Package: S7 (via r-universe)

November 2, 2025

Title An Object Oriented System Meant to Become a Successor to S3 and S4

Version 0.2.0

Description A new object oriented programming system designed to be a successor to S3 and S4. It includes formal class, generic, and method specification, and a limited form of multiple dispatch. It has been designed and implemented collaboratively by the R Consortium Object-Oriented Programming Working Group, which includes representatives from R-Core, 'Bioconductor', 'Posit'/'tidyverse', and the wider R community.

License MIT + file LICENSE

URL https://rconsortium.github.io/S7/, https://github.com/RConsortium/S7

BugReports https://github.com/RConsortium/S7/issues

Depends R (>= 3.5.0)

Imports utils

Suggests bench, callr, covr, knitr, methods, rmarkdown, testthat (>= 3.2.0), tibble

VignetteBuilder knitr

Config/build/compilation-database true

Config/Needs/website sloop

Config/testthat/edition 3

Config/testthat/parallel TRUE

Config/testthat/start-first external-generic

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository https://r-multiverse.r-universe.dev

Date/Publication 2024-11-06 17:03:05 UTC

RemoteUrl https://github.com/RConsortium/S7

RemoteRef v0.2.0

RemoteSha 860394b610799aac671ffbdc13ac69e9d1f31ccc

Contents

`base_classes` *S7 wrappers for base types*

Description

The following S7 classes represent base types allowing them to be used within S7:

- `class_logical`
- `class_integer`
- `class_double`
- `class_complex`
- `class_character`

- `class_raw`
- `class_list`
- `class_expression`
- `class_name`
- `class_call`
- `class_function`
- `class_environment` (can only be used for properties)

We also include three union types to model numerics, atomics, and vectors respectively:

- `class_numeric` is a union of `class_integer` and `class_double`.
- `class_atomic` is a union of `class_logical`, `class_numeric`, `class_complex`, `class_character`, and `class_raw`.
- `class_vector` is a union of `class_atomic`, `class_list`, and `class_expression`.
- `class_language` is a union of `class_name` and `class_call`.

Usage

```
class_logical

class_integer

class_double

class_complex

class_character

class_raw

class_list

class_expression

class_name

class_call

class_function

class_environment

class_numeric

class_atomic

class_vector
```

```
class_language
```

Value

S7 classes wrapping around common base types and S3 classes.

Examples

```
class_integer
class_numeric
class_factor
```

`base_s3_classes` *S7 wrappers for key S3 classes*

Description

S7 bundles S3 definitions for key S3 classes provided by the base packages:

- `class_data.frame` for data frames.
- `class_Date` for dates.
- `class_factor` for factors.
- `class_POSIXct`, `class_POSIXlt` and `class_POSIXt` for date-times.
- `class_formula` for formulas.

Usage

```
class_factor

class_Date

class_POSIXct

class_POSIXlt

class_POSIXt

class_data.frame

class_formula
```

class_any *Dispatch on any class*

Description

Use `class_any` to register a default method that is called when no other methods are matched.

Usage

```
class_any
```

Examples

```
foo <- new_generic("foo", "x")
method(foo, class_numeric) <- function(x) "number"
method(foo, class_any) <- function(x) "fallback"

foo(1)
foo("x")
```

class_missing *Dispatch on a missing argument*

Description

Use `class_missing` to dispatch when the user has not supplied an argument, i.e. it's missing in the sense of `missing()`, not in the sense of `is.na()`.

Usage

```
class_missing
```

Value

Sentinel objects used for special types of dispatch.

Examples

```
foo <- new_generic("foo", "x")
method(foo, class_numeric) <- function(x) "number"
method(foo, class_missing) <- function(x) "missing"
method(foo, class_any) <- function(x) "fallback"

foo(1)
foo()
foo("")
```

`convert` *Convert an object from one type to another*

Description

`convert(from, to)` is a built-in generic for converting an object from one type to another. It is special in three ways:

- It uses double-dispatch, because conversion depends on both `from` and `to`.
- It uses non-standard dispatch because `to` is a class, not an object.
- It doesn't use inheritance for the `to` argument. To understand why, imagine you have written methods to objects of various types to `classParent`. If you then create a new `classChild` that inherits from `classParent`, you can't expect the methods written for `classParent` to work because those methods will return `classParent` objects, not `classChild` objects.

`convert()` provides two default implementations:

1. When `from` inherits from `to`, it strips any properties that `from` possesses that `to` does not (downcasting).
2. When `to` is a subclass of `from`'s class, it creates a new object of class `to`, copying over existing properties from `from` and initializing new properties of `to` (upcasting).

If you are converting an object solely for the purposes of accessing a method on a superclass, you probably want `super()` instead. See its docs for more details.

S3 & S4:

`convert()` plays a similar role to the convention of defining `as.foo()` functions/generics in S3, and to `as()`/`setAs()` in S4.

Usage

```
convert(from, to, ...)
```

Arguments

`from`	An S7 object to convert.
`to`	An S7 class specification, passed to `as_class()`.
`...`	Other arguments passed to custom `convert()` methods. For upcasting, these can be used to override existing properties or set new ones.

Value

Either `from` coerced to class `to`, or an error if the coercion is not possible.

Examples

```
Foo1 <- new_class("Foo1", properties = list(x = class_integer))
Foo2 <- new_class("Foo2", Foo1, properties = list(y = class_double))

# Downcasting: S7 provides a default implementation for coercing an object
# to one of its parent classes:
convert(Foo2(x = 1L, y = 2), to = Foo1)

# Upcasting: S7 also provides a default implementation for coercing an object
# to one of its child classes:
convert(Foo1(x = 1L), to = Foo2)
convert(Foo1(x = 1L), to = Foo2, y = 2.5)  # Set new property
convert(Foo1(x = 1L), to = Foo2, x = 2L, y = 2.5)  # Override existing and set new

# For all other cases, you'll need to provide your own.
try(convert(Foo1(x = 1L), to = class_integer))

method(convert, list(Foo1, class_integer)) <- function(from, to) {
  from@x
}
convert(Foo1(x = 1L), to = class_integer)

# Note that conversion does not respect inheritance so if we define a
# convert method for integer to foo1
method(convert, list(class_integer, Foo1)) <- function(from, to) {
  Foo1(x = from)
}
convert(1L, to = Foo1)

# Converting to Foo2 will still error
try(convert(1L, to = Foo2))
# This is probably not surprising because foo2 also needs some value
# for `@y`, but it definitely makes dispatch for convert() special
```

`method` *Find a method for an S7 generic*

Description

`method()` takes a generic and class signature and performs method dispatch to find the corresponding method implementation. This is rarely needed because you'll usually rely on the the generic to do dispatch for you (via `S7_dispatch()`). However, this introspection is useful if you want to see the implementation of a specific method.

Usage

```
method(generic, class = NULL, object = NULL)
```

Arguments

generic	A generic function, i.e. an S7 generic, an external generic, an S3 generic, or an S4 generic.
class, object	Perform introspection either with a class (processed with as_class()) or a concrete object. If generic uses multiple dispatch then both object and class must be a list of classes/objects.

Value

Either a function with class S7_method or an error if no matching method is found.

See Also

method_explain() to explain why a specific method was picked.

Examples

```
# Create a generic and register some methods
bizarro <- new_generic("bizarro", "x")
method(bizarro, class_numeric) <- function(x) rev(x)
method(bizarro, class_factor) <- function(x) {
  levels(x) <- rev(levels(x))
  x
}

# Printing the generic shows the registered method
bizarro

# And you can use method() to inspect specific implementations
method(bizarro, class = class_integer)
method(bizarro, object = 1)
method(bizarro, class = class_factor)

# errors if method not found
try(method(bizarro, class = class_data.frame))
try(method(bizarro, object = "x"))
```

---

method<- *Register an S7 method for a generic*

Description

A generic defines the interface of a function. Once you have created a generic with new_generic(), you provide implementations for specific signatures by registering methods with method<-.

The goal is for method<- to be the single function you need when working with S7 generics or S7 classes. This means that as well as registering methods for S7 classes on S7 generics, you can also register methods for S7 classes on S3 or S4 generics, and S3 or S4 classes on S7 generics. But this

is not a general method registration function: at least one of `generic` and `signature` needs to be from S7.

Note that if you are writing a package, you must call `methods_register()` in your `.onLoad`. This ensures that all methods are dynamically registered when needed.

Usage

```
method(generic, signature) <- value
```

Arguments

`generic`	A generic function, i.e. an S7 generic, an external generic, an S3 generic, or an S4 generic.
`signature`	A method signature. For S7 generics that use single dispatch, this must be one of the following: • An S7 class (created by `new_class()`). • An S7 union (created by `new_union()`). • An S3 class (created by `new_S3_class()`). • An S4 class (created by `methods::getClass()` or `methods::new()`). • A base type like class_logical, class_integer, or class_numeric. • A special type like class_missing or class_any. For S7 generics that use multiple dispatch, this must be a list of any of the above types. For S3 generics, this must be a single S7 class. For S4 generics, this must either be an S7 class, or a list that includes at least one S7 class.
`value`	A function that implements the generic specification for the given `signature`.

Value

The `generic`, invisibly.

Examples

```
# Create a generic
bizarro <- new_generic("bizarro", "x")
# Register some methods
method(bizarro, class_numeric) <- function(x) rev(x)
method(bizarro, new_S3_class("data.frame")) <- function(x) {
  x[] <- lapply(x, bizarro)
  rev(x)
}

# Using a generic calls the methods automatically
bizarro(head(mtcars))
```

methods_register *Register methods in a package*

Description

When using S7 in a package you should always call `methods_register()` when your package is loaded. This ensures that methods are registered as needed when you implement methods for generics (S3, S4, and S7) in other packages. (This is not strictly necessary if you only register methods for generics in your package, but it's better to include it and not need it than forget to include it and hit weird errors.)

Usage

```
methods_register()
```

Value

Nothing; called for its side-effects.

Examples

```
.onLoad <- function(...) {
  S7::methods_register()
}
```

method_explain *Explain method dispatch*

Description

`method_explain()` shows all possible methods that a call to a generic might use, which ones exist, and which one will actually be called.

Note that method dispatch uses a string representation of each class in the class hierarchy. Each class system uses a slightly different convention to avoid ambiguity.

- S7: `pkg::class` or `class`
- S4: `S4/pkg::class` or `S4/class`
- S3: `class`

Usage

```
method_explain(generic, class = NULL, object = NULL)
```

Arguments

`generic`	A generic function, i.e. an S7 generic, an external generic, an S3 generic, or an S4 generic.
`class, object`	Perform introspection either with a `class` (processed with `as_class()`) or a concrete `object`. If `generic` uses multiple dispatch then both `object` and `class` must be a list of classes/objects.

Value

Nothing; this function is called for it's side effects.

Examples

```
Foo1 <- new_class("Foo1")
Foo2 <- new_class("Foo2", Foo1)

add <- new_generic("add", c("x", "y"))
method(add, list(Foo2, Foo1)) <- function(x, y) c(2, 1)
method(add, list(Foo1, Foo1)) <- function(x, y) c(1, 1)

method_explain(add, list(Foo2, Foo2))
```

`new_class` *Define a new S7 class*

Description

A class specifies the properties (data) that each of its objects will possess. The class, and its parent, determines which method will be used when an object is passed to a generic.

Learn more in `vignette("classes-objects")`

Usage

```
new_class(
  name,
  parent = S7_object,
  package = topNamespaceName(parent.frame()),
  properties = list(),
  abstract = FALSE,
  constructor = NULL,
  validator = NULL
)

new_object(.parent, ...)
```

Arguments

`name`	The name of the class, as a string. The result of calling `new_class()` should always be assigned to a variable with this name, i.e. `Foo <- new_class("Foo")`.
`parent`	The parent class to inherit behavior from. There are three options: • An S7 class, like S7_object. • An S3 class wrapped by `new_S3_class()`. • A base type, like class_logical, class_integer, etc.
`package`	Package name. This is automatically resolved if the class is defined in a package, and `NULL` otherwise. Note, if the class is intended for external use, the constructor should be exported. Learn more in `vignette("packages")`.
`properties`	A named list specifying the properties (data) that belong to each instance of the class. Each element of the list can either be a type specification (processed by `as_class()`) or a full property specification created `new_property()`.
`abstract`	Is this an abstract class? An abstract class can not be instantiated.
`constructor`	The constructor function. In most cases, you can rely on the default constructor, which will generate a function with one argument for each property. A custom constructor should call `new_object()` to create the S7 object. The first argument, `.data`, should be an instance of the parent class (if used). The subsequent arguments are used to set the properties.
`validator`	A function taking a single argument, `self`, the object to validate. The job of a validator is to determine whether the object is valid, i.e. if the current property values form an allowed combination. The types of the properties are always automatically validated so the job of the validator is to verify that the *values* of individual properties are ok (i.e. maybe a property should have length 1, or should always be positive), or that the *combination* of values of multiple properties is ok. It is called after construction and whenever any property is set. The validator should return `NULL` if the object is valid. If not, it should return a character vector where each element describes a single problem, using `@prop_name` to describe where the problem lies. See `validate()` for more details, examples, and how to temporarily suppress validation when needed.
`.parent, ...`	Parent object and named properties used to construct the object.

Value

A object constructor, a function that can be used to create objects of the given class.

Examples

```
# Create an class that represents a range using a numeric start and end
Range <- new_class("Range",
  properties = list(
    start = class_numeric,
    end = class_numeric
```

```
  )
)
r <- Range(start = 10, end = 20)
r
# get and set properties with @
r@start
r@end <- 40
r@end

# S7 automatically ensures that properties are of the declared types:
try(Range(start = "hello", end = 20))

# But we might also want to use a validator to ensure that start and end
# are length 1, and that start is < end
Range <- new_class("Range",
  properties = list(
    start = class_numeric,
    end = class_numeric
  ),
  validator = function(self) {
    if (length(self@start) != 1) {
      "@start must be a single number"
    } else if (length(self@end) != 1) {
      "@end must be a single number"
    } else if (self@end < self@start) {
      "@end must be great than or equal to @start"
    }
  }
)
try(Range(start = c(10, 15), end = 20))
try(Range(start = 20, end = 10))

r <- Range(start = 10, end = 20)
try(r@start <- 25)
```

`new_external_generic` *Generics in other packages*

Description

You need an explicit external generic when you want to provide methods for a generic (S3, S4, or S7) that is defined in another package, and you don't want to take a hard dependency on that package.

The easiest way to provide methods for generics in other packages is import the generic into your `NAMESPACE`. This, however, creates a hard dependency, and sometimes you want a soft dependency, only registering the method if the package is already installed. `new_external_generic()` allows you to provide the minimal needed information about a generic so that methods can be registered at run time, as needed, using `methods_register()`.

Note that in tests, you'll need to explicitly call the generic from the external package with `pkg::generic()`.

Usage

```
new_external_generic(package, name, dispatch_args, version = NULL)
```

Arguments

package	Package the generic is defined in.
name	Name of generic, as a string.
dispatch_args	Character vector giving arguments used for dispatch.
version	An optional version the package must meet for the method to be registered.

Value

An S7 external generic, i.e. a list with class `S7_external_generic`.

Examples

```
MyClass <- new_class("MyClass")

your_generic <- new_external_generic("stats", "median", "x")
method(your_generic, MyClass) <- function(x) "Hi!"
```

---

new_generic *Define a new generic*

Description

A generic function uses different implementations (*methods*) depending on the class of one or more arguments (the *signature*). Create a new generic with `new_generic()` then use method<- to add methods to it.

Method dispatch is performed by `S7_dispatch()`, which must always be included in the body of the generic, but in most cases `new_generic()` will generate this for you.

Learn more in `vignette("generics-methods")`

Usage

```
new_generic(name, dispatch_args, fun = NULL)

S7_dispatch()
```

Arguments

name	The name of the generic. This should be the same as the object that you assign it to.
dispatch_args	A character vector giving the names of one or more arguments used to find the method.

`fun` An optional specification of the generic, which must call `S7_dispatch()` to dispatch to methods. This is usually generated automatically from the `dispatch_args`, but you may want to supply it if you want to add additional required arguments, omit `...`, or perform some standardised computation in the generic.

The `dispatch_args` must be the first arguments to `fun`, and, if present, `...` must immediately follow them.

Value

An S7 generic, i.e. a function with class `S7_generic`.

Dispatch arguments

The arguments that are used to pick the method are called the **dispatch arguments**. In most cases, this will be one argument, in which case the generic is said to use **single dispatch**. If it consists of more than one argument, it's said to use **multiple dispatch**.

There are two restrictions on the dispatch arguments: they must be the first arguments to the generic and if the generic uses `...`, it must occur immediately after the dispatch arguments.

See Also

`new_external_generic()` to define a method for a generic in another package without taking a strong dependency on it.

Examples

```
# A simple generic with methods for some base types and S3 classes
type_of <- new_generic("type_of", dispatch_args = "x")
method(type_of, class_character) <- function(x, ...) "A character vector"
method(type_of, new_S3_class("data.frame")) <- function(x, ...) "A data frame"
method(type_of, class_function) <- function(x, ...) "A function"

type_of(mtcars)
type_of(letters)
type_of(mean)

# If you want to require that methods implement additional arguments,
# you can use a custom function:
mean2 <- new_generic("mean2", "x", function(x, ..., na.rm = FALSE) {
   S7_dispatch()
})

method(mean2, class_numeric) <- function(x, ..., na.rm = FALSE) {
  if (na.rm) {
    x <- x[!is.na(x)]
  }
  sum(x) / length(x)
}

# You'll be warned if you forget the argument:
method(mean2, class_character) <- function(x, ...) {
```

```
  stop("Not supported")
}
```

`new_property` *Define a new property*

Description

A property defines a named component of an object. Properties are typically used to store (meta) data about an object, and are often limited to a data of a specific `class`.

By specifying a `getter` and/or `setter`, you can make the property "dynamic" so that it's computed when accessed or has some non-standard behaviour when modified. Dynamic properties are not included as an argument to the default class constructor.

See the "Properties: Common Patterns" section in `vignette("class-objects")` for more examples.

Usage

```
new_property(
  class = class_any,
  getter = NULL,
  setter = NULL,
  validator = NULL,
  default = NULL,
  name = NULL
)
```

Arguments

`class`	Class that the property must be an instance of. See `as_class()` for details.
`getter`	An optional function used to get the value. The function should take `self` as its sole argument and return the value. If you supply a `getter`, you are responsible for ensuring that it returns an object of the correct `class`; it will not be validated automatically. If a property has a getter but doesn't have a setter, it is read only.
`setter`	An optional function used to set the value. The function should take `self` and `value` and return a modified object.
`validator`	A function taking a single argument, `value`, the value to validate. The job of a validator is to determine whether the property value is valid. It should return `NULL` if the object is valid, or if it's not valid, a single string describing the problem. The message should not include the name of the property as this will be automatically appended to the beginning of the message. The validator will be called after the `class` has been verified, so your code can assume that `value` has known type.

`default`	When an object is created and the property is not supplied, what should it default to? If `NULL`, it defaults to the "empty" instance of `class`. This can also be a quoted call, which then becomes a standard function promise in the default constructor, evaluated at the time the object is constructed.
`name`	Property name, primarily used for error messages. Generally don't need to set this here, as it's more convenient to supply as a the element name when defining a list of properties. If both `name` and a list-name are supplied, the list-name will be used.

Value

An S7 property, i.e. a list with class `S7_property`.

Examples

```
# Simple properties store data inside an object
Pizza <- new_class("Pizza", properties = list(
  slices = new_property(class_numeric, default = 10)
))
my_pizza <- Pizza(slices = 6)
my_pizza@slices
my_pizza@slices <- 5
my_pizza@slices

your_pizza <- Pizza()
your_pizza@slices

# Dynamic properties can compute on demand
Clock <- new_class("Clock", properties = list(
  now = new_property(getter = function(self) Sys.time())
))
my_clock <- Clock()
my_clock@now; Sys.sleep(1)
my_clock@now
# This property is read only, because there is a 'getter' but not a 'setter'
try(my_clock@now <- 10)

# Because the property is dynamic, it is not included as an
# argument to the default constructor
try(Clock(now = 10))
args(Clock)
```

`new_S3_class` *Declare an S3 class*

Description

To use an S3 class with S7, you must explicitly declare it using `new_S3_class()` because S3 lacks a formal class definition. (Unless it's an important base class already defined in base_s3_classes.)

Usage

```
new_S3_class(class, constructor = NULL, validator = NULL)
```

Arguments

`class`	S3 class vector (i.e. what `class()` returns). For method registration, you can abbreviate this to a single string, the S3 class name.
`constructor`	An optional constructor that can be used to create objects of the specified class. This is only needed if you wish to have an S7 class inherit from an S3 class or to use the S3 class as a property without a default. It must be specified in the same way as a S7 constructor: the first argument should be `.data` (the base type whose attributes will be modified). All arguments to the constructor should have default values so that when the constructor is called with no arguments, it returns returns an "empty", but valid, object.
`validator`	An optional validator used by `validate()` to check that the S7 object adheres to the constraints of the S3 class. A validator is a single argument function that takes the object to validate and returns `NULL` if the object is valid. If the object is invalid, it returns a character vector of problems.

Value

An S7 definition of an S3 class, i.e. a list with class `S7_S3_class`.

Method dispatch, properties, and unions

There are three ways of using S3 with S7 that only require the S3 class vector:

- Registering a S3 method for an S7 generic.
- Restricting an S7 property to an S3 class.
- Using an S3 class in an S7 union.

This is easy, and you can usually include the `new_S3_class()` call inline:

```
method(my_generic, new_S3_class("factor")) <- function(x) "A factor"
new_class("MyClass", properties = list(types = new_S3_class("factor")))
new_union("character", new_S3_class("factor"))
```

Extending an S3 class

Creating an S7 class that extends an S3 class requires more work. You'll also need to provide a constructor for the S3 class that follows S7 conventions. This means the first argument to the constructor should be `.data`, and it should be followed by one argument for each attribute used by the class.

This can be awkward because base S3 classes are usually heavily wrapped for user convenience and no low level constructor is available. For example, the factor class is an integer vector with a

character vector of `levels`, but there's no base R function that takes an integer vector of values and character vector of levels, verifies that they are consistent, then creates a factor object.

You may optionally want to also provide a `validator` function which will ensure that `validate()` confirms the validity of any S7 classes that build on this class. Unlike an S7 validator, you are responsible for validating the types of the attributes.

The following code shows how you might wrap the base Date class. A Date is a numeric vector with class `Date` that can be constructed with `.Date()`.

```
S3_Date <- new_S3_class("Date",
  function(.data = integer()) {
    .Date(.data)
  },
  function(self) {
    if (!is.numeric(self)) {
      "Underlying data must be numeric"
    }
  }
)
```

Examples

```
# No checking, just used for dispatch
Date <- new_S3_class("Date")

my_generic <- new_generic("my_generic", "x")
method(my_generic, Date) <- function(x) "This is a date"

my_generic(Sys.Date())
```

`new_union` *Define a class union*

Description

A class union represents a list of possible classes. You can create it with `new_union(a, b, c)` or `a | b | c`. Unions can be used in two places:

- To allow a property to be one of a set of classes, `new_property(class_integer | Range)`. The default `default` value for the property will be the constructor of the first object in the union. This means if you want to create an "optional" property (i.e. one that can be `NULL` or of a specified type), you'll need to write (e.g.) `NULL | class_integer`.
- As a convenient short-hand to define methods for multiple classes. `method(foo, X | Y) <- f` is short-hand for `method(foo, X) <- f; method(foo, Y) <- foo`

S7 includes built-in unions for "numeric" (integer and double vectors), "atomic" (logical, numeric, complex, character, and raw vectors) and "vector" (atomic vectors, lists, and expressions).

Usage

```
new_union(...)
```

Arguments

...	The classes to include in the union. See as_class() for details.

Value

An S7 union, i.e. a list with class S7_union.

Examples

```
logical_or_character <- new_union(class_logical, class_character)
logical_or_character
# or with shortcut syntax
logical_or_character <- class_logical | class_character

Foo <- new_class("Foo", properties = list(x = logical_or_character))
Foo(x = TRUE)
Foo(x = letters[1:5])
try(Foo(1:3))

bar <- new_generic("bar", "x")
# Use built-in union
method(bar, class_atomic) <- function(x) "Hi!"
bar
bar(TRUE)
bar(letters)
try(bar(NULL))
```

---

prop *Get/set a property*

Description

- prop(x, "name") / prop@name get the value of the a property, erroring if it the property doesn't exist.
- prop(x, "name") <- value / prop@name <- value set the value of a property.

Usage

```
prop(object, name)

prop(object, name, check = TRUE) <- value

object@name
```

Arguments

`object`	An object from a S7 class
`name`	The name of the parameter as a character. Partial matching is not performed.
`check`	If `TRUE`, check that `value` is of the correct type and run `validate()` on the object before returning.
`value`	A new value for the property. The object is automatically checked for validity after the replacement is done.

Value

`prop()` and `@` return the value of the property. `prop<-()` and `@<-` are called for their side-effects and return the modified object, invisibly.

Examples

```
Horse <- new_class("Horse", properties = list(
  name = class_character,
  colour = class_character,
  height = class_numeric
))
lexington <- Horse(colour = "bay", height = 15, name = "Lex")
lexington@colour
prop(lexington, "colour")

lexington@height <- 14
prop(lexington, "height") <- 15
```

`props` *Get/set multiple properties*

Description

- `props(x)` returns all properties.
- `props(x) <- list(name1 = val1, name2 = val2)` modifies an existing object by setting multiple properties simultaneously.
- `set_props(x, name1 = val1, name2 = val2)` creates a copy of an existing object with new values for the specified properties.

Usage

```
props(object, names = prop_names(object))

props(object) <- value

set_props(object, ...)
```

Arguments

object	An object from a S7 class
names	A character vector of property names to retrieve. Default is all properties.
value	A named list of values. The object is checked for validity only after all replacements are performed.
...	Name-value pairs given property to modify and new value.

Value

A named list of property values.

Examples

```
Horse <- new_class("Horse", properties = list(
  name = class_character,
  colour = class_character,
  height = class_numeric
))
lexington <- Horse(colour = "bay", height = 15, name = "Lex")

props(lexington)
props(lexington) <- list(height = 14, name = "Lexington")
lexington
```

---

prop_names *Property introspection*

Description

- `prop_names(x)` returns the names of the properties
- `prop_exists(x, "prop")` returns `TRUE` iif `x` has property `prop`.

Usage

```
prop_names(object)

prop_exists(object, name)
```

Arguments

object	An object from a S7 class
name	The name of the parameter as a character. Partial matching is not performed.

Value

`prop_names()` returns a character vector; `prop_exists()` returns a single `TRUE` or `FALSE`.

Examples

```
Foo <- new_class("Foo", properties = list(a = class_character, b = class_integer))
f <- Foo()

prop_names(f)
prop_exists(f, "a")
prop_exists(f, "c")
```

`S4_register` *Register an S7 class with S4*

Description

If you want to use method<- to register an method for an S4 generic with an S7 class, you need to call `S4_register()` once.

Usage

```
S4_register(class, env = parent.frame())
```

Arguments

`class`	An S7 class created with `new_class()`.
`env`	Expert use only. Environment where S4 class will be registered.

Value

Nothing; the function is called for its side-effect.

Examples

```
methods::setGeneric("S4_generic", function(x) {
  standardGeneric("S4_generic")
})

Foo <- new_class("Foo")
S4_register(Foo)
method(S4_generic, Foo) <- function(x) "Hello"

S4_generic(Foo())
```

S7_class *Retrieve the S7 class of an object*

Description

Given an S7 object, find it's class.

Usage

```
S7_class(object)
```

Arguments

object	The S7 object

Value

An S7 class.

Examples

```
Foo <- new_class("Foo")
S7_class(Foo())
```

S7_data *Get/set underlying "base" data*

Description

When an S7 class inherits from an existing base type, it can be useful to work with the underlying object, i.e. the S7 object stripped of class and properties.

Usage

```
S7_data(object)

S7_data(object, check = TRUE) <- value
```

Arguments

object	An object from a S7 class
check	If TRUE, check that value is of the correct type and run validate() on the object before returning.
value	Object used to replace the underlying data.

Value

`S7_data()` returns the data stored in the base object; `S7_data<-()` is called for its side-effects and returns `object` invisibly.

Examples

```
Text <- new_class("Text", parent = class_character)
y <- Text(c(foo = "bar"))
y
S7_data(y)

S7_data(y) <- c("a", "b")
y
```

S7_inherits *Does this object inherit from an S7 class?*

Description

- `S7_inherits()` returns `TRUE` or `FALSE`.
- `check_is_S7()` throws an error if `x` isn't the specified `class`.

Usage

```
S7_inherits(x, class = NULL)

check_is_S7(x, class = NULL, arg = deparse(substitute(x)))
```

Arguments

`x`	An object
`class`	An S7 class or `NULL`. If `NULL`, tests whether `x` is an S7 object without testing for a specific class.
`arg`	Argument name used in error message.

Value

- `S7_inherits()` returns a single `TRUE` or `FALSE`.
- `check_is_S7()` returns nothing; it's called for its side-effects.

Note

Starting with R 4.3.0, `base::inherits()` can accept an S7 class as the second argument, supporting usage like `inherits(x, Foo)`.

Examples

```
Foo1 <- new_class("Foo1")
Foo2 <- new_class("Foo2")

S7_inherits(Foo1(), Foo1)
check_is_S7(Foo1())
check_is_S7(Foo1(), Foo1)

S7_inherits(Foo1(), Foo2)
try(check_is_S7(Foo1(), Foo2))

if (getRversion() >= "4.3.0")
  inherits(Foo1(), Foo1)
```

`super` *Force method dispatch to use a superclass*

Description

`super(from, to)` causes the dispatch for the next generic to use the method for the superclass `to` instead of the actual class of `from`. It's needed when you want to implement a method in terms of the implementation of its superclass.

S3 & S4:

`super()` performs a similar role to `NextMethod()` in S3 or `methods::callNextMethod()` in S4, but is much more explicit:

- The super class that `super()` will use is known when write `super()` (i.e. statically) as opposed to when the generic is called (i.e. dynamically).
- All arguments to the generic are explicit; they are not automatically passed along.

This makes `super()` more verbose, but substantially easier to understand and reason about.

`super()` in S3 generics:

Note that you can't use `super()` in methods for an S3 generic. For example, imagine that you have made a subclass of "integer":

```
MyInt <- new_class("MyInt", parent = class_integer, package = NULL)
```

Now you go to write a custom print method:

```
method(print, MyInt) <- function(x, ...) {
   cat("<MyInt>")
   print(super(x, to = class_integer))
}

MyInt(10L)
#> <MyInt>super(<MyInt>, <integer>)
```

This doesn't work because `print()` isn't an S7 generic so doesn't understand how to interpret the special object that `super()` produces. While you could resolve this problem with `NextMethod()` (because S7 is implemented on top of S3), we instead recommend using `S7_data()` to extract the underlying base object:

```
method(print, MyInt) <- function(x, ...) {
   cat("<MyInt>")
   print(S7_data(x))
}

MyInt(10L)
#> <MyInt>[1] 10
```

Usage

```
super(from, to)
```

Arguments

`from`	An S7 object to cast.
`to`	An S7 class specification, passed to `as_class()`. Must be a superclass of `object`.

Value

An `S7_super` object which should always be passed immediately to a generic. It has no other special behavior.

Examples

```
Foo1 <- new_class("Foo1", properties = list(x = class_numeric, y = class_numeric))
Foo2 <- new_class("Foo2", Foo1, properties = list(z = class_numeric))

total <- new_generic("total", "x")
method(total, Foo1) <- function(x) x@x + x@y

# This won't work because it'll be stuck in an infinite loop:
method(total, Foo2) <- function(x) total(x) + x@z

# We could write
method(total, Foo2) <- function(x) x@x + x@y + x@z
# but then we'd need to remember to update it if the implementation
# for total(<Foo1>) ever changed.

# So instead we use `super()` to call the method for the parent class:
method(total, Foo2) <- function(x) total(super(x, to = Foo1)) + x@z
total(Foo2(1, 2, 3))

# To see the difference between convert() and super() we need a
# method that calls another generic
```

```
bar1 <- new_generic("bar1", "x")
method(bar1, Foo1) <- function(x) 1
method(bar1, Foo2) <- function(x) 2

bar2 <- new_generic("bar2", "x")
method(bar2, Foo1) <- function(x) c(1, bar1(x))
method(bar2, Foo2) <- function(x) c(2, bar1(x))

obj <- Foo2(1, 2, 3)
bar2(obj)
# convert() affects every generic:
bar2(convert(obj, to = Foo1))
# super() only affects the _next_ call to a generic:
bar2(super(obj, to = Foo1))
```

`validate` *Validate an S7 object*

Description

`validate()` ensures that an S7 object is valid by calling the `validator` provided in `new_class()`. This is done automatically when constructing new objects and when modifying properties.

`valid_eventually()` disables validation, modifies the object, then revalidates. This is useful when a sequence of operations would otherwise lead an object to be temporarily invalid, or when repeated property modification causes a performance bottleneck because the validator is relatively expensive.

`valid_implicitly()` does the same but does not validate the object at the end. It should only be used rarely, and in performance critical code where you are certain a sequence of operations cannot produce an invalid object.

Usage

```
validate(object, recursive = TRUE, properties = TRUE)

valid_eventually(object, fun)

valid_implicitly(object, fun)
```

Arguments

`object`	An S7 object
`recursive`	If `TRUE`, calls validator of parent classes recursively.
`properties`	If `TRUE`, the default, checks property types before executing the validator.
`fun`	A function to call on the object before validation.

Value

Either `object` invisibly if valid, otherwise an error.

Examples

```
# A range class might validate that the start is less than the end
Range <- new_class("Range",
  properties = list(start = class_double, end = class_double),
  validator = function(self) {
    if (self@start >= self@end) "start must be smaller than end"
  }
)
# You can't construct an invalid object:
try(Range(1, 1))

# And you can't create an invalid object with @<-
r <- Range(1, 2)
try(r@end <- 1)

# But what if you want to move a range to the right?
rightwards <- function(r, x) {
  r@start <- r@start + x
  r@end <- r@end + x
  r
}
# This function doesn't work because it creates a temporarily invalid state
try(rightwards(r, 10))

# This is the perfect use case for valid_eventually():
rightwards <- function(r, x) {
  valid_eventually(r, function(object) {
    object@start <- object@start + x
    object@end <- object@end + x
    object
  })
}
rightwards(r, 10)

# Alternatively, you can set multiple properties at once using props<-,
# which validates once at the end
rightwards <- function(r, x) {
  props(r) <- list(start = r@start + x, end = r@end + x)
  r
}
rightwards(r, 20)
```

Index