n <- ifelse(!is.null(nx), nx, ny)

  x <- unclass(x)
  y <- unclass(y)
  modulus(x + y, n)
}

x + y
```

```
## Error in `+.modulus`(x, y): Incompatible types
```

```
y <- modulus(rev(1:6), 3)
x + y
```

```
## Modulus 3 values:
## [1] 1 1 1 1 1 1
```

Group operators

Using generic functions, we can define all relevant operators for a user-defined type, but it is also possible to handle all operators in a single function, `Ops`. This function is called a "group generic method" because it handles a group of generic functions; other group methods are `Math`, `Complex`, and `Summary`, which we will not cover here.

If we define the function `Ops.modulus`, it will be called for all operators of `modulus` objects where the operator function is not defined. That is, if we have defined `` `+.modulus` `` as above, that function will be preferred over `Ops.modulus`, but otherwise, if one or both of the operands are of type `modulus`, then `Ops.modulus` will be called.

We can define it like this:

```
Ops.modulus <- function(e1, e2) {
  nx <- attr(e1, "modulus")
  ny <- attr(e2, "modulus")
  if (!is.null(nx) && !is.null(ny) && nx != ny)
    stop("Incompatible types")
  n <- ifelse(!is.null(nx), nx, ny)

  result <- unclass(NextMethod()) %% n
  modulus(result, n)
}
```

The testing of input values at the beginning of the function is the same as for `` `+.modulus` ``. After the testing, we use `NextMethod` to call the operation using the underlying type. This strips the `modulus` class from the operands and evaluates whatever operation we are currently handling. We unclass the result, necessary because the result will inherit the attributes of the operands of `Ops` so if we don't the result will have class `modulus`, and we then compute modulus `n`. If we didn't `unclass`, this would be a recursive call, but since we do remove the class we just do modulus in the underlying type. We finally create a `modulus` object of the result.

With this function defined we get all binary operators in one go.

```
y <- modulus(rev(1:6), 3)
x - y
```

```
## Modulus 3 values:
## [1] 1 0 2 1 0 2
```

```
x * y
```

```
## Modulus 3 values:
## [1] 0 1 0 0 1 0
```

This includes comparison operators as well:

```
x == y
```

```
## Modulus 3 values:
## [1] 0 1 0 0 1 0
```

```
x == x
```

```
## Modulus 3 values:
## [1] 1 1 1 1 1 1
```

```
x != y
```

```
## Modulus 3 values:
## [1] 1 0 1 1 0 1
```

```
x != x
```

```
## Modulus 3 values:
## [1] 0 0 0 0 0 0
```

It even includes unary operators. If we use a unary minus, however, the argument `e2` will be missing in the function call, which we do not handle correctly right now.

```
- x
```

```
## Error in Ops.modulus(x): argument "e2" is missing, with no default
```

We can easily fix this, however, by checking if `e2` is missing. Otherwise, the function will work as it is.

```
Ops.modulus <- function(e1, e2) {
  nx <- attr(e1, "modulus")
  ny <- if (!missing(e2)) attr(e2, "modulus") else NULL
  if (!is.null(nx) && !is.null(ny) && nx != ny)
    stop("Incompatible types")
  n <- ifelse(!is.null(nx), nx, ny)

  result <- unclass(NextMethod()) %% n
  modulus(result, n)
}

- x
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

Units example

For a slightly more involved example, we define a class for associating physical units with values. This will allow us to check that units we manipulate are compatible—so we do not subtract meters from seconds and such—and will do unit analysis as part of arithmetic operations. The example is a simplified version of the package units[1]. The `units` package also hands unit conversion and unit simplification. Here we just implement a simple arithmetic of symbolic units and simple equality check of them.

The idea is to have a representation of physical units and then associate these to numeric values. Physical units, here, refers to units like square kilometres, metres per second, etc. In general, these will be symbolic expressions, but we will only consider the slightly simpler situation where the units are a fraction of physical constants. In that case, we can represent these as a list of terms in

[1] https://github.com/edzer/units

the nominator and a list of terms in the denominator. If we always keep these lists sorted, we have a canonical representation of them, and we can check equality of two units by checking equality of the nominator and denominator lists. We can implement the constructor like this:

```
symbolic_unit <- function(nominator, denominator = "") {
  non_empty <- function(x) x != ""
  nominator <- sort(Filter(non_empty, nominator))
  denominator <- sort(Filter(non_empty, denominator))
  structure(list(nominator = nominator, denominator = denominator),
            class = "symbolic_unit")
}
```

We can translate these units into a string representation of the fraction for pretty-printing.

```
as.character.symbolic_unit <- function(x, ...) {
  format_terms <- function(terms, op) {
    if (length(terms) == 0) return("1")
    paste0(terms, collapse = op)
  }
  nominator <- format_terms(x$nominator, "*")
  denominator <- format_terms(x$denominator, "/")
  paste(nominator, "/", denominator)
}

print.symbolic_unit <- function(x, ...) {
  cat(as.character(x, ...), "\n")
}

(x <- symbolic_unit("m"))

## m / 1

(y <- symbolic_unit("m", "s"))

## m / s
```

Comparing two symbolic units involves checking that nominator and denominator are equal.

```
`==.symbolic_unit` <- function(x, y) {
  if (!(inherits(x, "symbolic_unit") && inherits(y, "symbolic_unit")))
      stop("Comparison only defined when both x and y are both symbolic_units")
  return(identical(x$nominator, y$nominator) &&
           identical(x$denominator, y$denominator))
}

`!=.symbolic_unit` <- function(x, y) !(x == y)

x == y

## [1] FALSE

x != y

## [1] TRUE
```

Adding and subtracting physical quantities is only possible if they have the same units, but it is always possible to multiply and divide units. The resulting unit is then obtained by doing the same operation on the (symbolic) units as you do on the quantities. To be able to handle this, we define multiplication and division on symbolic units.

```
`*.symbolic_unit` <- function(x, y) {
  symbolic_unit(c(x$nominator, y$nominator),
                c(x$denominator, y$denominator))
}

`/.symbolic_unit` <- function(x, y) {
  symbolic_unit(c(x$nominator, y$denominator),
                c(x$denominator, y$nominator))
}

x * y
```

```
## m*m / s
```

```
x / y
```

```
## m*s / m
```

We now have everything in place to represent units. We just need to define the class for associating units with quantities. This class is very similar to the `modulus` class we wrote earlier. We take a (numeric) value, associate a symbolic unit in an attribute, and set the class.

```
units <- function(value, nominator, denominator = "") {
  attr(value, "units") <- symbolic_unit(nominator, denominator)
  class(value) <- c("units", class(value))
  value
}
```

Pretty-printing follows the pattern we saw with `modulus`. We need to strip the class and attributes to use the underlying print method, called through `NextMethod`, but that is all there is to it.

```
print.units <- function(x, ...) {
  cat("Units: ", as.character(attr(x, "units")), "\n")
  # remove attributes to get plain numeric printing
  x <- unclass(x)
  attributes(x) <- NULL
  NextMethod()
}
```

```
(x <- units(1:6, "m"))
```

```
## Units:  m / 1
## [1] 1 2 3 4 5 6
```

Handling operators for `units` is only slightly more involved than it was for `modulus`. We need to distinguish between operators where we require that the units match and those where we need to construct new units. The former of

those are addition, subtraction, and comparisons, assuming we only want to consider numbers equal if they agree in both quantity and associated units, the latter are multiplication and division where the resulting units must be computed from the operands. It is not obvious how to handle logical operators on physical quantities if that is something that even makes sense, so for operators that do not fall into these two categories, we should just default to what the underlying type does.

We implement the operators using the `Ops` group function. Inside this function, we can get hold of the actual operator being evaluated using the variable `.Generic`. This is not a parameter of the function, but it will be set to the operator being evaluated when the function is called, and we can check the operator and handle it appropriately by switching on it.

```
Ops.units <- function(e1, e2) {
  su1 <- attr(e1, "units")
  su2 <- if (!missing(e2)) attr(e2, "units") else NULL

  if (.Generic %in% c("+", "-", "==", "!=", "<", "<=", ">=", ">")) {
    if (!is.null(su1) && !is.null(su2) && su1 != su2)
      stop("Incompatible units")
    su <- ifelse(!is.null(su1), su1, su2)
    return(NextMethod())
  }

  if (.Generic == "*" || .Generic == "/") {
    if (is.null(su1))
      su1 <- symbolic_unit("")
    if (is.null(su2))
      su2 <- symbolic_unit("")
    su <- switch(.Generic, "*" = su1 * su2, "/" = su1 / su2)
    result <- NextMethod()
    attr(result, "units") <- su
    return(result)
  }

  # For the remaining operators we don't really have a good
  # way of treating the units so we strip that info and go back
  # to numeric values
  e1 <- unclass(e1)
```

```
  e2 <- unclass(e2)
  attributes(e1) <- attributes(e2) <- NULL
  NextMethod()
}
```

With this definition of the units operators we can combine units with scalars:

```
2 * x
```

```
## Units:  m / 1
## [1]  2  4  6  8 10 12
```

```
x + 2
```

```
## Units:  m / 1
## [1] 3 4 5 6 7 8
```

```
x - 2
```

```
## Units:  m / 1
## [1] -1  0  1  2  3  4
```

If we attempt to add two quantities with incompatible types we will be warned that this is incorrect

```
(y <- units(1:6, "m", "s"))
```

```
## Units:  m / s
## [1] 1 2 3 4 5 6
```

```
x + y
```

```
## Error in Ops.units(x, y): Incompatible units
```

but when the units are compatible we can add and subtract

```
(z <- units(1:6, "m"))
```

```
## Units:  m / 1
## [1] 1 2 3 4 5 6
```

```
x + z
```

```
## Units:  m / 1
## [1]  2  4  6  8 10 12
```

```
x - z
```

```
## Units:  m / 1
## [1] 0 0 0 0 0 0
```

Multiplication and division is always permissible and the resulting units are derived from the operands.

```
2 * x
```

```
## Units:  m / 1
## [1]  2  4  6  8 10 12
```

```
x * y
```

```
## Units:  m*m / s
## [1]  1  4  9 16 25 36
```

```
x / y
```

```
## Units:  m*s / m
## [1] 1 1 1 1 1 1
```

implementation can look like this:

```
library(R6)

VectorStack <- R6Class("VectorStack",
                       private = list(elements = NULL),
                       public = list(
                         top = function() {
                           private$elements[1]
                         },
                         pop = function() {
                           private$elements <- private$elements[-1]
                           invisible(self)
                         },
                         push = function(e) {
                           private$elements <- c(e, private$elements)
                           invisible(self)
                         },
                         is_empty = function() {
                           length(private$elements) == 0
                         }
                       ))
```

Here, besides the class name, we use two arguments to `R6Class`: `private` and `public`. These are used to define attributes of the class, either values stored in objects or methods we can call on the objects. For both, the arguments are lists. The names used for the list elements become the names of the attributes and the values, naturally, the attribute values.

The difference between the two arguments is that attributes in the `public` list can be accessed anywhere while attributes in the `private` list can only be accessed in methods you define for the class. In methods, you can access elements in the `public` list using the variable `self`, and you can access attributes in the `private` list using the variable `private`. In this `VectorStack` implementation we have made the vector used for storing the stack private, and we have implemented the stack interface as public methods. Inside the methods we access the elements as `private$elements`, and in `push` and `pop` we return the object itself using the variable `self`. We return this object wrapped in `invisible`, so it isn't automatically printed when we call these methods, but we didn't have to. We didn't have to return an object at all for

these methods, but doing so allows us to chain together method calls, as we will see below, so it is good practice.

Notice that the `self` and `private` objects are not arguments to the methods. They just exist in the namespace of the functions as part of the magic R6 uses to implement its mutable object semantics.

The `VectorStack` object we create this way is not a constructor function itself, as it was for S4. It is a so-called object generator. To create an object we use the attribute `new` of this generator thus:

```
(stack <- VectorStack$new())
```

```
## <VectorStack>
##   Public:
##     clone: function (deep = FALSE)
##     is_empty: function ()
##     pop: function ()
##     push: function (e)
##     top: function ()
##   Private:
##     elements: NULL
```

Printing this object is rather verbose, but if we define the method `print` for the class we can modify how it is displayed.

```
VectorStack <- R6Class("VectorStack",
                       private = list(elements = NULL),
                       public = list(
                         # ... rest of the methods
                         print = function() {
                           cat("Stack elements:\n")
                           print(private$elements)
                         }
                       ))
```

This doesn't modify the existing object

```
stack
```

```
## <VectorStack>
##   Public:
##     clone: function (deep = FALSE)
##     is_empty: function ()
##     pop: function ()
##     push: function (e)
##     top: function ()
##   Private:
##     elements: NULL
```

but it has an effect if we create a new one

```
(stack <- VectorStack$new())
```

```
## Stack elements:
## NULL
```

We can access the (public) attributes of the stack object, and call methods if the attributes are functions, using `$` indexing:

```
stack$push(1)$push(2)$push(3)
stack
```

```
## Stack elements:
## [1] 3 2 1
```

```
while (!stack$is_empty()) stack$pop()
stack
```

```
## Stack elements:
## numeric(0)
```

The chained call to `push` here is possible because the `push` method returns the object itself. Unlike the previous implementations where `push` returns a new object, for the R6 object, the existing object is modified. We do not need to assign the result of the three `push` calls back to `stack`, and we do not need to assign the calls to `pop` back to `stack` either. Returning the object itself in these functions allows us to chain method calls, but that is all this does. The R6 object is not immutable.

Object initialisation

If we want to set attributes of objects when constructing them, we need to do a little more work than in S4. We cannot just use named arguments in the constructor; this call would give us an error:

```
stack <- VectorStack$new(elements = 1:4)
```

To be able to initialise objects this way, we need to explicitly write a function for it. This function must be a `public` method called `initialise`. If you want the constructor to take arguments, you must specify the arguments in this function. You cannot make this function `private`; it is an error to put a function named `initialise` in the `private` list.

To initialise `VectorStack` objects with a sequence of elements we can implement its `initialize` function like this:

```
VectorStack <- R6Class("VectorStack",
                       private = list(elements = NULL),
                       public = list(
                         initialize = function(elements = NULL) {
                           private$elements <- elements
                         },
                         # ... rest of the methods
                       ))
```

With this initialisation function we can now construct objects with `elements` initialised.

```
(stack <- VectorStack$new(elements = 1:4))
```

```
## Stack elements:
## [1] 1 2 3 4
```

Private and public attributes

The elements in the stack are private, so we cannot access them the same way we can the public methods. You might hope that `stack$elements` would then give you an error, but unfortunately not.

```
stack$elements
```

```
## NULL
```

This is because accessing a list with a key it doesn't contain gives you `NULL` and it is this behaviour you are getting.

```
list()$elements
```

```
## NULL
```

We can see the difference between private and public attributes with this little example:

```
A <- R6Class("A", public = list(x = 5), private = list(y = 13))
```

With this definition of class `A`, we should be able to access object's `x` attributes, and we can:

```
a <- A$new()
a$x
```

```
## [1] 5
```

```
a$x <- 7
a$x
```

```
## [1] 7
```

We can also get `y`, but it has the behaviour we saw above for stacks, and we are not allowed to modify it since it isn't really an attribute of the object.

```
a$y
```

```
## NULL
```

```
a$y <- 12
```

```
## Error in a$y <- 12: cannot add bindings to a locked environment
```

```
a$y
```

```
## NULL
```

In general, you cannot create new attributes to R6 objects just by assigning to `$` indexed values, as you can in S3. Attributes must be defined in the class definition.

```
a$z <- "foo"
```

```
## Error in a$z <- "foo": cannot add bindings to a locked environment
```

You can modify public data attributes, as we saw above for `x`, but don't try to be clever and modify methods. It is really bad practise to change methods for a single object, to begin with, but luckily it is also "verboten" in R6, and you will get you an error.

```
stack$pop <- NULL
```

```
## Error in stack$pop <- NULL: cannot change value of locked binding for 'pop'
```

In general, it is considered good practice to keep data private and methods that are part of a class interface public. There are several reasons for this: If data is only modified through a class' methods then you have more control over the state of objects and can ensure that an object is always in a valid state before and after all method calls, but, perhaps more importantly, keeping the representation of objects hidden away limits the dependency between a class and code that uses the class. If any code can access the inner workings of objects, there is a good chance that eventually a lot of code will. This means that you will have to modify all the uses of a class if you change how objects of the class are represented. If on the other hand, the code only accesses objects through a public interface, then you can modify all the private attributes as much as you want as long as you keep the public interface unchanged. You

will, of course, have to modify some of the class' methods, but changes will be limited to that.

In the R6 system, private attributes can be accessed only by methods you define for the class or in methods defined in sub-classes. If you are used to languages such as C++ or Java, this might surprise you, but the private attributes in R6 are similar to the protected attributes in those languages and not the private attributes.

Active bindings

There is a way of getting the syntax of accessing data attributes without actually doing so. If you have code that already uses a public attribute and you want to change that into a function to hide or modify implementation details, you can use this, or if you just like the syntax for data better than method calls.

This is achieved through the `active` argument to `R6Class`. Here you can provide a list of attributes, as for `private` and `public`, but these attributes should be functions, and they will define a value-like syntax for calling the functions.

As an example, we can take the elements in the vector stack. We want to be able to write `stack$elements`, but we do not want to make the elements public. So we write a function for `elements` and add it to `active`. We cannot have the same name used both in `private` and `active` (or in `public` for that matter), so we have to change the name for the private data attribute first, and of course update all the existing methods. After doing that, we can add the `active` function like this:

```
VectorStack <- R6Class("VectorStack",
                       private = list(elements_ = NULL),
                       public = list(
                         # ... methods
                       ),
                       active = list(
                         elements = function(value) {
                           if (!missing(value))
                             stop("elements are read-only")
                           private$elements_
```

```
    }
  ))
```

Functions in the `active` list should take one argument, `value`. This value will be missing when we read the attribute and contain data when we assign to the attribute. In this implementation we consider assigning to the elements an error and we return the private `elements_` when we read the attribute.

This will give us the elements:

```
stack <- VectorStack$new(elements = 1:4)
stack$elements
```

```
## [1] 1 2 3 4
```

while this will raise an error:

```
stack$elements <- rev(1:3)
```

```
## Error in (function (value) : elements are read-only
```

You can use these `active` functions to modify values you assign, to ensure object consistency, or to fake an attribute that isn't directly stored but exists implicitly by being computable from other data. It all depends on how you choose to use them.

Inheritance

The way we specify class hierarchies, and the way method-calls are dispatched to the most specialised implementation of a method, is relatively straightforward. We can take the example of three classes we have seen two times earlier and implement it in R6. To specify that one class inherits from another we use the `inherit` argument to `R6Class` and to write a more specialised version of a method we simply add the method to the `public` or `private` lists. Overall, writing methods and class hierarchies is done with much simpler code in R6 than in both S3 and S4.

```
A <- R6Class("A",
             public = list(
               f = function() print("A::f"),
               g = function() print("A::g"),
               h = function() print("A::h")
             ))
B <- R6Class("B", inherit = A,
             public = list(
               g = function() print("B::g"),
               h = function() print("B::h")
             ))
C <- R6Class("C", inherit = B,
             public = list(
               h = function() print("C::h")
             ))
```

There are no surprises in how we instantiate objects of the classes; we have to use the `new` method in the object generators:

```
x <- A$new()
y <- B$new()
z <- C$new()
```

For method `f` we only have an implementation for class `A`, so calling `f` on all three objects will call that version. Except that the method call has a different syntax from the implementations for S3 and S4, there are no surprises here.

```
x$f()
```

```
## [1] "A::f"
```

```
y$f()
```

```
## [1] "A::f"
```

```
z$f()
```

```
## [1] "A::f"
```

For `g`, we have implementations in both `A` and `B`, and the `C` object will call the `B` implementation since this is the most specialised for that class.

```
x$g()
```

```
## [1] "A::g"
```

```
y$g()
```

```
## [1] "B::g"
```

```
z$g()
```

```
## [1] "B::g"
```

Finally, for `h` we have implementations in all three classes, so we call different methods for the three objects.

```
x$h()
```

```
## [1] "A::h"
```

```
y$h()
```

```
## [1] "B::h"
```

```
z$h()
```

```
## [1] "C::h"
```

There should not be any surprises in how inheritance and method dispatching works in R6.

References to objects and object sharing

One important thing is different from R6 objects and all other R objects: the R6 objects have a state that can be modified. If you are used to other object-oriented programming languages, this might not sound like much of a deal, but in general, we can assume that calling functions do not have side-effects in R, except for changing values that variables point to. When objects can suddenly change state, we need to worry about when two references are to the same object or merely references to two objects that represent the same values.

The first thing you need to know is that values set in the definition of `private` and `public` lists are shared between objects of a class. To see this in action we can define these two classes:

```
A <- R6Class("A", public = list(x = 1:5))
B <- R6Class("B",
             public = list(
               x = 1:5,
               a = A$new()
             ))
```

Here, I am breaking the rule about not having public data to simplify the example. In any case, what we have is one class, `A`, that contains a vector and another, `B`, that contains another vector and a reference to an `A` object. Let's create two objects of class `B`.

```
x <- B$new()
y <- B$new()
```

We can first check the behaviour of the vector in the objects. It is initialled to the first five natural numbers, so that is what both objects contain initially.

```
x$x
```

```
## [1] 1 2 3 4 5
```

```
y$x
```

```
## [1] 1 2 3 4 5
```

If we then modify the vector in `x` we see that this vector changes but the vector in `y` does not. This is how vectors behave in R and generally what we would expect.

```
x$x <- 1:3
x$x
```

```
## [1] 1 2 3
```

```
y$x
```

```
## [1] 1 2 3 4 5
```

If we modify the vector in the nested `A` object, however, we get a different behaviour. Here, changing the value through `x` *also* changes the value in `y`.

```
x$a$x
```

```
## [1] 1 2 3 4 5
```

```
y$a$x
```

```
## [1] 1 2 3 4 5
```

```
x$a$x <- 1:3
x$a$x
```

```
## [1] 1 2 3
```

```
y$a$x
```

```
## [1] 1 2 3
```

Even creating a new object from the class will give us an object that contains the modified value.

```
z <- B$new()
z$a$x
```

```
## [1] 1 2 3
```

All three objects are referring to the same `A` object and modifications to this object are reflected in all of them. This is generally how R6 classes behave. The copy-on-modification semantics of other R objects is not how R6 objects behave. When you have two references to the same object then modifying one of them will also modify the other.

Modifying `x$x` didn't change `y$x` because `x` and `y` are different objects, but if we make another reference to the object pointed to by `x` then changes to `x` will be reflected in the other.

```
w <- x
w$x
```

```
## [1] 1 2 3
```

```
x$x <- 1:5
w$x
```

```
## [1] 1 2 3 4 5
```

If you want each object of a class to contain distinct objects of an R6 class, then you can create the objects in the `initialize` function instead of in the `public` or `private` lists. This function is called whenever you create a new object and objects that are created in the initialisation function will be distinct and thus not shared.

We can modify `B` like this:

```
B <- R6Class("B",
             public = list(
               x = 1:5,
               a = NULL,
               initialize = function() {
                 self$a <- A$new()
               }))
```

We need to re-create `x` and `y` to have them refer to this new class

```
x <- B$new()
y <- B$new()
```

but now we can modify one without modifying the other.

```
x$a$x
```

```
## [1] 1 2 3 4 5
```

```
x$a$x <- 1:3
x$a$x
```

```
## [1] 1 2 3
```

```
y$a$x
```

```
## [1] 1 2 3 4 5
```

Since assigning from one variable to another just create another reference to the same object, we need another way of creating an effective copy. This is done with the `clone` method that all `R6` objects automatically implement.

If we clone object `x` we get a new copy of the object, which contains the same state as `x` does at the time of cloning, but which can be modified without changing `x`.

```
z <- x$clone()
z$x
```

```
## [1] 1 2 3 4 5
```

```
z$x <- 1:2
x$x
```

```
## [1] 1 2 3 4 5
```

The default cloning is shallow, however. It makes a copy of the object, but if the object contains a reference to an R6 class, then the clone will contain a reference to the same object. If we modify the `a` attribute of `z`, we will also modify the `a` attribute of `x`.

```
x$a$x
```

```
## [1] 1 2 3
```

```
z$a$x <- 1:5
x$a$x
```

```
## [1] 1 2 3 4 5
```

If we call `clone` with the option `deep = TRUE` we will instead get a deep copy; here we get a transitive closure of cloned references, so here we can modify the `a` attribute safe in the knowledge that they are distinct between an object and its clone.

```
y <- x$clone(deep = TRUE)
```

```
x$a$x
```

```
## [1] 1 2 3 4 5
```

```
y$a$x <- NULL
x$a$x
```

```
## [1] 1 2 3 4 5
```

Interaction with S3 and operator overloading

We don't have a mechanism for defining new operators for R6 objects, but we can use the S3 system for this. Objects create from R6 object generators are assigned a `class` attribute, a list of the name we give the class when creating the generator and `"R6"`, so we can define generic function specialisations for them.

We can implement the `modulus` class in R6 like this:

```
modulus <- R6Class("modulus",
                   private = list(
                     value_ = c(),
                     n_ = c()
                   ),
                   public = list(
                     initialize = function(value, n) {
                       private$value_ <- value
                       private$n_ <- n
                     },
                     print = function() {
                       cat("Modulus", private$n_, "values:\n")
                       print(private$value_)
                     }
                   ),
                   active = list(
                     value = function(value) {
                       if (missing(value)) private$value_
                       else private$value_ <- value %% private$n_
                     },
                     n = function(value) {
                       if (!missing(value)) stop("Cannot change n")
                       private$n_
                     }
                   ))

(x <- modulus$new(value = 1:6, n = 3))

## Modulus 3 values:
## [1] 1 2 3 4 5 6
```

There are a few things going on in this class definition. We define the attributes for holding the data in the `private` list, define an initialisation function and a print function, and then we define two `action` attributes for accessing the data. We allow users of the class to modify `values` but not `n` (for no good reason other than it gives us an example of two different behaviour), and for the `values` attribute we make sure that we modify the data before we store it in the private `values_` variable.

The `class` attribute of objects of this class contains

```
class(x)
```

```
## [1] "modulus" "R6"
```

This means that we can define arithmetic operations on the class using the S3 system like this:

```
Ops.modulus <- function(e1, e2) {
  nx <- ny <- NULL
  if (inherits(e1, "modulus")) nx <- e1$n
  if (inherits(e2, "modulus")) ny <- e2$n
  if (!is.null(nx) && !is.null(ny) && nx != ny)
    stop("Incompatible types")
  n <- ifelse(!is.null(nx), nx, ny)

  v1 <- e1
  v2 <- e2
  if (inherits(e1, "modulus")) v1 <- e1$value
  if (inherits(e2, "modulus")) v2 <- e2$value

  e1 <- v1 ; e2 <- v2
  result <- NextMethod() %% n
  modulus$new(result, n)
}
```

The implementation is slightly different from the S3 version, because the data in the objects are represented differently, but the general control flow is the same, and with this definition we have modulus arithmetic.

```
x + 1:6
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

```
1:6 + x
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

```
2 * x
```

```
## Modulus 3 values:
## [1] 2 1 0 2 1 0
```

If you make subclasses of an R6 class like `modulus` you will get a `class` attribute that also reflects this, so the S3 dispatch mechanism will also work for sub-classes in the R6 system.

```
modulus2 <- R6Class("modulus2", inherit = modulus)
y <- modulus2$new(value = 1:2, n = 3)
class(y)
```

```
## [1] "modulus2" "modulus"  "R6"
```

```
x + y
```

```
## Modulus 3 values:
## [1] 2 1 1 0 0 2
```

That being said, don't go crazy with combing R6 and S3 either; it will only confuse the maintainers of your code (which are likely to include yourself somewhere in the future).

Conclusions

This concludes this book on object-oriented programming in R. You now know the three different systems for object-oriented programming in R and how to use them to define class hierarchies and polymorphic functions.

Object-oriented programming in R, at least for the S3 and S4 system, differs from most other object-oriented programming languages. Most languages consider objects mutable, and most object-oriented software designs involve wiring up objects with references to each other such that their behaviour depends on the changing states of other objects. The R6 system is closer to this type of language design. Still, the S3 and S4 systems combine two powerful programming language paradigms: functional programming and object-oriented programming. The combination of dynamic function dispatch based on the argument types and high-level functional programming lets you construct flexible and extensible software.

It can be confusing with three very different systems for object-oriented programming in the same language, and I would recommend that you stick to one for any single project. Knowing all three, however, and knowing the pros and cons of using them lets you pick the right tool for any particular job. The S3 system is the simplest of the three and useful for getting a small model up and running in short time. The more formal classes of S4 makes it easier to structure more complex frameworks, and the reference semantics of R6 makes it simpler to implement traditional mutable data structures than you can otherwise easily do in R.

Getting familiar with these systems, of course, requires practise and you will not be an expert object-oriented programming just from reading this book. You know enough now, though, to get started.

This ends the book. I hope it has been useful in learning object-oriented

programming as understood by the R programming language. If you liked this book, why not check out my list of other books[1] or sign up to my mailing list[2]?

[1] http://wp.me/P9B2l-DN
[2] http://eepurl.com/cwIbR5