Package: reportr (via r-universe)

November 12, 2026

Version 2.0.0

Date 2026-10-12

Title A General Message and Error Reporting System

Depends R (>= 3.5.0)

Imports ore

Suggests tinytest, covr

Description Provides a system for reporting messages, which provides certain useful features over the standard R system, such as the incorporation of output consolidation, message filtering, assertions, expression substitution, automatic generation of stack traces for debugging, and conditional reporting based on the current ``output level''. Messages are signalled as classed R conditions, so that calling code can decide how serious a situation is, rather than the function which detects it having to be told in advance.

Encoding UTF-8

License GPL-2

URL https://github.com/jonclayden/reportr

BugReports https://github.com/jonclayden/reportr/issues

Config/roxygen2/version 8.1.0

Config/roxygen2/markdown TRUE

Config/roxygen2/old_usage TRUE

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

Date/Publication 2026-10-13 07:19:01 UTC

RemoteUrl https://github.com/jonclayden/reportr

RemoteRef HEAD

RemoteSha 4d70a0c5455a9441aba108db5ad5e55732c2774a

Contents

reportr-package 2
fallback 3
handlers 5
report 7
reportrCondition 10
setOutputLevel 12
targets 13

Index **16**

`reportr-package` *reportr: A general message and error reporting system*

Description

The reportr package provides alternatives to `message()`, `warning()` and `stop()`, with output consolidation, message filtering, expression substitution, automatic stack traces for debugging, and reporting which depends on the current output level. Every message is also signalled as a classed R condition, so that calling code can decide how serious a situation is, rather than the function which detects it having to be told in advance.

Levels

Every message has a level. The levels are given by the `OL` object, a list whose components are, in ascending order of priority, `Debug`, `Verbose`, `Info`, `Warning`, `Question`, `Error` and `Fatal`. Messages at level `Error` or `Fatal` stop execution. There is also an `Ignore` level, below `Debug`, which is used with `reportAs()` to suppress a class of message entirely.

Wherever reportr expects a level, it may be given as `OL$Info`, as the bare name `Info`, or as the string `"Info"`.

Main functions

- `report()`, `flag()` and `assert()` report messages, and `ask()` asks questions.
- `fallback()` raises an error which the caller may choose to demote, in return for a substitute value.
- `reportAs()` changes the level of particular classes of message, and `withReportrHandlers()` brings all of R's messages, warnings and errors under reportr's control. See handlers.
- `setOutputLevel()` controls which messages are written, and `setOutputTargets()` where they are written and how they are formatted. See targets.
- `reportrCondition()` describes the condition objects signalled for each message.

Options

The following options influence reportr's behaviour. Each has a default which applies if it is not set.

- `reportrOutputLevel`: The current output level, which is more usually set using `setOutputLevel()`.
- `reportrPrefixFormat`: The prefix format used by output targets which don't specify their own; see targets. The default is `"%d%L: "`.
- `reportrTimeFormat`: The format used for the time in prefixes, as understood by `format.POSIXct()`. The default is `"%Y-%m-%d %H:%M:%S"`.
- `reportrStderrLevel`: The level at and above which the terminal writes messages to standard error rather than standard output, unless the terminal target specifies its own. The default is `Warning`.
- `reportrStackTraceLevel`: The level at and above which a stack trace is written with each message, when the output level is `Debug`. The default is `Error`.
- `reportrMessageFilterIn`, `reportrMessageFilterOut`: Perl-style regular expressions. If set, only messages which match the first, and do not match the second, are reported. Filtering affects only what is reported: the corresponding condition is signalled either way, and a filtered error is still fatal.
- `reportrBaseClasses`: Should reportr conditions also inherit from base R's `"message"` and `"warning"` classes? The default is `FALSE`; see `reportrCondition()`.
- `reportrAskInScripts`: Should `ask()` read answers directly from standard input in a non-interactive session, such as a script run with `Rscript`, if standard input is a terminal? The default is `TRUE`. If `FALSE`, the default answer is always used in non-interactive sessions.

Author(s)

Jon Clayden

See Also

Useful links:

- https://github.com/jonclayden/reportr
- Report bugs at https://github.com/jonclayden/reportr/issues

---

`fallback` *Return a fallback value, with a recoverable error*

Description

A function which cannot do what was asked of it can use `fallback()` to report the problem and return a substitute value instead. By default the problem is an error, but one which the caller may choose to demote, in which case the substitute value is returned and execution continues.

Usage

```
fallback(value, ..., class = NULL, level = OL$Error, call = sys.call(-1))
```

Arguments

value	The value to return if the condition is not handled, or is handled by demoting it. It may be NULL, but must be given.
...	Objects which can be coerced to mode character, and optionally arguments to ore::es() such as round. These are handled exactly as by report().
class	A character vector of classes to attach to the condition, or NULL. A class derived from the level of the message, such as "reportrError", is always attached in addition to these.
level	The level of the message. See reportr for the available levels and the ways they may be named.
call	The call to associate with the condition. Defaults to the call of the function which is signalling.

Details

Often only the caller knows how serious a problem is: a missing file may be fatal in one context and unremarkable in another. One workaround is a logical argument indicating whether a certain condition should be raised as an error, but that puts the decision in the wrong place, and has to be passed down through every intervening layer. Instead, the function can signal a classed condition, and supply the value it would return if the caller decides that execution should continue:

```
locateFile <- function (path) {
  if (!file.exists(path))
    return(fallback(NULL, "File #{path} does not exist", class="missingFile"))
  ...
}
```

Callers who do nothing get an error. Callers for whom a missing file is expected can say so with reportAs(locateFile(path), missingFile=Debug), and get NULL back. A handler may also supply a different value, using the useValue restart described in reportrCondition().

Value

The value of value, or a value supplied by a handler through the useValue restart. At level Error or above fallback() does not return unless the condition is handled.

Choosing between fallback(), report() and assert()

Escalating a condition to Error is always safe, because execution simply stops sooner than it would have. Demoting an error is only safe when the code which signalled it has said what value to continue with, and so fallback() is the only way to signal an error which can be demoted. A condition which should merely be reported, whatever its class, is better signalled with report(); and a requirement which must hold, with nothing sensible to return if it does not, with assert().

The level should suit a caller who does nothing, and Error is usually right when the value returned would mislead code that expected a real result, since the mistake would otherwise surface later and somewhere else. Where the problem is visible in the result itself, such as a partial result containing

NA values, a lower level with report() and an ordinary return lets strict callers escalate it instead. A lower level may nevertheless be given here, which allows a handler to substitute a value.

Like the other signalling functions, fallback() signals its condition even when the current output level means that nothing will be reported.

Author(s)

Jon Clayden

See Also

reportAs() for acting on these conditions, reportrCondition() for the structure of the condition objects, and report() for conditions which are only reported.

Examples

```
setOutputLevel(OL$Info)

findThing <- function (name) {
  if (name != "widget")
    return(fallback(NA, "There is no #{name}", class="missingThing"))
  return("the widget")
}

# An error, if the caller does nothing
## Not run: findThing("sprocket")

# Reported as a warning instead, and NA returned
reportAs(findThing("sprocket"), missingThing=Warning)

# Suppressed entirely
reportAs(findThing("sprocket"), missingThing=Ignore)

# Suppressed, with a different value supplied through the useValue restart
reportAs(findThing("sprocket"),
         missingThing=function (cond) invokeRestart("useValue", "a sprocket"))
```

---

handlers *Handling reportr conditions*

Description

These functions evaluate an expression in a context where conditions are handled by reportr. reportAs() changes the level at which particular classes of condition are reported, which is how a caller decides how serious a situation is. withReportrHandlers() additionally arranges for R's own messages, warnings and errors to be reported by reportr, and reportrHandlers() does the same for a whole session.

Usage

```
reportAs(expr, ...)

withReportrHandlers(expr, ...)

reportrHandlers(install = TRUE)
```

Arguments

`expr`	An expression to evaluate.
`...`	Named arguments mapping condition classes to levels, or to handler functions. See Details.
`install`	Should the session-wide handlers be installed, or removed?

Details

Conditions to remap are given as named arguments, where the name is the class of condition to match and the value is the level at which it should be reported. Levels may be named in any of the usual ways, so `Error`, `OL$Error` and `"Error"` are equivalent. Mapping a class to `Ignore` suppresses it entirely.

```
# A missing file is an error, as signalled ...
file <- locateFile(path)

# ... but merely worth noting here
file <- reportAs(locateFile(path), missingFile=Debug)
```

Escalating a condition is always safe. Demoting one below `Error` is not, because code which signalled an error was generally not written to carry on afterwards. A demotion is therefore honoured only for conditions signalled by `fallback()`, which supplies a value to continue with. Demoting any other error reports it at the requested level, but it remains fatal.

A named argument may also be a function, in which case it is used as a calling handler for that class of condition, exactly as it would be by `withCallingHandlers()`. Such a handler may invoke the `muffleReport` restart, or for a condition signalled by `fallback()` the `useValue` restart; see `reportrCondition()`.

Value

`reportAs()` and `withReportrHandlers()` return the value of `expr`, invisibly if it was invisible. `reportrHandlers()` returns `NULL`, invisibly.

Handling R's own conditions

`withReportrHandlers()` also translates conditions raised by code which knows nothing of reportr: calls to `message()` are reported at level `Info`, `warning()` is flagged at level `Warning`, and `stop()` is reported at level `Error`. The classes of the original condition are carried over to the reportr condition, and the original is stored in its `original` element, so a handler further out can still match it by class.

`reportrHandlers(TRUE)` installs those same translations for the rest of the session, using `globalCallingHandlers()`, so that code need not be wrapped at all. Like that function, it may only be called when no handlers are already established, which in practice means at top level. It requires R 4.0.0 or later, and gives an error on earlier versions; the rest of the package works with R 3.5.0 or later. `reportrHandlers(FALSE)` removes them again.

Author(s)

Jon Clayden

See Also

`report()` and `fallback()` for signalling these conditions, and `reportrCondition()` for their structure.

Examples

```
setOutputLevel(OL$Info)

findThing <- function (name) {
  if (name != "widget")
    return(fallback(NA, "There is no #{name}", class="missingThing"))
  return("the widget")
}

# Reported as a warning, rather than an error
reportAs(findThing("sprocket"), missingThing=Warning)

# Suppressed entirely
reportAs(findThing("sprocket"), missingThing=Ignore)

# Consolidate duplicated warnings from code that doesn't use reportr
withReportrHandlers(sqrt(-5:-1))
```

---

report *Report messages, warnings and errors*

Description

These functions report informative messages, warnings and errors, as alternatives to `message()`, `warning()` and `stop()`. `report()` reports a message immediately, while `flag()` stores it to be reported later. `assert()` reports a message only if an expression is not `TRUE`, and `ask()` asks the user a question.

Usage

```
report(level, ..., class = NULL, call = sys.call(-1), prefix = TRUE,
  prefixFormat = NULL)

flag(level, ..., class = NULL, call = sys.call(-1))

reportFlags()

clearFlags()

assert(expr, ..., class = NULL, level = OL$Error, call = sys.call(-1),
  prefix = TRUE, prefixFormat = NULL, envir = parent.frame())

ask(..., default = NULL, valid = NULL, prefix = TRUE,
  prefixFormat = NULL)
```

Arguments

level	The level of the message. See Details.
...	Objects which can be coerced to mode character. These are pasted together and passed through ore::es() for expression substitution. Arguments to ore::es(), such as round, may also be given.
class	A character vector of classes to attach to the condition, or NULL. A class derived from the level of the message, such as "reportrWarning", is always attached in addition to these.
call	The call to associate with the condition. Defaults to the call of the function which is reporting.
prefix	Logical value: should the message be prefixed, according to the format for each output target?
prefixFormat	Deprecated. A value of "" is equivalent to prefix = FALSE, and any other value is ignored with a warning.
expr	For assert(), an expression which should evaluate to TRUE. An expression which cannot be evaluated counts as a failure.
envir	For assert(), the environment in which to evaluate expr and to substitute expressions into the message.
default	For ask(), the value to return if the question is not asked.
valid	For ask(), a character vector of valid responses. If necessary, the question will be asked repeatedly until the user gives a suitable response. Matching is not case-sensitive.

Details

Each message has a level, such as Info or Warning, which may be given as OL$Info, as the bare name Info, or as the string "Info"; see reportr for the full list. A message is reported only if its level is at least the current output level (see setOutputLevel()), and it is then written to each of the current output targets which accepts it (see targets).

The parts of the message given in ... are pasted together with no separator, after expression substitution by ore::es(), so "x is #{x}" includes the value of x in the message. Messages are written with a prefix, such as "** INFO: ", whose format is set by each output target. Setting prefix = FALSE writes a message without any prefix, to every target, which is useful for headings and other formatted output.

Value

ask() returns the user's response as a character string, or the value of default. The other functions are called for their side effects, and return NULL, invisibly.

Conditions

Every message is signalled as an R condition before it is reported, so that calling code can intercept it. Its class vector always includes one derived from the level of the message, such as "reportrInfo", so tryCatch(expr, reportrInfo=...) will match. See reportrCondition() for the structure of these objects.

The class argument attaches classes of your own to the condition. This is worth doing whenever a caller might reasonably want to act on a particular kind of message: reportAs() can then escalate, demote or suppress it, without the function which detects the problem having to be told in advance how serious it is. For example, a function which returns a partial result might report the omissions at level Warning with a class, and leave any caller who needs a complete result to escalate that class to Error. Unlike unclassed messages, classed ones are signalled even when the current output level means that they will not be reported, since the class is there for the benefit of callers.

Errors

A message at level Error or above stops execution, like stop(), but the message is formatted by reportr rather than by R, and a stack trace is written if the current output level is Debug and the level of the message is at least the reportrStackTraceLevel option. Since the condition is signalled before the message is reported, such errors can be caught with try() or tryCatch(expr, error=...) in the usual way. If nothing handles the condition the "abort" restart is invoked, which ends execution without R adding a second message of its own. An error is fatal whatever the current output level, and whether or not its message is filtered out of the output.

An error raised by report() or assert() cannot be demoted by the caller, because the function which raised it has nothing sensible to continue with. A function which cannot do what was asked of it, but has a value to return instead, should use fallback().

Flags

flag() is called like report(), but stores the message for later reporting, rather than reporting it immediately. Its condition is signalled straight away, though. Stored messages are reported when a message is next reported, at which point multiple instances of the same message are consolidated, and each is written as it would have been where it arose. reportFlags() reports stored messages immediately, and clearFlags() discards them. The output level at the time the stored messages are reported determines whether they are written. At output level Debug, flagged messages at or above the reportrStackTraceLevel option are reported immediately, so that they appear where they arise.

Questions

ask() requests input from the user at level Question. In an interactive session it uses readline(). Otherwise, as when a script is run with Rscript, it reads directly from standard input, provided that this is a terminal and the reportrAskInScripts option is not FALSE. It returns the text entered by the user, or default if there is nobody to ask, the output level is above Question, the question is filtered out, or input ends before an answer is given.

Author(s)

Jon Clayden

See Also

fallback() for errors which callers may demote; reportAs() for acting on classed messages; setOutputLevel() and targets for controlling what is written, and where; and reportrCondition() for the condition objects themselves.

Examples

```
setOutputLevel(OL$Warning)
report(Info, "Test message")   # no output
setOutputLevel(OL$Info)
report(Info, "Test message")   # prints the message

flag(Warning, "Test warning")  # no output
flag(Warning, "Test warning")  # repeated warning
reportFlags()  # consolidates the warnings and prints the message

x <- 3
report(Info, "The value of x is #{x}")
report(Info, "A heading", prefix=FALSE)

assert(x > 2, "x is too small")
tryCatch(report(Warning, "Something is amiss", class="myWarning"),
         myWarning=function (cond) conditionMessage(cond))

## Not run:
name <- ask("What is your name?")
report(Info, "Hello, #{name}")

## End(Not run)
```

---

reportrCondition *reportr conditions*

Description

Every message reported by reportr is first signalled as an R condition, which allows calling code to intercept it, inspect it, and decide how it should be treated. `reportrCondition()` constructs such a condition object.

Usage

```
reportrCondition(level, message, class = NULL, call = NULL,
  data = list())
```

Arguments

`level`	The level of the message. See reportr.
`message`	A character string giving the message, in its final form.
`class`	A character vector of additional classes for the condition, or `NULL`.
`call`	The call in which the condition arose, or `NULL`.
`data`	A list of further elements to store in the condition object.

Details

The class vector of the condition is assembled from most to least specific: any classes given in the `class` argument; a class derived from the level of the message, such as `"reportrWarning"`; the general class `"reportrCondition"`; and finally `"condition"`. The level-derived class is always present, so a condition can always be matched by a handler even when no explicit class is given.

Conditions at level `Error` or above additionally inherit from `"error"`, so they are caught by `try()` and by `tryCatch(expr, error=...)` in the usual way. If the `reportrBaseClasses` option is `TRUE`, conditions at level `Warning` also inherit from `"warning"`, and those at level `Info` and below from `"message"`, which makes them susceptible to `suppressWarnings()` and `suppressMessages()`. This is not the default, since it allows unrelated code to silence reportr output.

In addition to the standard `message` and `call` elements, the condition carries the `level` of the message, the `stack` of calls in effect when it was signalled, a `plain` flag indicating whether it is to be written without a prefix, and a `recoverable` flag. A condition signalled by `fallback()` is recoverable, which means that the signalling function has a value to return instead of doing what was asked of it, and that value is stored in the `default` element. For any other condition `default` is `NULL`.

Value

A condition object.

Restarts

Every condition establishes a `muffleReport` restart while it is being signalled. Invoking it suppresses the reporting of the message, but does not otherwise affect the flow of control, so an error will still be fatal. Recoverable conditions additionally establish a `useValue` restart. Invoking it with a single argument suppresses reporting *and* any error, and provides the value which the signalling function will return in place of its default.

Author(s)

Jon Clayden

See Also

`report()` and `fallback()` for signalling conditions, `reportAs()` for handling them, and conditions for R's condition system in general.

Examples

```
cond <- reportrCondition(OL$Warning, "Something is amiss", class="myWarning")
class(cond)
conditionMessage(cond)

# A handler can recover from an error raised by fallback(), and provide a
# value of its own for the signalling function to return
lookup <- function (key) {
  if (key != "a")
    return(fallback(NULL, "No value for key #{key}", class="missingKey"))
  return(1)
}
withCallingHandlers(lookup("b"), missingKey=function (cond) {
  if (cond$recoverable)
    invokeRestart("useValue", 0)
})

# Other conditions can only be muffled, which suppresses the message
withCallingHandlers(report(Warning, "Not shown"),
                    reportrWarning=function (cond) invokeRestart("muffleReport"))
```

`setOutputLevel` *The output level*

Description

The output level is the master gate for reported messages: a message whose level is below it is not written to any output target. `setOutputLevel()` sets it, and `getOutputLevel()` returns it.

Usage

```
setOutputLevel(level)

getOutputLevel()
```

Arguments

`level`	The minimum level of message to write. Levels may be named in any of the usual ways, so `Info`, `OL$Info` and `"Info"` are equivalent; see reportr for the full list. Levels outside the range `Debug` to `Fatal` are ignored.

Details

The output level is stored in the `reportrOutputLevel` option, which may also be set directly. If a message is reported before the output level has been set, it is set to `Info`, with a message to say so.

The output level governs only what is written. Errors are fatal whatever the output level, and the conditions corresponding to messages which are not written are still signalled when they have a class, or when handlers established by `reportAs()` or `withReportrHandlers()` are active. At level `Debug`, a stack trace is also written with any message at or above the level given by the `reportrStackTraceLevel` option. Output targets may apply stricter thresholds of their own; see targets.

Value

`setOutputLevel()` returns `NULL`, invisibly. `getOutputLevel()` returns the current output level, as an integer named after the level.

Author(s)

Jon Clayden

See Also

`report()`, and targets for controlling where messages are written.

Examples

```
setOutputLevel(Warning)
getOutputLevel()
report(Info, "Not written")

setOutputLevel(OL$Info)
report(Info, "Written")
```

`targets` *Output targets*

Description

Messages accepted for reporting are written to one or more output targets, which may be the terminal, files, connections or arbitrary functions. By default there is a single target, which writes to the terminal. `setOutputTargets()` replaces the current targets with those given, or adds to them, and `getOutputTargets()` returns them. `toTerminal()`, `toFile()` and `toFunction()` create targets.

Usage

```
toTerminal(prefix = NULL, stdout = NULL, stderr = NULL)

toFile(file, prefix = NULL, level = NULL, append = TRUE)

toFunction(fun, prefix = NULL, level = NULL)

setOutputTargets(..., add = FALSE)

getOutputTargets()
```

Arguments

prefix	The prefix format for messages written to this target. See the Prefixes section.
stdout, stderr	The minimum levels of message written to standard output and standard error, respectively. If NULL, all messages passing the master gate are written to standard output, and those at or above the level given by the reportrStderrLevel option (by default, Warning) are written to standard error instead.
file	The path to a file, or a connection.
level	The minimum level of message which this target will accept, or NULL to accept any which passes the master gate.
append	If file is a path, should the file be appended to rather than overwritten?
fun	A function, which is called with three arguments: the formatted text, the level of the message, and the condition object (which may be NULL).
...	For setOutputTargets(), target objects created by the constructor functions, or lists of them such as those returned by getOutputTargets(). Arguments may be named, and otherwise a name is chosen for each target. Giving no targets at all silences all output, unless add is TRUE.
add	Logical value: if TRUE, the targets given are added to the current ones, rather than replacing them.

Details

The current output level (see setOutputLevel()) is the master gate: a message which it suppresses never reaches any target. A target may additionally set its own level threshold, so that, for example, a log file captures everything from Debug upwards while the terminal shows only Info and above. A threshold of NULL, the default, accepts every message which passes the master gate. Levels may be named in any of the usual ways, so Info, OL$Info and "Info" are equivalent.

The terminal target writes messages below its stderr level to standard output, and the rest to standard error.

Files named by path are opened when they are first written to, and closed when a call to setOutputTargets() no longer includes them. Connections passed to toFile() are used as they are, and never closed by reportr.

Value

`setOutputTargets()` invisibly returns the targets which were in effect before the call, so that they can be restored later. `getOutputTargets()` returns a named list of the current targets. The constructors return a single target object.

Prefixes

Each target has its own prefix format, which describes the string prepended to each message. It takes the form of a `sprintf()`-style format string, but with different expansions for percent-escapes:

- `%d` expands to a series of stars indicating the stack depth at which the message arose;
- `%f` gives the name of the function which raised the message;
- `%l` and `%L` give lower and upper case versions of the level of the message;
- `%p` expands to the ID of the current R process (see `Sys.getpid()`); and
- `%t` expands to the current time, formatted according to the `reportrTimeFormat` option.

A prefix of `NULL`, the default, uses the value of the `reportrPrefixFormat` option at the time the message is written, whose own default is `"%d%L: "`, giving a prefix such as `"** INFO: "`. A prefix of `FALSE` or `""` means no prefix. Individual messages can be written without any prefix, whatever the target, using the `prefix` argument to `report()`.

Author(s)

Jon Clayden

See Also

`report()` and `setOutputLevel()`

Examples

```
## Not run:
# Log everything to a file, while the terminal shows warnings and above
setOutputLevel(OL$Debug)
old <- setOutputTargets(toTerminal(stdout=Warning),
                        toFile("run.log", prefix="%t %L: "))

# Add another target to the current ones
setOutputTargets(log=toFunction(function (text, level, condition) ...), add=TRUE)

# Restore the targets in effect before
setOutputTargets(old)

## End(Not run)
```

Index

ask *(report)*, 7
ask(), *2, 3*
assert *(report)*, 7
assert(), *2*, *4*

clearFlags *(report)*, 7
conditions, *12*

fallback, 3
fallback(), *2*, *6, 7*, *9–12*
flag *(report)*, 7
flag(), *2*
format.POSIXct(), *3*

getOutputLevel *(setOutputLevel)*, 12
getOutputTargets *(targets)*, 13

handlers, *2*, 5

message(), *2*, *6, 7*

OL *(reportr-package)*, 2
ore::es(), *4*, *8, 9*

readline(), *10*
report, 7
report(), *2*, *4, 5*, *7*, *12, 13*, *15*
reportAs *(handlers)*, 5
reportAs(), *2*, *5*, *9, 10*, *12, 13*
reportFlags *(report)*, 7
reportr, *4*, *8*, *11, 12*
reportr *(reportr-package)*, 2
reportr-package, 2
reportrCondition, 10
reportrCondition(), *2–7*, *9, 10*
reportrHandlers *(handlers)*, 5

setOutputLevel, 12
setOutputLevel(), *2, 3*, *8*, *10*, *14, 15*
setOutputTargets *(targets)*, 13
setOutputTargets(), *2*

sprintf(), *15*
stop(), *2*, *6, 7*, *9*
suppressMessages(), *11*
suppressWarnings(), *11*
Sys.getpid(), *15*

targets, *2, 3*, *8*, *10*, 13, *13*
toFile *(targets)*, 13
toFunction *(targets)*, 13
toTerminal *(targets)*, 13
try(), *9*, *11*

warning(), *2*, *6, 7*
withCallingHandlers(), *6*
withReportrHandlers *(handlers)*, 5
withReportrHandlers(), *2*, *13*