**Unit 4 Study Guide (Chapter 10 – Finish)**

* A **function factory** is a function that makes functions.
* I’ll call square() and cube() **manufactured functions**, but this is just a term to ease communication with other humans: from R’s perspective they are no different to functions created any other way.
* The usual assignment operator, [<-](https://rdrr.io/r/base/assignOps.html), always creates a binding in the current environment. The super assignment operator, [<<-](https://rdrr.io/r/base/assignOps.html) rebinds an existing name found in a parent environment.

Garbage collection

* you can rely on the garbage collector to clean up any large temporary objects created inside a function. However, manufactured functions hold on to the execution environment, so you’ll need to explicitly unbind any large temporary objects with [rm()](https://rdrr.io/r/base/rm.html)

**10.4 Statistical factories**

* The Box-Cox transformation.
  + The Box-Cox transformation (a type of [power transformation](https://en.wikipedia.org/wiki/Power_transform)) is a flexible transformation often used to transform data towards normality. It has a single parameter, λλ, which controls the strength of the transformation. We could express the transformation as a simple two argument function
* Bootstrap resampling.
  + Function factories are a useful approach for bootstrapping. Instead of thinking about a single bootstrap (you always need more than one!), you can think about a bootstrap generator, a function that yields a fresh bootstrap every time it is called
* Maximum likelihood estimation.
  + The goal of maximum likelihood estimation (MLE) is to find the parameter values for a distribution that make the observed data most likely. To do MLE, you start with a probability function. For example, take the Poisson distribution. If we know λλ, we can compute the probability of getting a vector xx of values (x1x1, x2x2, …, xnxn) by multiplying the Poisson probability function

**Chapter 11 Notes**

function operator - a function that takes one (or more) functions as input and returns a function as output.

Function operators are closely related to function factories; indeed they’re just a function factory that takes a function as input. Like factories, there’s nothing you can’t do without them, but they often allow you to factor out complexity in order to make your code more readable and reusable.

**11.2 Existing function operators**

There are two very useful function operators that will both help you solve common recurring problems, and give you a sense for what function operators can do: [purrr::safely()](https://purrr.tidyverse.org/reference/safely.html) and [memoise::memoise()](https://rdrr.io/pkg/memoise/man/memoise.html).

Dynamic programming, where a complex problem can be broken down into many overlapping subproblems, and remembering the results of a subproblem considerably improves performance.

**Object-Orientated Progamming (OOP)**

There are three types of OOP focused on in the following chapters

S3, R6, and S4

S3 > R6 > S4

* S3 allows your functions to return rich results with user-friendly display and programmer-friendly internals. S3 is used throughout base R, so it’s important to master if you want to extend base R functions to work with new types of input.
* R6 provides a standardised way to escape R’s copy-on-modify semantics. This is particularly important if you want to model objects that exist independently of R. Today, a common need for R6 is to model data that comes from a web API, and where changes come from inside or outside of R.
* S4 is a rigorous system that forces you to think carefully about program design. It’s particularly well-suited for building large systems that evolve over time and will receive contributions from many programmers. This is why it is used by the Bioconductor project, so another reason to learn S4 is to equip you to contribute to that project.

OO systems call the type of an object its **class**, and an implementation for a specific class is called a **method**.

 The class defines the **fields**, the data possessed by every instance of that class. Classes are organised in a hierarchy so that if a method does not exist for one class, its parent’s method is used, and the child is said to **inherit** behaviour.

* In **encapsulated** OOP, methods belong to objects or classes, and method calls typically look like object.method(arg1, arg2). This is called encapsulated because the object encapsulates both data (with fields) and behaviour (with methods), and is the paradigm found in most popular languages.
* In **functional** OOP, methods belong to **generic** functions, and method calls look like ordinary function calls: generic(object, arg2, arg3). This is called functional because from the outside it looks like a regular function call, and internally the components are also functions.

Base R provides three OOP systems: S3, S4, and reference classes (RC):

* **S3** is R’s first OOP system, and is described in *Statistical Models in S*.62 S3 is an informal implementation of functional OOP and relies on common conventions rather than ironclad guarantees. This makes it easy to get started with, providing a low cost way of solving many simple problems.
* **S4** is a formal and rigorous rewrite of S3, and was introduced in *Programming with Data*.63 It requires more upfront work than S3, but in return provides more guarantees and greater encapsulation. S4 is implemented in the base **methods** package, which is always installed with R.

(You might wonder if S1 and S2 exist. They don’t: S3 and S4 were named according to the versions of S that they accompanied. The first two versions of S didn’t have any OOP framework.)

* **RC** implements encapsulated OO. RC objects are a special type of S4 objects that are also **mutable**, i.e., instead of using R’s usual copy-on-modify semantics, they can be modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve in the functional OOP style of S3 and S4.

Stuff that’s more in RStudio?

A number of other OOP systems are provided by CRAN packages:

* **R6**64 implements encapsulated OOP like RC, but resolves some important issues. In this book, you’ll learn about R6 instead of RC, for reasons described in Section [14.5](https://adv-r.hadley.nz/r6.html#why-r6).
* **R.oo**65 provides some formalism on top of S3, and makes it possible to have mutable S3 objects.
* **proto**66 implements another style of OOP based on the idea of **prototypes**, which blur the distinctions between classes and instances of classes (objects). I was briefly enamoured with prototype based programming67 and used it in ggplot2, but now think it’s better to stick with the standard forms.

**Chapter 12 Notes**

**12.2 Base versus OO objects**

Technically, the difference between base and OO objects is that OO objects have a “class” attribute:

**12.3 Base types**

While only OO objects have a class attribute, every object has a **base type**:

In total, there are 25 different base types.

Vectors, Functions, Environments, S4 Type, Language Components

**12.3.1 Numeric type**

In the S3 and S4 systems, numeric is used as a shorthand for either integer or double type, and is used when picking methods:

**Chapter 13 Notes**

S3 is R’s first and simplest OO system. S3 is informal and ad hoc, but there is a certain elegance in its minimalism: you can’t take away any part of it and still have a useful OO system.

**13.2 Basics**

An S3 object is a base type with at least a class attribute (other attributes may be used to store other data).

An S3 object behaves differently from its underlying base type whenever it’s passed to a **generic** (short for generic function).

The generic is a middleman: its job is to define the interface (i.e. the arguments) then find the right implementation for the job. The implementation for a specific class is called a **method**, and the generic finds that method by performing **method dispatch**.

**13.3 Classes**

to make an object an instance of a class, you simply set the **class attribute**.

You can do that during creation with [structure()](https://rdrr.io/r/base/structure.html), or after the fact with class<-():

The class name can be any string, but I recommend using only letters and \_. Avoid . because (as mentioned earlier) it can be confused with the . separator between a generic name and a class name.

* A low-level **constructor**, new\_myclass(), that efficiently creates new objects with the correct structure.
* A **validator**, validate\_myclass(), that performs more computationally expensive checks to ensure that the object has correct values.
* A user-friendly **helper**, myclass(), that provides a convenient way for others to create objects of your class.

13.3.1 Constructors

S3 doesn’t provide a formal definition of a class, so it has no built-in way to ensure that all objects of a given class have the same structure (i.e. the same base type and the same attributes with the same types). Instead, you must enforce a consistent structure by using a **constructor**.

The constructor should follow three principles:

* Be called new\_myclass().
* Have one argument for the base object, and one for each attribute.
* Check the type of the base object and the types of each attribute.

The purpose of constructors is to help you, the developer. That means you can keep them simple, and you don’t need to optimise error messages for public consumption.

13.3.2 Validators

More complicated classes require more complicated checks for validity. Take factors, for example. A constructor only checks that types are correct, making it possible to create malformed factors:

13.3.3 Helpers

If you want users to construct objects from your class, you should also provide a helper method that makes their life as easy as possible. A helper should always:

* Have the same name as the class, e.g. myclass().
* Finish by calling the constructor, and the validator, if it exists.
* Create carefully crafted error messages tailored towards an end-user.
* Have a thoughtfully crafted user interface with carefully chosen default values and useful conversions.

13.4 Generics and methods

The job of an S3 generic is to perform method dispatch, i.e. find the specific implementation for a class. Method dispatch is performed by [UseMethod()](https://rdrr.io/r/base/UseMethod.html), which every generic calls71.

The output here is simple:

* => indicates the method that is called, here [print.Date()](https://rdrr.io/r/base/Dates.html)
* [\*](https://rdrr.io/r/base/Arithmetic.html) indicates a method that is defined, but not called, here [print.default()](https://rdrr.io/r/base/print.default.html).

The “default” class is a special **pseudo-class**. This is not a real class, but is included to make it possible to define a standard fallback that is found whenever a class-specific method is not available.

Before we continue we need a bit of vocabulary to describe the relationship between the classes that appear together in a class vector. We’ll say that ordered is a **subclass** of factor because it always appears before it in the class vector, and, conversely, we’ll say factor is a **superclass** of ordered.

[NextMethod()](https://rdrr.io/r/base/UseMethod.html) is the hardest part of inheritance to understand, so we’ll start with a concrete example for the most common use case: [[](https://rdrr.io/r/base/Extract.html). We’ll start by creating a simple toy class: a secret class that hides its output when printed:

13.6.2 Allowing subclassing

**13.7 Dispatch details**

13.7.1 S3 and base types

But unfortunately dispatch actually occurs on the **implicit class**, which has three components:

13.7.2 Internal generics

Some base functions, like [[](https://rdrr.io/r/base/Extract.html), [sum()](https://rdrr.io/r/base/sum.html), and [cbind()](https://rdrr.io/r/base/cbind.html), are called **internal generics** because they don’t call [UseMethod()](https://rdrr.io/r/base/UseMethod.html) but instead call the C functions DispatchGroup() or DispatchOrEval(). [s3\_dispatch()](https://sloop.r-lib.org/reference/s3_dispatch.html) shows internal generics by including the name of the generic followed by (internal):

**13.7.3 Group generics**

There are four group generics:

* **Math**: [abs()](https://rdrr.io/r/base/MathFun.html), [sign()](https://rdrr.io/r/base/sign.html), [sqrt()](https://rdrr.io/r/base/MathFun.html), [floor()](https://rdrr.io/r/base/Round.html), [cos()](https://rdrr.io/r/base/Trig.html), [sin()](https://rdrr.io/r/base/Trig.html), [log()](https://rdrr.io/r/base/Log.html), and more (see [?Math](https://rdrr.io/r/base/groupGeneric.html) for the complete list).
* **Ops**: [+](https://rdrr.io/r/base/Arithmetic.html), [-](https://rdrr.io/r/base/Arithmetic.html), [\*](https://rdrr.io/r/base/Arithmetic.html), [/](https://rdrr.io/r/base/Arithmetic.html), [^](https://rdrr.io/r/base/Arithmetic.html), [%%](https://rdrr.io/r/base/Arithmetic.html), [%/%](https://rdrr.io/r/base/Arithmetic.html), [&](https://rdrr.io/r/base/Logic.html), [|](https://rdrr.io/r/base/Logic.html), [!](https://rdrr.io/r/base/Logic.html), [==](https://rdrr.io/r/base/Comparison.html), [!=](https://rdrr.io/r/base/Comparison.html), [<](https://rdrr.io/r/base/Comparison.html), [<=](https://rdrr.io/r/base/Comparison.html), [>=](https://rdrr.io/r/base/Comparison.html), and [>](https://rdrr.io/r/base/Comparison.html).
* **Summary**: [all()](https://rdrr.io/r/base/all.html), [any()](https://rdrr.io/r/base/any.html), [sum()](https://rdrr.io/r/base/sum.html), [prod()](https://rdrr.io/r/base/prod.html), [min()](https://rdrr.io/r/base/Extremes.html), [max()](https://rdrr.io/r/base/Extremes.html), and [range()](https://rdrr.io/r/base/range.html).
* **Complex**: [Arg()](https://rdrr.io/r/base/complex.html), [Conj()](https://rdrr.io/r/base/complex.html), [Im()](https://rdrr.io/r/base/complex.html), [Mod()](https://rdrr.io/r/base/complex.html), [Re()](https://rdrr.io/r/base/complex.html).

**Chapter 14 Notes**

R6 has two special properties:

* It uses the encapsulated OOP paradigm, which means that methods belong to objects, not generics, and you call them like object$method().
* R6 objects are mutable, which means that they are modified in place, and hence have reference semantics.

R6 is very similar to a base OOP system called **reference classes**, or RC

R6 only needs a single function call to create both the class and its methods: [R6::R6Class()](https://r6.r-lib.org/reference/R6Class.html)

x$

**14.2.1 Method chaining**

add(10)$

add(10)$

sum

#> [1] 44

This technique is called **method chaining**

**14.2.2 Important methods**

There are two important methods that should be defined for most classes: $initialize() and $print(). They’re not required, but providing them will make your class easier to use.

$initialize() overrides the default behaviour of $new()

**14.2.3 Adding methods after creation**

Add new elements to an existing class with $set()

**14.2.4 Inheritance**

$add() overrides the superclass implementation, but we can still delegate to the superclass implementation by using super$

**14.3 Controlling access**

[R6Class()](https://r6.r-lib.org/reference/R6Class.html) has two other arguments that work similarly to public:

* private allows you to create fields and methods that are only available from within the class, not outside of it.
* active allows you to use accessor functions to define dynamic, or active, fields.

**14.3.1 Privacy**

With R6 you can define **private** fields and methods, elements that can only be accessed from within the class, not from the outside76.

* The private argument to R6Class works in the same way as the public argument: you give it a named list of methods (functions) and fields (everything else).
* Fields and methods defined in private are available within the methods using private$ instead of self$. You cannot access private fields or methods outside of the class.

Private methods tend to be less important in R compared to other programming languages because the object hierarchies in R tend to be simpler.

**14.3.2 Active fields**

Active fields allow you to define components that look like fields from the outside, but are defined with functions, like methods. Active fields are implemented using **active bindings**

**14.4 Reference semantics**

One of the big differences between R6 and most other objects is that they have reference semantics.

if you want a copy, you’ll need to explicitly $clone() the object

There are three other less obvious consequences:

* It is harder to reason about code that uses R6 objects because you need to understand more context.
* It makes sense to think about when an R6 object is deleted, and you can write a $finalize() to complement the $initialize().
* If one of the fields is an R6 object, you must create it inside $initialize(), not [R6Class()](https://r6.r-lib.org/reference/R6Class.html).

**14.4.1 Reasoning**

Generally, reference semantics makes code harder to reason about.

if f() calls methods of x or y, it might modify them as well as z. This is the biggest potential downside of R6 and you should take care to avoid it by writing functions that either return a value, or modify their R6 inputs, but not both.

**14.4.2 Finalizer**

One useful property of reference semantics is that it makes sense to think about when an R6 object is **finalized**, i.e. when it’s deleted.

**14.4.3 R6 fields**

A final consequence of reference semantics can crop up where you don’t expect it. If you use an R6 class as the default value of a field, it will be shared across all instances of the object!

**14.5 Why R6?**

R6 is very similar to a built-in OO system called **reference classes**, or RC for short.

* R6 is much simpler. Both R6 and RC are built on top of environments, but while R6 uses S3, RC uses S4. This means to fully understand RC, you need to understand how the more complicated S4 works.
* R6 has comprehensive online documentation at [https://r6.r-lib.org](https://r6.r-lib.org/).
* R6 has a simpler mechanism for cross-package subclassing, which just works without you having to think about it. For RC, read the details in the “External Methods; Inter-Package Superclasses” section of ?setRefClass.
* RC mingles variables and fields in the same stack of environments so that you get (field) and set (field <<- value) fields like regular values. R6 puts fields in a separate environment so you get (self$field) and set (self$field <- value) with a prefix. The R6 approach is more verbose but I like it because it is more explicit.
* R6 is much faster than RC. Generally, the speed of method dispatch is not important outside of microbenchmarks. However, RC is quite slow, and switching from RC to R6 led to a substantial performance improvement in the shiny package. For more details, see [vignette("Performance", "R6")](https://r6.r-lib.org/articles/Performance.html).
* RC is tied to R. That means if any bugs are fixed, you can only take advantage of the fixes by requiring a newer version of R. This makes it difficult for packages (like those in the tidyverse) that need to work across many R versions.
* Finally, because the ideas that underlie R6 and RC are similar, it will only require a small amount of additional effort to learn RC if you need to.

Chapter 15 Notes

S4 provides a formal approach to functional OOP. The underlying ideas are similar to S3 (the topic of Chapter [13](https://adv-r.hadley.nz/s3.html#s3)), but implementation is much stricter and makes use of specialised functions for creating classes ([setClass()](https://rdrr.io/r/methods/setClass.html)), generics ([setGeneric()](https://rdrr.io/r/methods/setGeneric.html)), and methods ([setMethod()](https://rdrr.io/r/methods/setMethod.html)).

An important new component of S4 is the **slot**, a named component of the object that is accessed using the specialised subsetting operator [@](https://rdrr.io/r/base/slotOp.html) (pronounced at).

We’ll start with a quick overview of the main components of S4. You define an S4 class by calling [setClass()](https://rdrr.io/r/methods/setClass.html)

setClass("Person",

slots = c(

name = "character",

age = "numeric"

)

)

john <- new("Person", name = "John Smith", age = NA\_real\_)

is(john)

#> [1] "Person"

john@name

#> [1] "John Smith"

slot(john, "age")

#> [1] NA

Generally, you should only use [@](https://rdrr.io/r/base/slotOp.html) in your methods. If you’re working with someone else’s class, look for **accessor** functions that allow you to safely set and get slot values.

Here we’ll create a setter and getter for the age slot by first creating generics with [setGeneric()](https://rdrr.io/r/methods/setGeneric.html):

setGeneric("age", function(x) standardGeneric("age"))

setGeneric("age<-", function(x, value) standardGeneric("age<-"))

**15.3 Classes**

To define an S4 class, call [setClass()](https://rdrr.io/r/methods/setClass.html) with three arguments:

* The class **name**. By convention, S4 class names use UpperCamelCase.
* A named character vector that describes the names and classes of the **slots** (fields). For example, a person might be represented by a character name and a numeric age: [c(name = "character", age = "numeric")](https://rdrr.io/r/base/c.html). The pseudo-class ANY allows a slot to accept objects of any type.
* A **prototype**, a list of default values for each slot. Technically, the prototype is optional80, but you should always provide it.

**15.3.1 Inheritance**

There is one other important argument to [setClass()](https://rdrr.io/r/methods/setClass.html): contains. This specifies a class (or classes) to inherit slots and behaviour from.

**15.3.2 Introspection**

To determine what classes an object inherits from, use [is()](https://rdrr.io/r/methods/is.html):

To test if an object inherits from a specific class, use the second argument of [is()](https://rdrr.io/r/methods/is.html):

**15.3.3 Redefinition**

In most programming languages, class definition occurs at compile-time and object construction occurs later, at run-time. In R, however, both definition and construction occur at run time. When you call [setClass()](https://rdrr.io/r/methods/setClass.html), you are registering a class definition in a (hidden) global variable. As with all state-modifying functions you need to use [setClass()](https://rdrr.io/r/methods/setClass.html) with care. It’s possible to create invalid objects if you redefine a class after already having instantiated an object:

**15.3.4 Helper**

[new()](https://rdrr.io/r/methods/new.html) is a low-level constructor suitable for use by you, the developer. User-facing classes should always be paired with a user-friendly helper. A helper should always:

* Have the same name as the class, e.g. myclass().
* Have a thoughtfully crafted user interface with carefully chosen default values and useful conversions.
* Create carefully crafted error messages tailored towards an end-user.
* Finish by calling [methods::new()](https://rdrr.io/r/methods/new.html).

**15.3.5 Validator**

The constructor automatically checks that the slots have correct classes:

To enforce these additional constraints we write a validator with [setValidity()](https://rdrr.io/r/methods/validObject.html). It takes a class and a function that returns TRUE if the input is valid, and otherwise returns a character vector describing the problem(s):

**15.4 Generics and methods**

The job of a generic is to perform method dispatch, i.e. find the specific implementation for the combination of classes passed to the generic. Here you’ll learn how to define S4 generics and methods, then in the next section we’ll explore precisely how S4 method dispatch works.

**15.4.1 Signature**

Like [setClass()](https://rdrr.io/r/methods/setClass.html), [setGeneric()](https://rdrr.io/r/methods/setGeneric.html) has many other arguments. There is only one that you need to know about: signature. This allows you to control the arguments that are used for method dispatch. If signature is not supplied, all arguments (apart from ...) are used. It is occasionally useful to remove arguments from dispatch. This allows you to require that methods provide arguments like verbose = TRUE or quiet = FALSE, but they don’t take part in dispatch.

**15.4.2 Methods**

A generic isn’t useful without some methods, and in S4 you define methods with [setMethod()](https://rdrr.io/r/methods/setMethod.html). There are three important arguments: the name of the generic, the name of the class, and the method itself.

More formally, the second argument to [setMethod()](https://rdrr.io/r/methods/setMethod.html) is called the **signature**. In S4, unlike S3, the signature can include multiple arguments. This makes method dispatch in S4 substantially more complicated, but avoids having to implement double-dispatch as a special case.

**15.4.3 Show method**

The most commonly defined S4 method that controls printing is [show()](https://rdrr.io/r/methods/show.html), which controls how the object appears when it is printed. To define a method for an existing generic, you must first determine the arguments. You can get those from the documentation or by looking at the [args()](https://rdrr.io/r/base/args.html) of the generic:

**15.4.4 Accessors**

Slots should be considered an internal implementation detail: they can change without warning and user code should avoid accessing them directly. Instead, all user-accessible slots should be accompanied by a pair of **accessors**. If the slot is unique to the class, this can just be a function:

**15.5 Method dispatch**

S4 dispatch is complicated because S4 has two important features:

* Multiple inheritance, i.e. a class can have multiple parents,
* Multiple dispatch, i.e. a generic can use multiple arguments to pick a method.

These features make S4 very powerful, but can also make it hard to understand which method will get selected for a given combination of inputs.

But it’s important to describe the full details, so here we’ll start simple with single inheritance and single dispatch, and work our way up to the more complicated cases.

**15.5.1 Single dispatch**

A diagram of a yellow smiley face

Description automatically generated

There are two parts to this diagram:

* The top part, f(...), defines the scope of the diagram. Here we have a generic with one argument, that has a class hierarchy that is three levels deep.
* The bottom part is the **method graph** and displays all the possible methods that could be defined. Methods that exist, i.e. that have been defined with [setMethod()](https://rdrr.io/r/methods/setMethod.html), have a grey background.

There are two **pseudo-classes** that you can define methods for. These are called pseudo-classes because they don’t actually exist, but allow you to define useful behaviours. The first pseudo-class is ANY which matches any class81. For technical reasons that we’ll get to later, the link to the ANY method is longer than the links between the other classes: A yellow smiley face in a diamond shape

Description automatically generated with medium confidence

**15.5.2 Multiple inheritance**

The basic process remains the same: you start from the actual class supplied to the generic, then follow the arrows until you find a defined method. The wrinkle is that now there are multiple arrows to follow, so you might find multiple methods. If that happens, you pick the method that is closest, i.e. requires travelling the fewest arrows.

This is called an **ambiguous** method, and in diagrams I’ll illustrate it with a thick dotted border. When this happens in R, you’ll get a warning, and the method for the class that comes earlier in the alphabet will be picked (this is effectively random and should not be relied upon).

The fallback ANY method still exists but the rules are little more complex. As indicated by the wavy dotted lines, the ANY method is always considered further away than a method for a real class. This means that it will never contribute to ambiguity.

With multiple inheritances it is hard to simultaneously prevent ambiguity, ensure that every terminal method has an implementation, and minimise the number of defined methods (in order to benefit from OOP).

**15.5.3 Multiple dispatch**

Once you understand multiple inheritance, understanding multiple dispatch is straightforward.

Multiple dispatch tends to be less tricky to work with than multiple inheritance because there are usually fewer terminal class combinations.

**15.5.4 Multiple dispatch and multiple inheritance**

As the method graph gets more and more complicated it gets harder and harder to predict which method will get called given a combination of inputs, and it gets harder and harder to make sure that you haven’t introduced ambiguity.

**15.6 S4 and S3**

When writing S4 code, you’ll often need to interact with existing S3 classes and generics.

**15.6.1 Classes**

In slots and contains you can use S4 classes, S3 classes, or the implicit class (Section [13.7.1](https://adv-r.hadley.nz/s3.html#implicit-class)) of a base type. To use an S3 class, you must first register it with [setOldClass()](https://rdrr.io/r/methods/setOldClass.html). You call this function once for each S3 class, giving it the class attribute.

However, it’s generally better to be more specific and provide a full S4 definition with slots and a prototype.

Generally, these definitions should be provided by the creator of the S3 class. If you’re trying to build an S4 class on top of an S3 class provided by a package, you should request that the package maintainer add this call to their package, rather than adding it to your own code.

If an S4 object inherits from an S3 class or a base type, it will have a special virtual slot called .Data. This contains the underlying base type or S3 object:

**15.6.2 Generics**

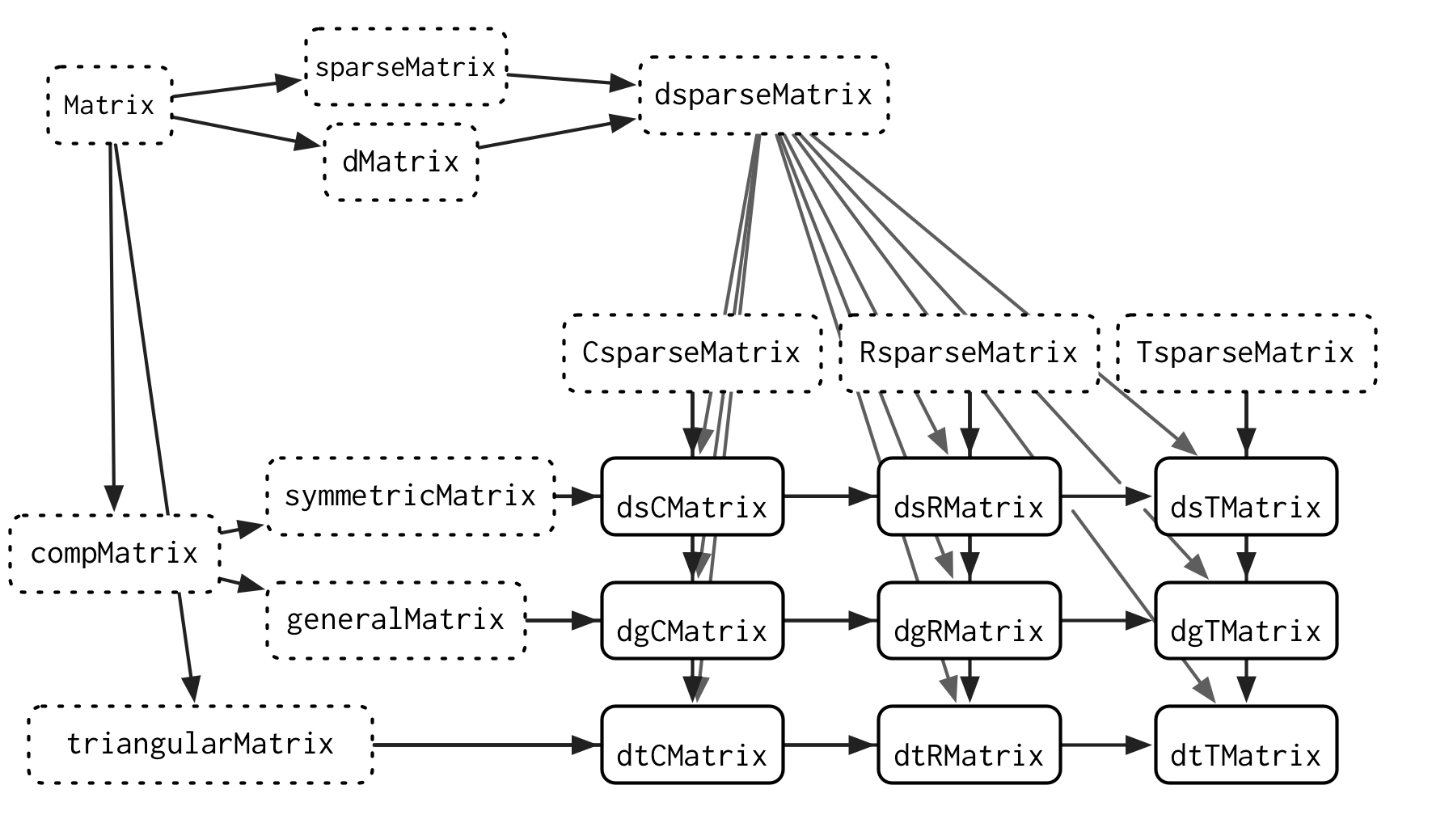
As well as creating a new generic from scratch, it’s also possible to convert an existing S3 generic to an S4 generic:

**Chapter 16 Notes**

You now know about the three most important OOP toolkits available in R. Now that you understand their basic operation and the principles that underlie them, we can start to compare and contrast the systems in order to understand their strengths and weaknesses. This will help you pick the system that is most likely to solve new problems.

Overall, when picking an OO system, I recommend that you default to S3. S3 is simple, and widely used throughout base R and CRAN. While it’s far from perfect, its idiosyncrasies are well understood and there are known approaches to overcome most shortcomings. If you have an existing background in programming you are likely to lean towards R6, because it will feel familiar.

**16.2 S4 versus S3**



S4 is also a good fit for complex systems of interrelated objects, and it’s possible to minimise code duplication through careful implementation of methods. The best example of such a system is the Matrix package.82 It is designed to efficiently store and compute with many different types of sparse and dense matrices.

This domain is a good fit for S4 because there are often computational shortcuts for specific combinations of sparse matrices. S4 makes it easy to provide a general method that works for all inputs, and then provide more specialised methods where the inputs allow a more efficient implementation. This requires careful planning to avoid method dispatch ambiguity, but the planning pays off with higher performance.

The biggest challenge to using S4 is the combination of increased complexity and absence of a single source of documentation. S4 is a complex system and it can be challenging to use effectively in practice.

**16.3 R6 versus S3**

R6 is a profoundly different OO system from S3 and S4 because it is built on encapsulated objects, rather than generic functions. Additionally R6 objects have reference semantics, which means that they can be modified in place. These two big differences have a number of non-obvious consequences which we’ll explore here:

* A generic is a regular function so it lives in the global namespace. An R6 method belongs to an object so it lives in a local namespace. This influences how we think about naming.
* R6’s reference semantics allow methods to simultaneously return a value and modify an object. This solves a painful problem called “threading state”.
* You invoke an R6 method using [$](https://rdrr.io/r/base/Extract.html), which is an infix operator. If you set up your methods correctly you can use chains of method calls as an alternative to the pipe.

**16.3.1 Namespacing**

One non-obvious difference between S3 and R6 is the space in which methods are found:

* Generic functions are global: all packages share the same namespace.
* Encapsulated methods are local: methods are bound to a single object.

The advantage of a global namespace is that multiple packages can use the same verbs for working with different types of objects. Generic functions provide a uniform API that makes it easier to perform typical actions with a new object because there are strong naming conventions. This works well for data analysis because you often want to do the same thing to different types of objects. In particular, this is one reason that R’s modelling system is so useful: regardless of where the model has been implemented you always work with it using the same set of tools ([summary()](https://rdrr.io/r/base/summary.html), [predict()](https://rdrr.io/r/stats/predict.html), …).

**16.3.2 Threading state**

For example, imagine you want to create a **stack** of objects. A stack has two main methods:

* push() adds a new object to the top of the stack.
* pop() returns the top most value, and removes it from the stack.

The implementation of the constructor and the push() method is straightforward. A stack contains a list of items, and pushing an object to the stack simply appends to this list.

This problem is known as **threading state** or **accumulator programming**, because no matter how deeply the pop() is called, you have to thread the modified stack object all the way back to where it lives.

One way that other FP languages deal with this challenge is to provide a **multiple assign** (or destructuring bind) operator that allows you to assign multiple values in a single step. The zeallot package83 provides multi-assign for R with [%<-%](https://rdrr.io/pkg/zeallot/man/operator.html). This makes the code more elegant, but doesn’t solve the key problem:

**16.3.3 Method chaining**

The pipe, [%>%](https://purrr.tidyverse.org/reference/pipe.html), is useful because it provides an infix operator that makes it easy to compose functions from left-to-right. Interestingly, the pipe is not so important for R6 objects because they already use an infix operator: [$](https://rdrr.io/r/base/Extract.html). This allows the user to chain together multiple method calls in a single expression, a technique known as **method chaining**: