Advanced R (/) by Hadley Wickham

Table of contents

Want to learn from me in person? I'm next teaching in DC, Sep 14-15 (https://www.eventbrite.com/e/master-r-developer-workshop-washington-dc-tickets-15220403637).

Want a physical copy of this material? Buy a book from amazon! (http://amzn.com/1466586966?tag=devtools-20).

Contents

How to contribute (/contribute.html)

Edit this page (https://github.com/hadley/adv-r/edit/master/OO-essentials.rmd)

OO field guide

This chapter is a field guide for recognising and working with R's objects in the wild. R has three object oriented systems (plus the base types), so it can be a bit intimidating. The goal of this guide is not to make you an expert in all four systems, but to help you identify which system you're working with and to help you use it effectively.

Central to any object-oriented system are the concepts of class and method. A **class** defines the behaviour of **objects** by describing their attributes and their relationship to other classes. The class is also used when selecting **methods**, functions that behave differently depending on the class of their input. Classes are usually organised in a hierarchy: if a method does not exist for a child, then the parent's method is used instead; the child **inherits** behaviour from the parent.

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

- **S3** implements a style of OO programming called generic-function OO. This is different from most programming languages, like Java, C++, and C#, which implement message-passing OO. With message-passing, messages (methods) are sent to objects and the object determines which function to call. Typically, this object has a special appearance in the method call, usually appearing before the name of the method/message: e.g., `canvas.drawRect("blue")`. S3 is different. While computations are still carried out via methods, a special type of function called a **generic function** decides which method to call, e.g., `drawRect(canvas, "blue")`. S3 is a very casual system. It has no formal definition of classes.

- **S4** works similarly to S3, but is more formal. There are two major differences to S3. S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods. S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.
- **Reference classes**, called RC for short, are quite different from S3 and S4. RC implements message-passing OO, so methods belong to classes, not functions. `$` is used to separate objects and methods, so method calls look like `canvas$drawRect("blue")`. RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve with S3 or S4.

There's also one other system that's not quite OO, but it's important to mention here:

- **base types**, the internal C-level types that underlie the other OO systems. Base types are mostly manipulated using C code, but they're important to know about because they provide the building blocks for the other OO systems.

The following sections describe each system in turn, starting with base types. You'll learn how to recognise the OO system that an object belongs to, how method dispatch works, and how to create new objects, classes, generics, and methods for that system. The chapter concludes with a few remarks on when to use each system.

Prerequisites

You'll need the pryr package, `install.packages("pryr")`, to access useful functions for examining OO properties.

Quiz

Think you know this material already? If you can answer the following questions correctly, you can safely skip this chapter. Find the answers at the end of the chapter in answers (OO-essentials.html#oo-answers).

1. How do you tell what OO system (base, S3, S4, or RC) an object is associated with?
2. How do you determine the base type (like integer or list) of an object?
3. What is a generic function?
4. What are the main differences between S3 and S4? What are the main differences between S4 & RC?

Outline

- Base types (OO-essentials.html#base-types) teaches you about R's base object system. Only R-core can add new classes to this system, but it's important to know about because it underpins the three other systems.
- S3 (OO-essentials.html#s3) shows you the basics of the S3 object system. It's the simplest and most commonly used OO system.

- S4 (OO-essentials.html#s4) discusses the more formal and rigorous S4 system.
- RC (OO-essentials.html#rc) teaches you about R's newest OO system: reference classes, or RC for short.
- Picking a system (OO-essentials.html#picking-a-system) advises on which OO system to use if you're starting a new project.

Base types

Underlying every R object is a C structure (or struct) that describes how that object is stored in memory. The struct includes the contents of the object, the information needed for memory management, and, most importantly for this section, a **type**. This is the **base type** of an R object. Base types are not really an object system because only the R core team can create new types. As a result, new base types are added very rarely: the most recent change, in 2011, added two exotic types that you never see in R, but are useful for diagnosing memory problems (`NEWSXP` and `FREESXP`). Prior to that, the last type added was a special base type for S4 objects (`S4SXP`) in 2005.

Data structures (Data-structures.html#data-structures) explains the most common base types (atomic vectors and lists), but base types also encompass functions, environments, and other more exotic objects likes names, calls, and promises that you'll learn about later in the book. You can determine an object's base type with `typeof()`. Unfortunately the names of base types are not used consistently throughout R, and type and the corresponding "is" function may use different names:

```
# The type of a function is "closure"
f <- function() {}
typeof(f)
#> [1] "closure"
is.function(f)
#> [1] TRUE

# The type of a primitive function is "builtin"
typeof(sum)
#> [1] "builtin"
is.primitive(sum)
#> [1] TRUE
```

You may have heard of `mode()` and `storage.mode()`. I recommend ignoring these functions because they're just aliases of the names returned by `typeof()`, and exist solely for S compatibility. Read their source code if you want to understand exactly what they do.

Functions that behave differently for different base types are almost always written in C, where dispatch occurs using switch statements (e.g., `switch(TYPEOF(x))`). Even if you never write C code, it's important to understand base types because everything else is built on top of them: S3 objects can be built on top of any base type, S4 objects use a special base type, and RC objects are a combination of S4 and environments (another base type). To see if an object is a pure base type, i.e., it doesn't also have S3, S4, or RC behaviour, check that `is.object(x)` returns `FALSE`.

S3

S3 is R's first and simplest OO system. It is the only OO system used in the base and stats packages, and it's the most commonly used system in CRAN packages. S3 is informal and ad hoc, but it has a certain elegance in its minimalism: you can't take away any part of it and still have a useful OO system.

Recognising objects, generic functions, and methods

Most objects that you encounter are S3 objects. But unfortunately there's no simple way to test if an object is an S3 object in base R. The closest you can come is `is.object(x) & !isS4(x)`, i.e., it's an object, but not S4. An easier way is to use `pryr::otype()`:

```
library(pryr)

df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)    # A data frame is an S3 class
#> [1] "S3"
otype(df$x)  # A numeric vector isn't
#> [1] "base"
otype(df$y)  # A factor is
#> [1] "S3"
```

In S3, methods belong to functions, called **generic functions**, or generics for short. S3 methods do not belong to objects or classes. This is different from most other programming languages, but is a legitimate OO style.

To determine if a function is an S3 generic, you can inspect its source code for a call to `UseMethod()`: that's the function that figures out the correct method to call, the process of **method dispatch**. Similar to `otype()`, pryr also provides `ftype()` which describes the object system, if any, associated with a function:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x21b6ea8>
#> <environment: namespace:base>
ftype(mean)
#> [1] "s3"      "generic"
```

Some S3 generics, like `[`, `sum()`, and `cbind()`, don't call `UseMethod()` because they are implemented in C. Instead, they call the C functions `DispatchGroup()` or `DispatchOrEval()`. Functions that do method dispatch in C code are called **internal generics** and are documented in `?"internal generic"`. `ftype()` knows about these special cases too.

Given a class, the job of an S3 generic is to call the right S3 method. You can recognise S3 methods by their names, which look like `generic.class()`. For example, the Date method for the `mean()` generic is called `mean.Date()`, and the factor method for `print()` is called `print.factor()`.

This is the reason that most modern style guides discourage the use of `.` in function names: it makes them look like S3 methods. For example, is `t.test()` the `t` method for `test` objects? Similarly, the use of `.` in class names can also be confusing: is `print.data.frame()` the `print()` method for `data.frames`, or the `print.data()` method for `frames`? `pryr::ftype()` knows about these exceptions, so you can use it to figure out if a function is an S3 method or generic:

```
ftype(t.data.frame) # data frame method for t()
#> [1] "s3"     "method"
ftype(t.test)       # generic function for t tests
#> [1] "s3"      "generic"
```

You can see all the methods that belong to a generic with `methods()`:

```
methods("mean")
#> [1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt
#> see '?methods' for accessing help and source code
methods("t.test")
#> [1] t.test.default* t.test.formula*
#> see '?methods' for accessing help and source code
```

(Apart from methods defined in the base package, most S3 methods will not be visible: use `getS3method()` to read their source code.)

You can also list all generics that have a method for a given class:

```
methods(class = "ts")
#>  [1] aggregate     as.data.frame cbind         coerce        cycle
#>  [6] diffinv       diff          initialize    kernapply     lines
#> [11] Math2         Math          monthplot     na.omit       Ops
#> [16] plot          print         show          slotsFromS3   time
#> [21] [<-           [             t             window<-      window
#> see '?methods' for accessing help and source code
```

There’s no way to list all S3 classes, as you’ll learn in the following section.

Defining classes and creating objects

S3 is a simple and ad hoc system; it has no formal definition of a class. To make an object an instance of a class, you just take an existing base object and set the class attribute. You can do that during creation with `structure()`, or after the fact with `class<-()`:

```
# Create and assign class in one step
foo <- structure(list(), class = "foo")

# Create, then set class
foo <- list()
class(foo) <- "foo"
```

S3 objects are usually built on top of lists, or atomic vectors with attributes. (You can refresh your memory of attributes with attributes (Data-structures.html#attributes).) You can also turn functions into S3 objects. Other base types are either rarely seen in R, or have unusual semantics that don't work well with attributes.

You can determine the class of any object using `class(x)`, and see if an object inherits from a specific class using `inherits(x, "classname")`.

```
class(foo)
#> [1] "foo"
inherits(foo, "foo")
#> [1] TRUE
```

The class of an S3 object can be a vector, which describes behaviour from most to least specific. For example, the class of the `glm()` object is `c("glm", "lm")` indicating that generalised linear models inherit behaviour from linear models. Class names are usually lower case, and you should avoid `.`. Otherwise, opinion is mixed whether to use underscores (`my_class`) or CamelCase (`MyClass`) for multi-word class names.

Most S3 classes provide a constructor function:

```
foo <- function(x) {
  if (!is.numeric(x)) stop("X must be numeric")
  structure(list(x), class = "foo")
}
```

You should use it if it's available (like for `factor()` and `data.frame()`). This ensures that you're creating the class with the correct components. Constructor functions usually have the same name as the class.

Apart from developer supplied constructor functions, S3 has no checks for correctness. This means you can change the class of existing objects:

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
print(mod)
#>
#> Call:
#> lm(formula = log(mpg) ~ log(disp), data = mtcars)
#>
#> Coefficients:
#> (Intercept)    log(disp)
#>      5.3810      -0.4586

# Turn it into a data frame (?!)
class(mod) <- "data.frame"
# But unsurprisingly this doesn't work very well
print(mod)
#>  [1] coefficients  residuals     effects       rank          fitted.values
#>  [6] assign        qr            df.residual   xlevels       call
#> [11] terms         model
#> <0 rows> (or 0-length row.names)
# However, the data is still there
mod$coefficients
#> (Intercept)   log(disp)
#>   5.3809725  -0.4585683
```

If you've used other OO languages, this might make you feel queasy. But surprisingly, this flexibility causes few problems: while you *can* change the type of an object, you never should. R doesn't protect you from yourself: you can easily shoot yourself in the foot. As long as you don't aim the gun at your foot and pull the trigger, you won't have a problem.

Creating new methods and generics

To add a new generic, create a function that calls `UseMethod()`. `UseMethod()` takes two arguments: the name of the generic function, and the argument to use for method dispatch. If you omit the second argument it will dispatch on the first argument to the function. There's no need to pass any of the arguments of the generic to `UseMethod()` and you shouldn't do so. `UseMethod()` uses black magic to find them out for itself.

```
f <- function(x) UseMethod("f")
```

A generic isn't useful without some methods. To add a method, you just create a regular function with the correct (`generic.class`) name:

```
f.a <- function(x) "Class a"

a <- structure(list(), class = "a")
class(a)
#> [1] "a"
f(a)
#> [1] "Class a"
```

Adding a method to an existing generic works in the same way:

```
mean.a <- function(x) "a"
mean(a)
#> [1] "a"
```

As you can see, there's no check to make sure that the method returns the class compatible with the generic. It's up to you to make sure that your method doesn't violate the expectations of existing code.

Method dispatch

S3 method dispatch is relatively simple. `UseMethod()` creates a vector of function names, like `paste0("generic", ".", c(class(x), "default"))` and looks for each in turn. The "default" class makes it possible to set up a fall back method for otherwise unknown classes.

```
f <- function(x) UseMethod("f")
f.a <- function(x) "Class a"
f.default <- function(x) "Unknown class"

f(structure(list(), class = "a"))
#> [1] "Class a"
# No method for b class, so uses method for a class
f(structure(list(), class = c("b", "a")))
#> [1] "Class a"
# No method for c class, so falls back to default
f(structure(list(), class = "c"))
#> [1] "Unknown class"
```

Group generic methods add a little more complexity. Group generics make it possible to implement methods for multiple generics with one function. The four group generics and the functions they include are:

- Math: `abs`, `sign`, `sqrt`, `floor`, `cos`, `sin`, `log`, `exp`, `…`
- Ops: `+`, `-`, `*`, `/`, `^`, `%%`, `%/%`, `&`, `|`, `!`, `==`, `!=`, `<`, `<=`, `>=`, `>`
- Summary: `all`, `any`, `sum`, `prod`, `min`, `max`, `range`
- Complex: `Arg`, `Conj`, `Im`, `Mod`, `Re`

Group generics are a relatively advanced technique and are beyond the scope of this chapter but you can find out more about them in `?groupGeneric`. The most important thing to take away from this is to recognise that `Math`, `Ops`, `Summary`, and `Complex` aren't real functions, but instead represent groups of functions. Note that inside a group generic function a special variable `.Generic` provides the actual generic function called.

If you have complex class hierarchies it's sometimes useful to call the "parent" method. It's a little bit tricky to define exactly what that means, but it's basically the method that would have been called if the current method did not exist. Again, this is an advanced technique: you can read about it in `?NextMethod`.

Because methods are normal R functions, you can call them directly:

```
c <- structure(list(), class = "c")
# Call the correct method:
f.default(c)
#> [1] "Unknown class"
# Force R to call the wrong method:
f.a(c)
#> [1] "Class a"
```

However, this is just as dangerous as changing the class of an object, so you shouldn't do it. Please don't point the loaded gun at your foot! The only reason to call the method directly is that sometimes you can get considerable performance improvements by skipping method dispatch. See performance (Profiling.html#be-lazy) for details.

You can also call an S3 generic with a non-S3 object. Non-internal S3 generics will dispatch on the **implicit class** of base types. (Internal generics don't do that for performance reasons.) The rules to determine the implicit class of a base type are somewhat complex, but are shown in the function below:

```
iclass <- function(x) {
  if (is.object(x)) {
    stop("x is not a primitive type", call. = FALSE)
  }

  c(
    if (is.matrix(x)) "matrix",
    if (is.array(x) && !is.matrix(x)) "array",
    if (is.double(x)) "double",
    if (is.integer(x)) "integer",
    mode(x)
  )
}
iclass(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
iclass(array(1.5))
#> [1] "array"   "double"  "numeric"
```

Exercises

1. Read the source code for `t()` and `t.test()` and confirm that `t.test()` is an S3 generic and not an S3 method. What happens if you create an object with class `test` and call `t()` with it?
2. What classes have a method for the `Math` group generic in base R? Read the source code. How do the methods work?
3. R has two classes for representing date time data, `POSIXct` and `POSIXlt`, which both inherit from `POSIXt`. Which generics have different behaviours for the two classes? Which generics share the same behaviour?
4. Which base generic has the greatest number of defined methods?
5. `UseMethod()` calls methods in a special way. Predict what the following code will return, then run it and read the help for `UseMethod()` to figure out what's going on. Write down the rules in the simplest form possible.

   ```
   y <- 1
   g <- function(x) {
     y <- 2
     UseMethod("g")
   }
   g.numeric <- function(x) y
   g(10)

   h <- function(x) {
     x <- 10
     UseMethod("h")
   }
   h.character <- function(x) paste("char", x)
   h.numeric <- function(x) paste("num", x)

   h("a")
   ```

6. Internal generics don't dispatch on the implicit class of base types. Carefully read `?"internal generic"` to determine why the length of `f` and `g` is different in the example below. What function helps distinguish between the behaviour of `f` and `g`?

```
f <- function() 1
g <- function() 2
class(g) <- "function"

class(f)
class(g)

length.function <- function(x) "function"
length(f)
length(g)
```

S4

S4 works in a similar way to S3, but it adds formality and rigour. Methods still belong to functions, not classes, but:

- Classes have formal definitions which describe their fields and inheritance structures (parent classes).
- Method dispatch can be based on multiple arguments to a generic function, not just one.
- There is a special operator, `@`, for extracting slots (aka fields) from an S4 object.

All S4 related code is stored in the `methods` package. This package is always available when you're running R interactively, but may not be available when running R in batch mode. For this reason, it's a good idea to include an explicit `library(methods)` whenever you're using S4.

S4 is a rich and complex system. There's no way to explain it fully in a few pages. Here I'll focus on the key ideas underlying S4 so you can use existing S4 objects effectively. To learn more, some good references are:

- S4 system development in Bioconductor (http://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf)
- John Chambers' *Software for Data Analysis* (http://amzn.com/0387759352?tag=devtools-20)
- Martin Morgan's answers to S4 questions on stackoverflow (http://stackoverflow.com/search?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe)

Recognising objects, generic functions, and methods

Recognising S4 objects, generics, and methods is easy. You can identify an S4 object because `str()` describes it as a "formal" class, `isS4()` returns `TRUE`, and `pryr::otype()` returns "S4". S4 generics and methods are also easy to identify because they are S4 objects with well defined classes.

There aren't any S4 classes in the commonly used base packages (stats, graphics, utils, datasets, and base), so we'll start by creating an S4 object from the built-in stats4 package, which provides some S4 classes and methods associated with maximum likelihood estimation:

```
library(stats4)

# From example(mle)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) - sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))

# An S4 object
isS4(fit)
#> [1] TRUE
otype(fit)
#> [1] "S4"

# An S4 generic
isS4(nobs)
#> [1] TRUE
ftype(nobs)
#> [1] "s4"      "generic"

# Retrieve an S4 method, described later
mle_nobs <- method_from_call(nobs(fit))
isS4(mle_nobs)
#> [1] TRUE
ftype(mle_nobs)
#> [1] "s4"     "method"
```

Use `is()` with one argument to list all classes that an object inherits from. Use `is()` with two arguments to test if an object inherits from a specific class.

```
is(fit)
#> [1] "mle"
is(fit, "mle")
#> [1] TRUE
```

You can get a list of all S4 generics with `getGenerics()`, and a list of all S4 classes with `getClasses()`. This list includes shim classes for S3 classes and base types. You can list all S4 methods with `showMethods()`, optionally restricting selection either by `generic` or by `class` (or both). It's also a good idea to supply `where = search()` to restrict the search to methods available in the global environment.

Defining classes and creating objects

In S3, you can turn any object into an object of a particular class just by setting the class attribute. S4 is much stricter: you must define the representation of a class with `setClass()`, and create a new object with `new()`. You can find the documentation for a class with a special syntax: `class?className`, e.g., `class?mle`.

An S4 class has three key properties:

- A **name**: an alpha-numeric class identifier. By convention, S4 class names use UpperCamelCase.
- A named list of **slots** (fields), which defines slot names and permitted classes. For example, a person class might be represented by a character name and a numeric age: `list(name = "character", age = "numeric")`.
- A string giving the class it inherits from, or, in S4 terminology, that it **contains**. You can provide multiple classes for multiple inheritance, but this is an advanced technique which adds much complexity.

In `slots` and `contains` you can use S4 classes, S3 classes registered with `setOldClass()`, or the implicit class of a base type. In `slots` you can also use the special class `ANY` which does not restrict the input.

S4 classes have other optional properties like a `validity` method that tests if an object is valid, and a `prototype` object that defines default slot values. See `?setClass` for more details.

The following example creates a Person class with fields name and age, and an Employee class that inherits from Person. The Employee class inherits the slots and methods from the Person, and adds an additional slot, boss. To create objects we call `new()` with the name of the class, and name-value pairs of slot values.

```
setClass("Person",
  slots = list(name = "character", age = "numeric"))
setClass("Employee",
  slots = list(boss = "Person"),
  contains = "Person")

alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)
```

Most S4 classes also come with a constructor function with the same name as the class: if that exists, use it instead of calling `new()` directly.

To access slots of an S4 object use `@` or `slot()`:

```
alice@age
#> [1] 40
slot(john, "boss")
#> An object of class "Person"
#> Slot "name":
#> [1] "Alice"
#>
#> Slot "age":
#> [1] 40
```

(`@` is equivalent to `$`, and `slot()` to `[[`.)

If an S4 object contains (inherits from) an S3 class or a base type, it will have a special `.Data` slot which contains the underlying base type or S3 object:

```
setClass("RangedNumeric",
  contains = "numeric",
  slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10, min = 1, max = 10)
rn@min
#> [1] 1
rn@.Data
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Since R is an interactive programming language, it's possible to create new classes or redefine existing classes at any time. This can be a problem when you're interactively experimenting with S4. If you modify a class, make sure you also recreate any objects of that class, otherwise you'll end up with invalid objects.

Creating new methods and generics

S4 provides special functions for creating new generics and methods. `setGeneric()` creates a new generic or converts an existing function into a generic. `setMethod()` takes the name of the generic, the classes the method should be associated with, and a function that implements the method. For example, we could take `union()`, which usually just works on vectors, and make it work with data frames:

```
setGeneric("union")
#> [1] "union"
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
#> [1] "union"
```

If you create a new generic from scratch, you need to supply a function that calls `standardGeneric()`:

```
setGeneric("myGeneric", function(x) {
  standardGeneric("myGeneric")
})
#> [1] "myGeneric"
```

`standardGeneric()` is the S4 equivalent to `UseMethod()`.

Method dispatch

If an S4 generic dispatches on a single class with a single parent, then S4 method dispatch is the same as S3 dispatch. The main difference is how you set up default values: S4 uses the special class `ANY` to match any class and "missing" to match a missing argument. Like S3, S4 also has group generics, documented in `?S4groupGeneric`, and a way to call the "parent" method, `callNextMethod()`.

Method dispatch becomes considerably more complicated if you dispatch on multiple arguments, or if your classes use multiple inheritance. The rules are described in `?Methods`, but they are complicated and it's difficult to predict which method will be called. For this reason, I strongly recommend avoiding multiple inheritance and multiple dispatch unless absolutely necessary.

Finally, there are two methods that find which method gets called given the specification of a generic call:

```
# From methods: takes generic name and class names
selectMethod("nobs", list("mle"))

# From pryr: takes an unevaluated function call
method_from_call(nobs(fit))
```

Exercises

1. Which S4 generic has the most methods defined for it? Which S4 class has the most methods associated with it?
2. What happens if you define a new S4 class that doesn't "contain" an existing class? (Hint: read about virtual classes in `?Classes`.)
3. What happens if you pass an S4 object to an S3 generic? What happens if you pass an S3 object to an S4 generic? (Hint: read `?setOldClass` for the second case.)

RC

Reference classes (or RC for short) are the newest OO system in R. They were introduced in version 2.12. They are fundamentally different to S3 and S4 because:

- RC methods belong to objects, not functions
- RC objects are mutable: the usual R copy-on-modify semantics do not apply

These properties make RC objects behave more like objects do in most other programming languages, e.g., Python, Ruby, Java, and C#. Reference classes are implemented using R code: they are a special S4 class that wraps around an environment.

Defining classes and creating objects

Since there aren't any reference classes provided by the base R packages, we'll start by creating one. RC classes are best used for describing stateful objects, objects that change over time, so we'll create a simple class to model a bank account.

Creating a new RC class is similar to creating a new S4 class, but you use `setRefClass()` instead of `setClass()`. The first, and only required argument, is an alphanumeric **name**. While you can use `new()` to create new RC objects, it's good style to use the object returned by `setRefClass()` to generate new objects. (You can also do that with S4 classes, but it's less common.)

```
Account <- setRefClass("Account")
Account$new()
#> Reference class object of class "Account"
```

`setRefClass()` also accepts a list of name-class pairs that define class **fields** (equivalent to S4 slots). Additional named arguments passed to `new()` will set initial values of the fields. You can get and set field values with `$`:

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"))

a <- Account$new(balance = 100)
a$balance
#> [1] 100
a$balance <- 200
a$balance
#> [1] 200
```

Instead of supplying a class name for the field, you can provide a single argument function which will act as an accessor method. This allows you to add custom behaviour when getting or setting a field. See `?setRefClass` for more details.

Note that RC objects are **mutable**, i.e., they have reference semantics, and are not copied-on-modify:

```
b <- a
b$balance
#> [1] 200
a$balance <- 0
b$balance
#> [1] 0
```

For this reason, RC objects come with a `copy()` method that allow you to make a copy of the object:

```
c <- a$copy()
c$balance
#> [1] 0
a$balance <- 100
c$balance
#> [1] 0
```

An object is not very useful without some behaviour defined by **methods**. RC methods are associated with a class and can modify its fields in place. In the following example, note that you access the value of fields with their name, and modify them with `<<-`. You'll learn more about `<<-` in Environments (Environments.html#binding).

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      balance <<- balance - x
    },
    deposit = function(x) {
      balance <<- balance + x
    }
  )
)
```

You call an RC method in the same way as you access a field:

```
a <- Account$new(balance = 100)
a$deposit(100)
a$balance
#> [1] 200
```

The final important argument to `setRefClass()` is `contains`. This is the name of the parent RC class to inherit behaviour from. The following example creates a new type of bank account that returns an error preventing the balance from going below 0.

```
NoOverdraft <- setRefClass("NoOverdraft",
  contains = "Account",
  methods = list(
    withdraw = function(x) {
      if (balance < x) stop("Not enough money")
      balance <<- balance - x
    }
  )
)
accountJohn <- NoOverdraft$new(balance = 100)
accountJohn$deposit(50)
accountJohn$balance
#> [1] 150
accountJohn$withdraw(200)
#> Error in accountJohn$withdraw(200): Not enough money
```

All reference classes eventually inherit from `envRefClass`. It provides useful methods like `copy()` (shown above), `callSuper()` (to call the parent field), `field()` (to get the value of a field given its name), `export()` (equivalent to `as()`), and `show()` (overridden to control printing). See the inheritance section in `setRefClass()` for more details.

Recognising objects and methods

You can recognise RC objects because they are S4 objects (`isS4(x)`) that inherit from “refClass” (`is(x, "refClass")`). `pryr::otype()` will return “RC”. RC methods are also S4 objects, with class `refMethodDef`.

Method dispatch

Method dispatch is very simple in RC because methods are associated with classes, not functions. When you call `x$f()`, R will look for a method f in the class of x, then in its parent, then its parent’s parent, and so on. From within a method, you can call the parent method directly with `callSuper(...)`.

Exercises

1. Use a field function to prevent the account balance from being directly manipulated. (Hint: create a “hidden” `.balance` field, and read the help for the fields argument in `setRefClass()`.)
2. I claimed that there aren’t any RC classes in base R, but that was a bit of a simplification. Use `getClasses()` and find which classes `extend()` from `envRefClass`. What are the classes used for? (Hint: recall how to look up the documentation for a class.)

Picking a system

Three OO systems is a lot for one language, but for most R programming, S3 suffices. In R you usually create fairly simple objects and methods for pre-existing generic functions like `print()`, `summary()`, and `plot()`. S3 is well suited to this task, and the majority of OO code that I have written in R is S3. S3 is a little quirky, but it gets the job done with a minimum of code.

If you are creating more complicated systems of interrelated objects, S4 may be more appropriate. A good example is the `Matrix` package by Douglas Bates and Martin Maechler. It is designed to efficiently store and compute with many different types of sparse matrices. As of version 1.1.3, it defines 102 classes and 20 generic functions. The package is well written and well commented, and the accompanying vignette (`vignette("Intro2Matrix", package = "Matrix")`) gives a good overview of the structure of the package. S4 is also used extensively by Bioconductor packages, which need to model complicated interrelationships between biological objects. Bioconductor provides many good resources (https://www.google.com/search?q=bioconductor+s4) for learning S4. If you’ve mastered S3, S4 is relatively easy to pick up; the ideas are all the same, it is just more formal, more strict, and more verbose.

If you’ve programmed in a mainstream OO language, RC will seem very natural. But because they can introduce side effects through mutable state, they are harder to understand. For example, when you usually call `f(a, b)` in R you can assume that `a` and `b` will not be modified. But if `a` and `b` are RC objects, they might be modified in the place. Generally, when using RC objects you want to minimise side effects as much as possible, and use them only where mutable states are absolutely required. The majority of functions should still be “functional”, and free of side effects. This makes code easier to reason about and easier for other R programmers to understand.

Quiz answers

1. To determine the OO system of an object, you use a process of elimination. If `!is.object(x)`, it’s a base object. If `!isS4(x)`, it’s S3. If `!is(x, "refClass")`, it’s S4; otherwise it’s RC.

2. Use `typeof()` to determine the base class of an object.

3. A generic function calls specific methods depending on the class of it inputs. In S3 and S4 object systems, methods belong to generic functions, not classes like in other programming languages.

4. S4 is more formal than S3, and supports multiple inheritance and multiple dispatch. RC objects have reference semantics, and methods belong to classes, not functions.