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:

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}

new_factor(1:5, "a")
#> Error in as.character.factor(x): malformed factor
new_factor(0:1, "a")
#> Error in as.character.factor(x): malformed factor
```

Rather than encumbering the constructor with complicated checks, it's better to put them in a separate function. Doing so allows you to cheaply create new objects when you know that the values are correct, and easily re-use the checks in other places.

```
validate_factor <- function(x) {
  values <- unclass(x)
  levels <- attr(x, "levels")

  if (!all(!is.na(values) & values > 0)) {
    stop(
      "All `x` values must be non-missing and greater than zero",
      call. = FALSE
    )
  }

  if (length(levels) < max(values)) {
    stop(
      "There must be at least as many `levels` as possible values in `x`",
      call. = FALSE
    )
  }

  x
}

validate_factor(new_factor(1:5, "a"))
#> Error: There must be at least as many `levels` as possible values in `x`
validate_factor(new_factor(0:1, "a"))
#> Error: All `x` values must be non-missing and greater than zero
```

This validator function is called primarily for its side-effects (throwing an error if the object is invalid) so you'd expect it to invisibly return its primary input. However, it's useful for validation methods to return visibly, as we'll see next.

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.

The last bullet is the trickiest, and it's hard to give general advice. However, there are three common patterns:

Sometimes all the helper needs to do is coerce its inputs to the desired type. For example, new_difftime() is very strict, and violates the usual convention that you can use an integer vector wherever you can use a double vector:

```
new_difftime(1:10)
#> Error in new_difftime(1:10): is.double(x) is not TRUE
```

It's not the job of the constructor to be flexible, so here we create a helper that just coerces the input to a double.

```
difftime <- function(x = double(), units = "secs") {
  x <- as.double(x)
  new_difftime(x, units = units)
}

difftime(1:10)
#> Time differences in secs
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Often, the most natural representation of a complex object is a string. For example, it's very convenient to specify factors with a character vector. The code below shows a simple version of factor(): it takes a character vector, and guesses that the levels should be the unique values. This is not always correct (since some levels might not be seen in the data), but it's a useful default.

```
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  validate_factor(new_factor(ind, levels))
}

factor(c("a", "a", "b"))
#> [1] a a b
#> Levels: a b
```

Some complex objects are most naturally specified by multiple simple

components. For example, I think it's natural to construct a date-time by supplying the individual components (year, month, day etc). That leads me to this POSIXct() helper that resembles the existing ISODatetime() function:

```
POSIXct <- function(year = integer(),
                    month = integer(),
                    day = integer(),
                    hour = 0L,
                    minute = 0L,
                    sec = 0,
                    tzone = "") {
  ISOdatetime(year, month, day, hour, minute, sec, tz = tzone)
```

```
}

POSIXct(2020, 1, 1, tzone = "America/New_York")
#> [1] "2020-01-01 EST"
```

For more complicated classes, you should feel free to go beyond these patterns to make life as easy as possible for your users.

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(), which every generic calls. UseMethod() takes two arguments: the name of the generic function (required), and the argument to use for method dispatch (optional). If you omit the second argument, it will dispatch based on the first argument, which is almost always what is desired.

Most generics are very simple, and consist of only a call to UseMethod(). Take mean() for example:

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x2cd4688>
#> <environment: namespace:base>
```

Creating your own generic is similarly simple:

```
my_new_generic <- function(x) {
  UseMethod("my_new_generic")
}
```

You don't pass any of the arguments of the generic to UseMethod(); it uses deep magic to pass to the method automatically. The precise process is complicated and frequently surprising, so you should avoid doing any computation in a generic. To learn the full details, carefully read the Technical Details section in ?UseMethod.

Method dispatch

How does UseMethod() work? It basically creates a vector of method names, paste0("generic", ".", c(class(x), "default")), and then looks for each potential method in turn. We can see this in action with sloop::s3_dispatch(). You give it a call to an S3 generic, and it lists all the possible methods. For example, what method is called when you print a Date object?

```
x <- Sys.Date()
s3_dispatch(print(x))
#> => print.Date
#>  * print.default
```

The output here is simple:

=> indicates the method that is called, here print.Date()

* indicates a method that is defined, but not called, here print.default().

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.

The essence of method dispatch is quite simple, but as the chapter proceeds you'll see it get progressively more complicated to encompass inheritance, base types, internal generics, and group generics.

```
x <- matrix(1:10, nrow = 2)
s3_dispatch(mean(x))
#>    mean.matrix
#>    mean.integer
#>    mean.numeric
#> => mean.default

s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
#>    Summary.default
#> -> sum (internal)
```

Finding methods

sloop::s3_dispatch() lets you find the specific method used for a single call. What if you want to find all methods defined for a generic or associated with a class? That's the job of sloop::s3_methods_generic() and sloop::s3_methods_class():

```
s3_methods_generic("mean")
#> # A tibble: 6 x 4
#>   generic class    visible source
#>   <chr>   <chr>    <lgl>   <chr>
#> 1 mean    Date     TRUE    base
#> 2 mean    default  TRUE    base
#> 3 mean    difftime TRUE    base
#> 4 mean    POSIXct  TRUE    base
#> 5 mean    POSIXlt  TRUE    base
#> 6 mean    quosure  FALSE   registered S3method

s3_methods_class("ordered")
#> # A tibble: 4 x 4
#>   generic       class   visible source
#>   <chr>         <chr>   <lgl>   <chr>
#> 1 as.data.frame ordered TRUE    base
#> 2 Ops           ordered TRUE    base
#> 3 relevel       ordered FALSE   registered S3method
#> 4 Summary       ordered TRUE    base
```

Creating methods

There are two wrinkles to be aware of when you create a new method:

First, you should only ever write a method if you own the generic or the class. R will allow you to define a method even if you don't, but it is exceedingly bad manners. Instead, work with the author of either the generic or the class to add the method in their code.

A method must have the same arguments as its generic. This is enforced in packages by R CMD check, but it's good practice even if you're not creating a package.

There is one exception to this rule: if the generic has ..., the method can contain a superset of the arguments. This allows methods to take arbitrary additional arguments. The downside of using ..., however, is that any misspelled arguments will be silently swallowed.

Object styles

So far I've focussed on vector style classes like Date and factor. These have the key property that length(x) represents the number of observations in the vector. There are three variants that do not have this property:

Record style objects use a list of equal-length vectors to represent individual components of the object. The best example of this is POSIXlt, which underneath the hood is a list of 11 date-time components like year, month, and day. Record style classes override length() and subsetting methods to conceal this implementation detail.

```
x <- as.POSIXlt(ISOdatetime(2020, 1, 1, 0, 0, 1:3))
x
#> [1] "2020-01-01 00:00:01 UTC" "2020-01-01 00:00:02 UTC"
#> [3] "2020-01-01 00:00:03 UTC"

length(x)
#> [1] 3
length(unclass(x))
#> [1] 9

x[[1]] # the first date time
#> [1] "2020-01-01 00:00:01 UTC"
unclass(x)[[1]] # the first component, the number of seconds
#> [1] 1 2 3
```

Data frames are similar to record style objects in that both use lists of equal length vectors. However, data frames are conceptually two dimensional, and the individual components are readily exposed to the user. The number of observations is the number of rows, not the length:

```
x <- data.frame(x = 1:100, y = 1:100)
length(x)
#> [1] 2
nrow(x)
#> [1] 100
```

Scalar objects typically use a list to represent a single thing. For example, an lm object is a list of length 12 but it represents one model.

```
mod <- lm(mpg ~ wt, data = mtcars)
length(mod)
#> [1] 12
```

Scalar objects can also be built on top of functions, calls, and environments. This is less generally useful, but you can see applications in stats::ecdf(), R6, and rlang::quo().

Unfortunately, describing the appropriate use of each of these object styles is beyond the scope of this book. However, you can learn more from the documentation of the vctrs package (https://vctrs.r-lib.org); the package also provides constructors and helpers that make implementation of the different styles easier.

Inheritance

S3 classes can share behaviour through a mechanism called inheritance. Inheritance is powered by three ideas:

The class can be a character vector. For example, the ordered and POSIXct classes have two components in their class:

```
class(ordered("x"))
#> [1] "ordered" "factor"
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

If a method is not found for the class in the first element of the vector, R looks for a method for the second class (and so on):

```
s3_dispatch(print(ordered("x")))
#>    print.ordered
#> => print.factor
#>  * print.default
s3_dispatch(print(Sys.time()))
#> => print.POSIXct
#>    print.POSIXt
#>  * print.default
```

A method can delegate work by calling NextMethod(). We'll come back to that very shortly; for now, note that s3_dispatch() reports delegation with ->.

```
s3_dispatch(ordered("x")[1])
#>    [.ordered
#> => [.factor
#>    [.default
#> -> [ (internal)
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

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.

S3 imposes no restrictions on the relationship between sub- and superclasses but your life will be easier if you impose some. I recommend that you adhere to two simple principles when creating a subclass:

The base type of the subclass should be that same as the superclass.

The attributes of the subclass should be a superset of the attributes of the superclass.

POSIXt does not adhere to these principles because POSIXct has type double, and POSIXlt has type list. This means that POSIXt is not a superclass, and illustrates that it's quite possible to use the S3 inheritance system to implement other styles of code sharing (here POSIXt plays a role more like an interface), but you'll need to figure out safe conventions yourself.

NextMethod()

NextMethod() is the hardest part of inheritance to understand, so we'll start with a concrete example for the most common use case: [. We'll start by creating a simple toy class: a secret class that hides its output when printed:

```
new_secret <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "secret")
}
```

```
print.secret <- function(x, ...) {
  print(strrep("x", nchar(x)))
  invisible(x)
}

x <- new_secret(c(15, 1, 456))
x
#> [1] "xx"  "x"   "xxx"
```

This works, but the default [method doesn't preserve the class:

```
s3_dispatch(x[1])
#>    [.secret
#>    [.default
#> => [ (internal)
x[1]
#> [1] 15
```

To fix this, we need to provide a [.secret method. How could we implement this method? The naive approach won't work because we'll get stuck in an infinite loop:

```
`[.secret` <- function(x, i) {
  new_secret(x[i])
}
```

Instead, we need some way to call the underlying [code, i.e. the implementation that would get called if we didn't have a [.secret method. One approach would be to unclass() the object:

```
`[.secret` <- function(x, i) {
  x <- unclass(x)
  new_secret(x[i])
}
x[1]
#> [1] "xx"
```

This works, but is inefficient because it creates a copy of x. A better approach is to use NextMethod(), which concisely solves the problem delegating to the method that would've have been called if [.secret didn't exist:

```
`[.secret` <- function(x, i) {
  new_secret(NextMethod())
}
x[1]
#> [1] "xx"
```

We can see what's going on with sloop::s3_dispatch():

```
s3_dispatch(x[1])
#> => [.secret
#>    [.default
#> -> [ (internal)
```

The => indicates that [.secret is called, but that NextMethod() delegates work to the underlying internal [method, as shown by the ->.

As with UseMethod(), the precise semantics of NextMethod() are complex. In particular, it tracks the list of potential next methods with a special variable, which means that modifying the object that's being dispatched upon will have no impact on which method gets called next.

Allowing subclassing

When you create a class, you need to decide if you want to allow subclasses, because it requires some changes to the constructor and careful thought in your methods.

To allow subclasses, the parent constructor needs to have ... and class arguments:

```
new_secret <- function(x, ..., class = character()) {
  stopifnot(is.double(x))

  structure(
    x,
    ...,
    class = c(class, "secret")
  )
}
```

Then the subclass constructor can just call to the parent class constructor with additional arguments as needed. For example, imagine we want to create a supersecret class which also hides the number of characters:

```
new_supersecret <- function(x) {
  new_secret(x, class = "supersecret")
}

print.supersecret <- function(x, ...) {
  print(rep("xxxxx", length(x)))
  invisible(x)
}

x2 <- new_supersecret(c(15, 1, 456))
x2
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

To allow inheritance, you also need to think carefully about your methods, as you can no longer use the constructor. If you do, the method will always return the same class, regardless of the input. This forces whoever makes a subclass to do a lot of extra work.

Concretely, this means we need to revise the [.secret method. Currently it always returns a secret(), even when given a supersecret:

```
`[.secret` <- function(x, ...) {
  new_secret(NextMethod())
}

x2[1:3]
#> [1] "xx"  "x"   "xxx"
```

We want to make sure that [.secret returns the same class as x even if it's a subclass. As far as I can tell, there is no way to solve this problem using base R alone. Instead, you'll need to use the vctrs package, which provides a solution in the form of the vctrs::vec_restore() generic. This generic takes two inputs: an object which has lost subclass information, and a template object to use for restoration.

Typically vec_restore() methods are quite simple: you just call the constructor with appropriate arguments:

```
vec_restore.secret <- function(x, to, ...) new_secret(x)
vec_restore.supersecret <- function(x, to, ...) new_supersecret(x)
```

(If your class has attributes, you'll need to pass them from to into the constructor.)

Now we can use vec_restore() in the [.secret method:

```
`[.secret` <- function(x, ...) {
  vctrs::vec_restore(NextMethod(), x)
}
x2[1:3]
#> [1] "xxxxx" "xxxxx" "xxxxx"
```

(I only fully understood this issue quite recently, so at time of writing it is not used in the tidyverse. Hopefully by the time you're reading this, it will have rolled out, making it much easier to (e.g.) subclass tibbles.)

If you build your class using the tools provided by the vctrs package, [will gain this behaviour automatically. You will only need to provide your own [method if you use attributes that depend on the data or want non-standard subsetting behaviour. See ?vctrs::new_vctr for details.

S3 and base types

What happens when you call an S3 generic with a base object, i.e. an object with no class? You might think it would dispatch on what class() returns:

```
class(matrix(1:5))
#> [1] "matrix"
```

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

The string "array" or "matrix" if the object has dimensions

The result of typeof() with a few minor tweaks

The string "numeric" if object is "integer" or "double"

There is no base function that will compute the implicit class, but you can use sloop::s3_class()

```
s3_class(matrix(1:5))
#> [1] "matrix"  "integer" "numeric"
```

This is used by s3_dispatch():

```
s3_dispatch(print(matrix(1:5)))
#>    print.matrix
#>    print.integer
#>    print.numeric
#> => print.default
```

This means that the class() of an object does not uniquely determine its dispatch:

```
x1 <- 1:5
class(x1)
#> [1] "integer"
s3_dispatch(mean(x1))
#>    mean.integer
#>    mean.numeric
#> => mean.default

x2 <- structure(x1, class = "integer")
class(x2)
#> [1] "integer"
s3_dispatch(mean(x2))
#>    mean.integer
#> => mean.default
```

Internal generics

Some base functions, like [, sum(), and cbind(), are called internal generics because they don't call UseMethod() but instead call the C functions DispatchGroup() or DispatchOrEval(). s3_dispatch() shows internal generics by including the name of the generic followed by (internal):

```
s3_dispatch(Sys.time()[1])
#> => [.POSIXct
#>    [.POSIXt
#>    [.default
#> -> [ (internal)
```

For performance reasons, internal generics do not dispatch to methods unless the class attribute has been set, which means that internal generics do not use the implicit class. Again, if you're ever confused about method dispatch, you can rely on s3_dispatch().

Group generics

Group generics are the most complicated part of S3 method dispatch because they involve both NextMethod() and internal generics. Like internal generics, they only exist in base R, and you cannot define your own group generic.

There are four group generics:

Math: abs(), sign(), sqrt(), floor(), cos(), sin(), log(), and more (see ?Math for the complete list).

Ops: +, -, *, /, ^, %%, %/%, &, |, !, ==, !=, <, <=, >=, and >.

Summary: all(), any(), sum(), prod(), min(), max(), and range().

Complex: Arg(), Conj(), Im(), Mod(), Re().

Defining a single group generic for your class overrides the default behaviour for all of the members of the group. Methods for group generics are looked for only if the methods for the specific generic do not exist:

```
s3_dispatch(sum(Sys.time()))
#>    sum.POSIXct
#>    sum.POSIXt
#>    sum.default
#> => Summary.POSIXct
#>    Summary.POSIXt
#>    Summary.default
#> -> sum (internal)
```

Most group generics involve a call to NextMethod(). For example, take difftime() objects. If you look at the method dispatch for abs(), you'll see there's a Math group generic defined.

```
y <- as.difftime(10, units = "mins")
s3_dispatch(abs(y))
#>    abs.difftime
#>    abs.default
#> => Math.difftime
#>    Math.default
#> -> abs (internal)
```

Math.difftime basically looks like this:

```
Math.difftime <- function(x, ...) {
  new_difftime(NextMethod(), units = attr(x, "units"))
}
```

It dispatches to the next method, here the internal default, to perform the actual computation, then restore the class and attributes. (To better support subclasses of difftime this would need to call vec_restore())

Inside a group generic function a special variable .Generic provides the actual generic function called. This can be useful when producing error messages, and can sometimes be useful if you need to manually re-call the generic with different arguments.

Double dispatch

Generics in the Ops group, which includes the two-argument arithmetic and Boolean operators like - and &, implement a special type of method dispatch. They dispatch on the type of both of the arguments, which is called double dispatch. This is necessary to preserve the commutative property of many operators, i.e. a + b should equal b + a. Take the following simple example:

```
date <- as.Date("2017-01-01")
integer <- 1L

date + integer
#> [1] "2017-01-02"
integer + date
#> [1] "2017-01-02"
```

If + dispatched only on the first argument, it would return different values for the two cases. To overcome this problem, generics in the Ops group use a slightly different strategy from usual. Rather than doing a single method dispatch, they do two, one for each input. There are three possible outcomes of this lookup:

The methods are the same, so it doesn't matter which method is used.

The methods are different, and R falls back to the internal method with a warning.

One method is internal, in which case R calls the other method.

This approach is error prone so if you want to implement robust double dispatch for algebraic operators, I recommend using the vctrs package. See ?vctrs::vec_arith for details.