

# Package: ooplah (via r-universe)

August 11, 2024

**Title** Helper Functions for Class Object-Oriented Programming

**Version** 0.2.0

**Description** Helper functions for coding object-oriented programming with a focus on R6. Includes functions for assertions and testing, looping, and re-usable design patterns including Abstract and Decorator classes.

**License** MIT + file LICENSE

**URL** <https://xoopR.github.io/ooplah/>, <https://github.com/xoopR/ooplah>

**BugReports** <https://github.com/xoopR/ooplah/issues>

**Imports** R6

**Suggests** devtools, testthat, withr

**Config/testthat/edition** 3

**Encoding** UTF-8

**LazyData** true

**Roxygen** list(markdown = TRUE, r6 = TRUE)

**RoxygenNote** 7.1.1

**SystemRequirements** C++11

**Repository** <https://raphaels1.r-universe.dev>

**RemoteUrl** <https://github.com/xoopR/ooplah>

**RemoteRef** HEAD

**RemoteSha** 4d88a65d349ba524060229704adcee25ec7bf4f8

## Contents

AbstractClass . . . . .	2
decorate . . . . .	3
DecoratorClass . . . . .	4
is.R6 . . . . .	6
is.R6Class . . . . .	6

is.R6Object . . . . .	7
loapply . . . . .	7
object_class . . . . .	8
ooplah . . . . .	9
private . . . . .	9
super . . . . .	9
vxapply . . . . .	10
<b>Index</b>	<b>12</b>

---

AbstractClass	<i>Create an abstract R6 Class</i>
---------------	------------------------------------

---

### Description

Creates an abstract R6 class by placing a thin wrapper around [R6::R6Class](#) which causes an error to be thrown if the class is directly constructed instead of one of its descendants.

### Details

An abstract class is a class that cannot be constructed directly. Instead they are used to define common fields/methods for child classes that inherit from them.

All arguments of [R6::R6Class](#) can be used as usual, see full details at [R6::R6Class](#).

### References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1996). Design Patterns: Elements of Reusable Software. Addison-Wesley Professional Computing Series (p. 395).

### Examples

```
library(R6)

ab <- AbstractClass("abstract", public = list(hello = "Hello World"))
## Not run:
# errors
ab$new()

## End(Not run)
child <- R6Class("child", inherit = ab)
child$new()$hello
```

---

decorate	<i>Sugar function for decoration</i>
----------	--------------------------------------

---

**Description**

Simple wrapper around `decorator$new(object, exists)`

**Usage**

```
decorate(object, decorators, exists = c("skip", "error", "overwrite"), ...)
```

**Arguments**

object	[R6::R6Class] R6 class to decorate.
decorators	([DecorateClass] character()) One or more decorators (by name or class) to decorate with.
exists	(character(1)) Expected behaviour if method exists in object and decorator. One of: 1. exists = "error" (default) - This will throw an error and prevent the object being decorated. 2. exists = "skip" - This will decorate the object with all fields/methods that don't already exist. 3. exists = "overwrite" - This will decorate the object with all fields/methods from the decorator and overwrite ones with the same name if they already exist.
...	ANY Additional arguments passed to <a href="#">get</a> .

**See Also**

[DecoratorClass](#)

**Examples**

```
library(R6)

## Define decorators
dec1 <- DecoratorClass("dec1", public = list(goodbye = "Goodbye World"))
dec2 <- DecoratorClass("dec2", public = list(goodbye2 = "Goodbye World 2"))

oop <- ooplah$new()
oop$goodbye
dec_oop <- decorate(oop, c(dec1, dec2))
dec_oop$goodbye
dec_oop$goodbye2

## Equivalently
oop <- ooplah$new()
decorate(oop, c("dec1", "dec2"))
```

---

DecoratorClass

*Create an abstract R6 Class*

---

### Description

Creates a decorator R6 class by placing a thin wrapper around [R6::R6Class](#) which allows the constructed class to inherit the fields and methods of the given object.

### Details

The decorator design pattern allows methods to be added to an object without bloating the interface with too many methods on construction and without causing large inheritance trees. A decorator class contains fields/methods that are 'added' to the given object in construction, this is made clearer in examples.

There are three possibilities when trying to decorate an object with a field/method that already exists:

1. `exists = "skip"` (default) - This will decorate the object with all fields/methods that don't already exist
2. `exists = "error"` - This will throw an error and prevent the object being decorated
3. `exists = "overwrite"` - This will decorate the object with all fields/methods from the decorator and overwrite ones with the same name if they already exist

Decorators are currently not cloneable.

All arguments of [R6::R6Class](#) can be used as usual, see full details at [R6::R6Class](#).

### References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1996). Design Patterns: Elements of Reusable Software. Addison-Wesley Professional Computing Series (p. 395).

### See Also

[decorate](#)

### Examples

```
library(R6)

## Create two decorators
# Works with active bindings...
dec1 <- DecoratorClass("dec1", active = list(hi = function() "Hi World"))
# And public fields...
dec2 <- DecoratorClass("dec2", public = list(goodbye = "Goodbye World"))

## Create an object to decorate
oop <- ooplah$new()
oop$hello()
```

```

## Decorate with dec1 by constructing dec1 with object oop:
dec_oop <- dec1$new(oop) # equiv `decorate(oop, dec1)`
## We have all original methods from oop
dec_oop$hello()
# It's inherited methods
dec_oop$init
# And now decorated methods
dec_oop$hi

## We can decorate again
redec_oop <- dec2$new(dec_oop)
redec_oop$hello()
redec_oop$init
redec_oop$hi
# And now
redec_oop$goodbye

# Notice the class reflects all decorators, the original object and parents,
# and adds the 'Decorator' class
class(redec_oop)

## Decorators also work with inheritance
parent_dec <- DecoratorClass("parent_dec",
  public = list(hi = function() "Hi!"))
child_dec <- DecoratorClass("child_dec", inherit = parent_dec)
dec_oop <- child_dec$new(ooplah$new())
dec_oop$hi()

## Three possibilities if the method/field name already exists:
oop <- ooplah$new()
exists_dec <- DecoratorClass("exists_dec",
  public = list(hello = function() "Hi!"))

# 1. skip (default)
oop$hello()
exists_dec$new(oop, exists = "skip")$hello()

# 2. error
## Not run:
exists_dec$new(oop)
exists_dec$new(oop, exists = "error")

## End(Not run)

# 3. overwrite
oop$hello()
exists_dec$new(oop, exists = "overwrite")$hello()

## Cloning
# Note that by default the decorated object is not cloned
dec <- DecoratorClass("dec", active = list(hi = function() "Hi World"))

```

```
dec_oop <- dec$new(oop)
dec_oop$logically
oop$logically <- FALSE
dec_oop$logically
```

---

is.R6 *Is 'x' a R6 object or class?*

---

### Description

Assert/test if 'x' is a R6 object or class

### Usage

```
is.R6(x)

assert_R6(x)
```

### Arguments

x                    Object to test

### Value

Either TRUE/FALSE is testing if x inherits from R6 or R6ClassGenerator, otherwise returns x invisibly on assertion if TRUE or returns an error if FALSE

---

is.R6Class *Is 'x' a R6 class?*

---

### Description

Assert/test if 'x' is a R6 class

### Usage

```
is.R6Class(x)

assert_R6Class(x)
```

### Arguments

x                    Object to test

### Value

Either TRUE/FALSE is testing if x inherits from R6ClassGenerator, otherwise returns x invisibly on assertion if TRUE or returns an error if FALSE

---

is.R6Object	<i>Is 'x' a R6 object?</i>
-------------	----------------------------

---

**Description**

Assert/test if 'x' is a R6 object

**Usage**

```
is.R6Object(x)
```

```
assert_R6Object(x)
```

**Arguments**

x	Object to test
---	----------------

**Value**

Either TRUE/FALSE is testing if x inherits from R6, otherwise returns x invisibly on assertion if TRUE or returns an error if FALSE

---

loapply	<i>Specialised lapply for objects</i>
---------	---------------------------------------

---

**Description**

Specialised lapply functions for R6 or other OOP classes. This is simply a wrapper that detects if FUN is a function, in which case lapply is used as usual, or a string, in which case the given field/method is returned as a list.

**Usage**

```
loapply(X, FUN, ...)
```

**Arguments**

X, ...	See <a href="#">lapply</a>
--------	----------------------------

FUN	Either a function to apply to each element of X, as in <a href="#">lapply</a> or the field/method name of an OOP object (see examples)
-----	--

**Examples**

```
## lapply as usual
loapply(c(1, 2, 3), identity)

## For R6 objects
objs <- list(ooplah$new(), ooplah$new())
# Public field
loapply(objs, "oop")
# Public method
loapply(objs, "hello")
```

---

object\_class

*Get class of an object (possibly with inheritance)*


---

**Description**

Find class of an object or an ancestor of the object. In contrast to `class` which returns a class object and all its ancestors, this function returns either the class of the object itself, or the class of one of its ancestors.

**Usage**

```
object_class(object, ancestor = 0)

get_object_class(object, ancestor = 0, ...)

object_classes(..., objects = list(...))
```

**Arguments**

object	ANY Object to get the class of
ancestor	(integer(1)) If greater than 0 then the given ancestor to get the class for, see examples
...	ANY Objects to vapply over
objects	(list(1)) Alternative constructor with list of objects

**Details**

`object_classes` is a stripped-down wrapper to get the class of multiple objects



**Examples**

```
library(R6)

class_a <- R6Class("class_a")
class_b <- R6Class("class_b", inherit = class_a)
class(class_b$new())
object_class(class_b$new())
object_class(class_b$new(), 1)
```

---

ooplah	<i>R6 Class for testing and examples</i>
--------	--

---

**Description**

R6 Class for testing and examples

---

private	<i>Get R6 object private environment</i>
---------	--

---

**Description**

Access the private environment of an R6 object

**Usage**

```
private(x)
```

**Arguments**

x	(R6) R6 object to get environment from, errors if not R6
---	---

---

super	<i>Get R6 object parent environment</i>
-------	---

---

**Description**

Access the parent environment of an R6 object

**Usage**

```
super(x)
```

**Arguments**

x	(R6) R6 object to get environment from, errors if not R6
---	---

---

`vxapply`*Specialised vapply methods for atomic classes*

---

## Description

Specialised vapply functions for scalars of each of the six atomic classes in R:

## Usage

```
vlapply(X, FUN, ..., USE.NAMES = TRUE)
```

```
viapply(X, FUN, ..., USE.NAMES = TRUE)
```

```
vnapply(X, FUN, ..., USE.NAMES = TRUE)
```

```
vcapply(X, FUN, ..., USE.NAMES = TRUE)
```

```
vzapply(X, FUN, ..., USE.NAMES = TRUE)
```

```
vrapply(X, FUN, ..., USE.NAMES = TRUE)
```

## Arguments

`X, ..., USE.NAMES`

See [vapply](#)

`FUN`

Either a function to apply to each element of `X`, as in [vapply](#) or the field/method name of an OOP object (see examples)

## Details

- logical ([vlapply](#))
- integer ([viapply](#))
- numeric/real ([vnapply](#))
- character/string ([vcapply](#))
- complex ([vzapply](#))
- raw ([vrapply](#))

These are simply wrappers around [vapply](#) where `FUN.VALUE` is pre-filled with a scalar of the given class.

In addition these can be applied to pull-out fields or methods from R6 or other OOP objects by supplying the field/method name to `FUN`. See examples.

**Examples**

```
## Specialised vapply
vapply(logical(10), identity)
vzapply(complex(10), identity)

## For R6 objects
objs <- list(ooplah$new(), ooplah$new())

# Public field
vcapply(objs, "oop")

# Public method
vcapply(objs, "exclaim", "ARGH")
vcapply(objs, "hello")
vnapply(objs, "generate", 1)

# Active binding
vlapply(objs, "logically")
```

# Index

AbstractClass, [2](#)  
assert\_R6 (is.R6), [6](#)  
assert\_R6Class (is.R6Class), [6](#)  
assert\_R6Object (is.R6Object), [7](#)  
  
decorate, [3, 4](#)  
DecoratorClass, [3, 4](#)  
  
get, [3](#)  
get\_object\_class (object\_class), [8](#)  
  
is.R6, [6](#)  
is.R6Class, [6](#)  
is.R6Object, [7](#)  
  
lapply, [7](#)  
lapply, [7](#)  
  
object\_class, [8](#)  
object\_classes (object\_class), [8](#)  
ooplah, [9](#)  
  
private, [9](#)  
  
R6::R6Class, [2, 4](#)  
  
super, [9](#)  
  
vapply, [10](#)  
vcapply (vxapply), [10](#)  
viapply (vxapply), [10](#)  
vlapply (vxapply), [10](#)  
vnapply (vxapply), [10](#)  
vrapply (vxapply), [10](#)  
vxapply, [10](#)  
vzapply (vxapply), [10](#)