# Active Support Core Extensions

January 13, 2015

Active Support is the Ruby on Rails component responsible for providing Ruby language extensions, utilities, and other transversal stuff.

It offers a richer bottom-line at the language level, targeted both at the development of Rails applications, and at the development of Ruby on Rails itself.

After reading this guide, you will know:

- What Core Extensions are.
- How to load all extensions.
- How to cherry-pick just the extensions you want.
- What extensions Active Support provides.

## 1 How to Load Core Extensions

### 1.1 Stand-Alone Active Support

In order to have a near-zero default footprint, Active Support does not load anything by default. It is broken in small pieces so that you can load just what you need, and also has some convenience entry points to load related extensions in one shot, even everything.

Thus, after a simple require like:

```
require 'active_support'
```

objects do not even respond to `blank?`. Let's see how to load its definition.

#### 1.1.1 Cherry-picking a Definition

The most lightweight way to get `blank?` is to cherry-pick the file that defines it.

For every single method defined as a core extension this guide has a note that says where such a method is defined. In the case of `blank?` the note reads:

Defined in `active_support/core_ext/object/blank.rb`.

That means that you can require it like this:

```
require 'active_support'
require 'active_support/core_ext/object/blank'
```

Active Support has been carefully revised so that cherry-picking a file loads only strictly needed dependencies, if any.

**1.1.2 Loading Grouped Core Extensions** The next level is to simply load all extensions to `Object`. As a rule of thumb, extensions to `SomeClass` are available in one shot by loading `active_support/core_ext/some_class`.

Thus, to load all extensions to `Object` (including `blank?`):

```
require 'active_support'
require 'active_support/core_ext/object'
```

**1.1.3 Loading All Core Extensions** You may prefer just to load all core extensions, there is a file for that:

```
require 'active_support'
require 'active_support/core_ext'
```

**1.1.4 Loading All Active Support** And finally, if you want to have all Active Support available just issue:

```
require 'active_support/all'
```

That does not even put the entire Active Support in memory upfront indeed, some stuff is configured via `autoload`, so it is only loaded if used.

### 1.2 Active Support Within a Ruby on Rails Application

A Ruby on Rails application loads all Active Support unless `config.active_support.bare` is true. In that case, the application will only load what the framework itself cherry-picks for its own needs, and can still cherry-pick itself at any granularity level, as explained in the previous section.

## 2 Extensions to All Objects

### 2.1 `blank?` and `present?`

The following values are considered to be blank in a Rails application:

- `nil` and `false`,
- strings composed only of whitespace (see note below),
- empty arrays and hashes, and
- any other object that responds to `empty?` and is empty.

The predicate for strings uses the Unicode-aware character class `[:space:]`, so for example U+2029 (paragraph separator) is considered to be whitespace.

Note that numbers are not mentioned. In particular, 0 and 0.0 are **not** blank.

For example, this method from `ActionController::HttpAuthentication::Token::ControllerMethods` uses `blank?` for checking whether a token is present:

```
def authenticate(controller, &login_procedure)
  token, options = token_and_options(controller.request)
  unless token.blank?
    login_procedure.call(token, options)
  end
end
```

The method `present?` is equivalent to `!blank?`. This example is taken from `ActionDispatch::Http::Cache::Response`:

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  ...
end
```

Defined in `active_support/core_ext/object/blank.rb`.

## 2.2 presence

The `presence` method returns its receiver if `present?`, and `nil` otherwise. It is useful for idioms like this:

```
host = config[:host].presence || 'localhost'
```

Defined in `active_support/core_ext/object/blank.rb`.

## 2.3 duplicable?

A few fundamental objects in Ruby are singletons. For example, in the whole life of a program the integer 1 refers always to the same instance:

```
1.object_id                 # => 3
Math.cos(0).to_i.object_id  # => 3
```

Hence, there's no way these objects can be duplicated through `dup` or `clone`:

```
true.dup  # => TypeError: can't dup TrueClass
```

Some numbers which are not singletons are not duplicable either:

```
0.0.clone        # => allocator undefined for Float
(2**1024).clone  # => allocator undefined for Bignum
```

Active Support provides `duplicable?` to programmatically query an object about this property:

```
"foo".duplicable? # => true
"".duplicable?    # => true
0.0.duplicable?   # => false
false.duplicable? # => false
```

By definition all objects are `duplicable?` except `nil`, `false`, `true`, symbols, numbers, class, module, and method objects.

Any class can disallow duplication by removing `dup` and `clone` or raising exceptions from them. Thus only `rescue` can tell whether a given arbitrary object is duplicable. `duplicable?` depends on the hard-coded list above, but it is much faster than `rescue`. Use it only if you know the hard-coded list is enough in your use case.

Defined in `active_support/core_ext/object/duplicable.rb`.

## 2.4 deep_dup

The `deep_dup` method returns deep copy of a given object. Normally, when you `dup` an object that contains other objects, Ruby does not `dup` them, so it creates a shallow copy of the object. If you have an array with a string, for example, it will look like this:

```
array     = ['string']
duplicate = array.dup

duplicate.push 'another-string'

# the object was duplicated, so the element was added only to the duplicate
array     # => ['string']
duplicate # => ['string', 'another-string']

duplicate.first.gsub!('string', 'foo')

# first element was not duplicated, it will be changed in both arrays
array     # => ['foo']
duplicate # => ['foo', 'another-string']
```

As you can see, after duplicating the `Array` instance, we got another object, therefore we can modify it and the original object will stay unchanged. This is not true for array's elements, however. Since `dup` does not make deep copy, the string inside the array is still the same object.

If you need a deep copy of an object, you should use `deep_dup`. Here is an example:

```
array     = ['string']
duplicate = array.deep_dup

duplicate.first.gsub!('string', 'foo')

array     # => ['string']
duplicate # => ['foo']
```

If the object is not duplicable, `deep_dup` will just return it:

```
number = 1
duplicate = number.deep_dup
number.object_id == duplicate.object_id   # => true
```

Defined in `active_support/core_ext/object/deep_dup.rb`.

## 2.5 try

When you want to call a method on an object only if it is not `nil`, the simplest way to achieve it is with conditional statements, adding unnecessary clutter. The alternative is to use `try`. `try` is like `Object#send` except that it returns `nil` if sent to `nil`.

Here is an example:

```
# without try
unless @number.nil?
  @number.next
end

# with try
@number.try(:next)
```

Another example is this code from `ActiveRecord::ConnectionAdapters::AbstractAdapter` where `@logger` could be `nil`. You can see that the code uses `try` and avoids an unnecessary check.

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

`try` can also be called without arguments but a block, which will only be executed if the object is not nil:

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

Defined in `active_support/core_ext/object/try.rb`.

## 2.6 class_eval(*args, &block)

You can evaluate code in the context of any object's singleton class using `class_eval`:

```
class Proc
  def bind(object)
    block, time = self, Time.current
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

Defined in `active_support/core_ext/kernel/singleton_class.rb`.

## 2.7 acts_like?(duck)

The method `acts_like?` provides a way to check whether some class acts like some other class based on a simple convention: a class that provides the same interface as `String` defines

```
def acts_like_string?
end
```

which is only a marker, its body or return value are irrelevant. Then, client code can query for duck-type-safeness this way:

```
some_klass.acts_like?(:string)
```

Rails has classes that act like `Date` or `Time` and follow this contract.

Defined in `active_support/core_ext/object/acts_like.rb`.

## 2.8 to_param

All objects in Rails respond to the method `to_param`, which is meant to return something that represents them as values in a query string, or as URL fragments.

By default `to_param` just calls `to_s`:

```
7.to_param # => "7"
```

The return value of `to_param` should **not** be escaped:

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Several classes in Rails overwrite this method.

For example `nil`, `true`, and `false` return themselves. `Array#to_param` calls `to_param` on the elements and joins the result with "/":

```
[0, true, String].to_param # => "0/true/String"
```

Notably, the Rails routing system calls `to_param` on models to get a value for the `:id` placeholder. `ActiveRecord::Base#to_param` returns the `id` of a model, but you can redefine that method in your models. For example, given

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

we get:

```
user_path(@user) # => "/users/357-john-smith"
```

Controllers need to be aware of any redefinition of `to_param` because when a request like that comes in "357-john-smith" is the value of `params[:id]`.

Defined in `active_support/core_ext/object/to_param.rb`.

## 2.9 to_query

Except for hashes, given an unescaped `key` this method constructs the part of a query string that would map such key to what `to_param` returns. For example, given

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

we get:

```
current_user.to_query('user') # => user=357-john-smith
```

This method escapes whatever is needed, both for the key and the value:

```
account.to_query('company[name]')
# => "company%5Bname%5D=Johnson+%26+Johnson"
```

so its output is ready to be used in a query string.

Arrays return the result of applying `to_query` to each element with `_key_[]` as key, and join the result with "&":

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

Hashes also respond to `to_query` but with a different signature. If no argument is passed a call generates a sorted series of key/value assignments calling `to_query(key)` on its values. Then it joins the result with "&":

```
{c: 3, b: 2, a: 1}.to_query # => "a=1&b=2&c=3"
```

The method `Hash#to_query` accepts an optional namespace for the keys:

```
{id: 89, name: "John Smith"}.to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

Defined in `active_support/core_ext/object/to_query.rb`.

## 2.10 with_options

The method `with_options` provides a way to factor out common options in a series of method calls.

Given a default options hash, `with_options` yields a proxy object to a block. Within the block, methods called on the proxy are forwarded to the receiver with their options merged. For example, you get rid of the duplication in:

```
class Account < ActiveRecord::Base
  has_many :customers, dependent: :destroy
  has_many :products,  dependent: :destroy
  has_many :invoices,  dependent: :destroy
  has_many :expenses,  dependent: :destroy
end
```

this way:

```
class Account < ActiveRecord::Base
  with_options dependent: :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

That idiom may convey *grouping* to the reader as well. For example, say you want to send a newsletter whose language depends on the user. Somewhere in the mailer you could group locale-dependent bits like this:

```
I18n.with_options locale: user.locale, scope: "newsletter" do |i18n|
  subject i18n.t :subject
  body    i18n.t :body, user_name: user.name
end
```

Since `with_options` forwards calls to its receiver they can be nested. Each nesting level will merge inherited defaults in addition to their own.

Defined in `active_support/core_ext/object/with_options.rb`.

## 2.11 JSON support

Active Support provides a better implementation of `to_json` than the `json` gem ordinarily provides for Ruby objects. This is because some classes, like `Hash`, `OrderedHash` and `Process::Status` need special handling in order to provide a proper JSON representation.

Defined in `active_support/core_ext/object/json.rb`.

## 2.12 Instance Variables

Active Support provides several methods to ease access to instance variables.

**2.12.1** `instance_values` The method `instance_values` returns a hash that maps instance variable names without "@" to their corresponding values. Keys are strings:

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

Defined in `active_support/core_ext/object/instance_variables.rb`.

**2.12.2** `instance_variable_names` The method `instance_variable_names` returns an array. Each name includes the "@" sign.

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_variable_names # => ["@x", "@y"]
```

Defined in `active_support/core_ext/object/instance_variables.rb`.

## 2.13 Silencing Warnings, Streams, and Exceptions

The methods `silence_warnings` and `enable_warnings` change the value of `$VERBOSE` accordingly for the duration of their block, and reset it afterwards:

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

You can silence any stream while a block runs with `silence_stream`:

```
silence_stream(STDOUT) do
  # STDOUT is silent here
end
```

The `quietly` method addresses the common use case where you want to silence STDOUT and STDERR, even in subprocesses:

```
quietly { system 'bundle install' }
```

For example, the railties test suite uses that one in a few places to prevent command messages from being echoed intermixed with the progress status.

Silencing exceptions is also possible with `suppress`. This method receives an arbitrary number of exception classes. If an exception is raised during the execution of the block and is `kind_of?` any of the arguments, `suppress` captures it and returns silently. Otherwise the exception is reraised:

```
# If the user is locked the increment is lost, no big deal.
suppress(ActiveRecord::StaleObjectError) do
  current_user.increment! :visits
end
```

Defined in `active_support/core_ext/kernel/reporting.rb`.

### 2.14 in?

The predicate `in?` tests if an object is included in another object. An `ArgumentError` exception will be raised if the argument passed does not respond to `include?`.

Examples of `in?`:

```
1.in?([1,2])        # => true
"lo".in?("hello")   # => true
25.in?(30..50)      # => false
1.in?(1)            # => ArgumentError
```

Defined in `active_support/core_ext/object/inclusion.rb`.

## 3 Extensions to Module

### 3.1 alias_method_chain

Using plain Ruby you can wrap methods with other methods, that's called *alias chaining*.

For example, let's say you'd like params to be strings in functional tests, as they are in real requests, but still want the convenience of assigning integers and other kind of values. To accomplish that you could wrap `ActionController::TestCase#process` this way in `test/test_helper.rb`:

```
ActionController::TestCase.class_eval do
  # save a reference to the original process method
  alias_method :original_process, :process

  # now redefine process and delegate to original_process
  def process(action, params=nil, session=nil, flash=nil, http_method='GET')
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    original_process(action, params, session, flash, http_method)
  end
end
```

That's the method `get`, `post`, etc., delegate the work to.

That technique has a risk, it could be the case that `:original_process` was taken. To try to avoid collisions people choose some label that characterizes what the chaining is about:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method :process_without_stringified_params, :process
  alias_method :process, :process_with_stringified_params
end
```

The method `alias_method_chain` provides a shortcut for that pattern:

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method_chain :process, :stringified_params
end
```

Rails uses `alias_method_chain` all over the code base. For example validations are added to `ActiveRecord::Base#save` by wrapping the method that way in a separate module specialized in validations.

Defined in `active_support/core_ext/module/aliasing.rb`.

## 3.2 Attributes

**3.2.1** `alias_attribute` Model attributes have a reader, a writer, and a predicate. You can alias a model attribute having the corresponding three methods defined for you in one shot. As in other aliasing methods, the new name is the first argument, and the old name is the second (one mnemonic is that they go in the same order as if you did an assignment):

```
class User < ActiveRecord::Base
  # You can refer to the email column as "login".
  # This can be meaningful for authentication code.
  alias_attribute :login, :email
end
```

Defined in `active_support/core_ext/module/aliasing.rb`.

**3.2.2 Internal Attributes** When you are defining an attribute in a class that is meant to be subclassed, name collisions are a risk. That's remarkably important for libraries.

Active Support defines the macros `attr_internal_reader`, `attr_internal_writer`, and `attr_internal_accessor`. They behave like their Ruby built-in `attr_*` counterparts, except they name the underlying instance variable in a way that makes collisions less likely.

The macro `attr_internal` is a synonym for `attr_internal_accessor`:

```
# library
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# client code
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

In the previous example it could be the case that `:log_level` does not belong to the public interface of the library and it is only used for development. The client code, unaware of the potential conflict, subclasses and defines its own `:log_level`. Thanks to `attr_internal` there's no collision.

By default the internal instance variable is named with a leading underscore, `@_log_level` in the example above. That's configurable via `Module.attr_internal_naming_format` though, you can pass any `sprintf`-like format string with a leading `@` and a `%s` somewhere, which is where the name will be placed. The default is `"@_%s"`.

Rails uses internal attributes in a few spots, for examples for views:

```
module ActionView
  class Base
    attr_internal :captures
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

Defined in `active_support/core_ext/module/attr_internal.rb`.

**3.2.3 Module Attributes** The macros `mattr_reader`, `mattr_writer`, and `mattr_accessor` are the same as the `cattr_*` macros defined for class. In fact, the `cattr_*` macros are just aliases for the `mattr_*` macros. Check Class Attributes.

For example, the dependencies mechanism uses them:

```
module ActiveSupport
  module Dependencies
    mattr_accessor :warnings_on_first_load
    mattr_accessor :history
    mattr_accessor :loaded
    mattr_accessor :mechanism
    mattr_accessor :load_paths
    mattr_accessor :load_once_paths
    mattr_accessor :autoloaded_constants
    mattr_accessor :explicitly_unloadable_constants
    mattr_accessor :logger
    mattr_accessor :log_activity
    mattr_accessor :constant_watch_stack
    mattr_accessor :constant_watch_stack_mutex
  end
end
```

Defined in `active_support/core_ext/module/attribute_accessors.rb`.

## 3.3 Parents

**3.3.1 `parent`** The `parent` method on a nested named module returns the module that contains its corresponding constant:

```
module X
  module Y
```

```
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent # => X::Y
M.parent       # => X::Y
```

If the module is anonymous or belongs to the top-level, `parent` returns `Object`.
Note that in that case `parent_name` returns `nil`.
Defined in `active_support/core_ext/module/introspection.rb`.

**3.3.2** `parent_name` The `parent_name` method on a nested named module returns the fully-qualified name of the module that contains its corresponding constant:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent_name # => "X::Y"
M.parent_name       # => "X::Y"
```

For top-level or anonymous modules `parent_name` returns `nil`.
Note that in that case `parent` returns `Object`.
Defined in `active_support/core_ext/module/introspection.rb`.

**3.3.3** `parents` The method `parents` calls `parent` on the receiver and upwards until `Object` is reached. The chain is returned in an array, from bottom to top:

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parents # => [X::Y, X, Object]
M.parents       # => [X::Y, X, Object]
```

Defined in `active_support/core_ext/module/introspection.rb`.

## 3.4 Constants

The method `local_constants` returns the names of the constants that have been defined in the receiver module:

```
module X
  X1 = 1
  X2 = 2
  module Y
    Y1 = :y1
    X1 = :overrides_X1_above
  end
end

X.local_constants    # => [:X1, :X2, :Y]
X::Y.local_constants # => [:Y1, :X1]
```

The names are returned as symbols.

Defined in `active_support/core_ext/module/introspection.rb`.

**3.4.1 Qualified Constant Names** The standard methods `const_defined?`, `const_get` , and `const_set` accept bare constant names. Active Support extends this API to be able to pass relative qualified constant names.

The new methods are `qualified_const_defined?`, `qualified_const_get`, and `qualified_const_set`. Their arguments are assumed to be qualified constant names relative to their receiver:

```
Object.qualified_const_defined?("Math::PI")       # => true
Object.qualified_const_get("Math::PI")            # => 3.141592653589793
Object.qualified_const_set("Math::Phi", 1.618034) # => 1.618034
```

Arguments may be bare constant names:

```
Math.qualified_const_get("E") # => 2.718281828459045
```

These methods are analogous to their built-in counterparts. In particular, `qualified_constant_defined?` accepts an optional second argument to be able to say whether you want the predicate to look in the ancestors. This flag is taken into account for each constant in the expression while walking down the path.

For example, given

```
module M
  X = 1
end

module N
  class C
    include M
  end
end
```

qualified_const_defined? behaves this way:

```
N.qualified_const_defined?("C::X", false) # => false
N.qualified_const_defined?("C::X", true)  # => true
N.qualified_const_defined?("C::X")        # => true
```

As the last example implies, the second argument defaults to true, as in const_defined?.

For coherence with the built-in methods only relative paths are accepted. Absolute qualified constant names like ::Math::PI raise NameError.

Defined in active_support/core_ext/module/qualified_const.rb.

## 3.5 Reachable

A named module is reachable if it is stored in its corresponding constant. It means you can reach the module object via the constant.

That is what ordinarily happens, if a module is called "M", the M constant exists and holds it:

```
module M
end

M.reachable? # => true
```

But since constants and modules are indeed kind of decoupled, module objects can become unreachable:

```
module M
end

orphan = Object.send(:remove_const, :M)

# The module object is orphan now but it still has a name.
orphan.name # => "M"

# You cannot reach it via the constant M because it does not even exist.
orphan.reachable? # => false

# Let's define a module called "M" again.
module M
end

# The constant M exists now again, and it stores a module
# object called "M", but it is a new instance.
orphan.reachable? # => false
```

Defined in active_support/core_ext/module/reachable.rb.

## 3.6 Anonymous

A module may or may not have a name:

```
module M
end
M.name # => "M"

N = Module.new
N.name # => "N"

Module.new.name # => nil
```

You can check whether a module has a name with the predicate `anonymous?`:

```
module M
end
M.anonymous? # => false

Module.new.anonymous? # => true
```

Note that being unreachable does not imply being anonymous:

```
module M
end

m = Object.send(:remove_const, :M)

m.reachable? # => false
m.anonymous? # => false
```

though an anonymous module is unreachable by definition.

Defined in `active_support/core_ext/module/anonymous.rb`.

## 3.7 Method Delegation

The macro `delegate` offers an easy way to forward methods.

Let's imagine that users in some application have login information in the `User` model but name and other data in a separate `Profile` model:

```
class User < ActiveRecord::Base
  has_one :profile
end
```

With that configuration you get a user's name via their profile, `user.profile.name`, but it could be handy to still be able to access such attribute directly:

```
class User < ActiveRecord::Base
  has_one :profile

  def name
    profile.name
  end
end
```

That is what `delegate` does for you:

```
class User < ActiveRecord::Base
  has_one :profile

  delegate :name, to: :profile
end
```

It is shorter, and the intention more obvious.

The method must be public in the target.

The `delegate` macro accepts several methods:

```
delegate :name, :age, :address, :twitter, to: :profile
```

When interpolated into a string, the `:to` option should become an expression that evaluates to the object the method is delegated to. Typically a string or symbol. Such an expression is evaluated in the context of the receiver:

```
# delegates to the Rails constant
delegate :logger, to: :Rails

# delegates to the receiver's class
delegate :table_name, to: :class
```

If the `:prefix` option is `true` this is less generic, see below.

By default, if the delegation raises `NoMethodError` and the target is `nil` the exception is propagated. You can ask that `nil` is returned instead with the `:allow_nil` option:

```
delegate :name, to: :profile, allow_nil: true
```

With `:allow_nil` the call `user.name` returns `nil` if the user has no profile.

The option `:prefix` adds a prefix to the name of the generated method. This may be handy for example to get a better name:

```
delegate :street, to: :address, prefix: true
```

The previous example generates `address_street` rather than `street`.

Since in this case the name of the generated method is composed of the target object and target method names, the `:to` option must be a method name.

A custom prefix may also be configured:

```
delegate :size, to: :attachment, prefix: :avatar
```

In the previous example the macro generates `avatar_size` rather than `size`.

Defined in `active_support/core_ext/module/delegation.rb`

### 3.8 Redefining Methods

There are cases where you need to define a method with `define_method`, but don't know whether a method with that name already exists. If it does, a warning is issued if they are enabled. No big deal, but not clean either.

The method `redefine_method` prevents such a potential warning, removing the existing method before if needed.

Defined in `active_support/core_ext/module/remove_method.rb`

## 4 Extensions to `Class`

### 4.1 Class Attributes

**4.1.1** `class_attribute` The method `class_attribute` declares one or more inheritable class attributes that can be overridden at any level down the hierarchy.

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b
```

For example `ActionMailer::Base` defines:

```
class_attribute :default_params
self.default_params = {
  mime_version: "1.0",
  charset: "UTF-8",
  content_type: "text/plain",
  parts_order: [ "text/plain", "text/enriched", "text/html" ]
}.freeze
```

They can also be accessed and overridden at the instance level.

```
A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, comes from A
a2.x # => 2, overridden in a2
```

The generation of the writer instance method can be prevented by setting the option `:instance_writer` to `false`.

```
module ActiveRecord
  class Base
    class_attribute :table_name_prefix, instance_writer: false
    self.table_name_prefix = ""
  end
end
```

A model may find that option useful as a way to prevent mass-assignment from setting the attribute.

The generation of the reader instance method can be prevented by setting the option `:instance_reader` to `false`.

```
class A
  class_attribute :x, instance_reader: false
end

A.new.x = 1 # NoMethodError
```

For convenience `class_attribute` also defines an instance predicate which is the double negation of what the instance reader returns. In the examples above it would be called `x?`.

When `:instance_reader` is `false`, the instance predicate returns a `NoMethodError` just like the reader method.

If you do not want the instance predicate, pass `instance_predicate: false` and it will not be defined.

Defined in `active_support/core_ext/class/attribute.rb`

**4.1.2 `cattr_reader`, `cattr_writer`, and `cattr_accessor`** The macros `cattr_reader`, `cattr_writer`, and `cattr_accessor` are analogous to their `attr_*` counterparts but for classes. They initialize a class variable to `nil` unless it already exists, and generate the corresponding class methods to access it:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans.
  cattr_accessor :emulate_booleans
  self.emulate_booleans = true
end
```

Instance methods are created as well for convenience, they are just proxies to the class attribute. So, instances can change the class attribute, but cannot override it as it happens with `class_attribute` (see above). For example given

```
module ActionView
  class Base
    cattr_accessor :field_error_proc
    @@field_error_proc = Proc.new{ ... }
  end
end
```

we can access `field_error_proc` in views.

Also, you can pass a block to `cattr_*` to set up the attribute with a default value:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans with default value of true.
  cattr_accessor(:emulate_booleans) { true }
end
```

The generation of the reader instance method can be prevented by setting `:instance_reader` to `false` and the generation of the writer instance method can be prevented by setting `:instance_writer` to `false`. Generation of both methods can be prevented by setting `:instance_accessor` to `false`. In all cases, the value must be exactly `false` and not any false value.

```
module A
  class B
    # No first_name instance reader is generated.
    cattr_accessor :first_name, instance_reader: false
    # No last_name= instance writer is generated.
    cattr_accessor :last_name, instance_writer: false
    # No surname instance reader or surname= writer is generated.
    cattr_accessor :surname, instance_accessor: false
  end
end
```

A model may find it useful to set `:instance_accessor` to `false` as a way to prevent mass-assignment from setting the attribute.

Defined in `active_support/core_ext/module/attribute_accessors.rb`.

## 4.2 Subclasses & Descendants

**4.2.1** `subclasses` The `subclasses` method returns the subclasses of the receiver:

```
class C; end
C.subclasses # => []

class B < C; end
```

```
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

class D < C; end
C.subclasses # => [B, D]
```

The order in which these classes are returned is unspecified.
Defined in `active_support/core_ext/class/subclasses.rb`.

**4.2.2** `descendants` The `descendants` method returns all classes that are < than its receiver:

```
class C; end
C.descendants # => []

class B < C; end
C.descendants # => [B]

class A < B; end
C.descendants # => [B, A]

class D < C; end
C.descendants # => [B, A, D]
```

The order in which these classes are returned is unspecified.
Defined in `active_support/core_ext/class/subclasses.rb`.

# 5 Extensions to `String`

## 5.1 Output Safety

**5.1.1 Motivation** Inserting data into HTML templates needs extra care. For example, you can't just interpolate `@review.title` verbatim into an HTML page. For one thing, if the review title is "Flanagan & Matz rules!" the output won't be well-formed because an ampersand has to be escaped as "&". What's more, depending on the application, that may be a big security hole because users can inject malicious HTML setting a hand-crafted review title. Check out the section about cross-site scripting in the Security guide for further information about the risks.

**5.1.2 Safe Strings** Active Support has the concept of *(html) safe* strings. A safe string is one that is marked as being insertable into HTML as is. It is trusted, no matter whether it has been escaped or not.

Strings are considered to be *unsafe* by default:

```
"".html_safe? # => false
```

You can obtain a safe string from a given one with the `html_safe` method:

```
s = "".html_safe
s.html_safe? # => true
```

It is important to understand that `html_safe` performs no escaping whatsoever, it is just an assertion:

```
s = "<script>...</script>".html_safe
s.html_safe? # => true
s            # => "<script>...</script>"
```

It is your responsibility to ensure calling `html_safe` on a particular string is fine.

If you append onto a safe string, either in-place with `concat`/<<, or with `+`, the result is a safe string. Unsafe arguments are escaped:

```
"".html_safe + "<" # => "&lt;"
```

Safe arguments are directly appended:

```
"".html_safe + "<".html_safe # => "<"
```

These methods should not be used in ordinary views. Unsafe values are automatically escaped:

```
<%= @review.title %> <%# fine, escaped if needed %>
```

To insert something verbatim use the `raw` helper rather than calling `html_safe`:

```
<%= raw @cms.current_template %> <%# inserts @cms.current_template as is %>
```

or, equivalently, use <%==:

```
<%== @cms.current_template %> <%# inserts @cms.current_template as is %>
```

The `raw` helper calls `html_safe` for you:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

Defined in `active_support/core_ext/string/output_safety.rb`.

**5.1.3 Transformation** As a rule of thumb, except perhaps for concatenation as explained above, any method that may change a string gives you an unsafe string. These are `downcase`, `gsub`, `strip`, `chomp`, `underscore`, etc.

In the case of in-place transformations like `gsub!` the receiver itself becomes unsafe.

The safety bit is lost always, no matter whether the transformation actually changed something.

**5.1.4 Conversion and Coercion** Calling `to_s` on a safe string returns a safe string, but coercion with `to_str` returns an unsafe string.

**5.1.5 Copying** Calling `dup` or `clone` on safe strings yields safe strings.

## 5.2 remove

The method `remove` will remove all occurrences of the pattern:

```
"Hello World".remove(/Hello /) => "World"
```

There's also the destructive version `String#remove!`.
Defined in `active_support/core_ext/string/filters.rb`.

## 5.3 squish

The method `squish` strips leading and trailing whitespace, and substitutes runs of whitespace with a single space each:

```
" \n  foo\n\r \t bar \n".squish # => "foo bar"
```

There's also the destructive version `String#squish!`.
Note that it handles both ASCII and Unicode whitespace.
Defined in `active_support/core_ext/string/filters.rb`.

## 5.4 truncate

The method `truncate` returns a copy of its receiver truncated after a given `length`:

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the `:omission` option:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, omission: '&hellip;')
# => "Oh dear! Oh &hellip;"
```

Note in particular that truncation takes into account the length of the omission string.
Pass a `:separator` to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
# => "Oh dear! Oh dea..."
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: ' ')
# => "Oh dear! Oh..."
```

The option `:separator` can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: /\s/)
# => "Oh dear! Oh..."
```

In above examples "dear" gets cut first, but then `:separator` prevents it.
Defined in `active_support/core_ext/string/filters.rb`.

## 5.5 truncate_words

The method truncate_words returns a copy of its receiver truncated after a given number of words:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the :omission option:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, omission: '&hellip;')
# => "Oh dear! Oh dear!&hellip;"
```

Pass a :separator to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(3, separator: '!')
# => "Oh dear! Oh dear! I shall be late..."
```

The option :separator can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, separator: /\s/)
# => "Oh dear! Oh dear!..."
```

Defined in active_support/core_ext/string/filters.rb.

## 5.6 inquiry

The inquiry method converts a string into a StringInquirer object making equality checks prettier.

```
"production".inquiry.production? # => true
"active".inquiry.inactive?       # => false
```

## 5.7 starts_with? and ends_with?

Active Support defines 3rd person aliases of String#start_with? and String#end_with?:

```
"foo".starts_with?("f") # => true
"foo".ends_with?("o")   # => true
```

Defined in active_support/core_ext/string/starts_ends_with.rb.

## 5.8 strip_heredoc

The method strip_heredoc strips indentation in heredocs.

For example in

```
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h         This message
      ...
  USAGE
end
```

the user would see the usage message aligned against the left margin.

Technically, it looks for the least indented line in the whole string, and removes that amount of leading whitespace.

Defined in `active_support/core_ext/string/strip.rb`.

## 5.9 indent

Indents the lines in the receiver:

```
<<EOS.indent(2)
def some_method
  some_code
end
EOS
# =>
  def some_method
    some_code
  end
```

The second argument, `indent_string`, specifies which indent string to use. The default is `nil`, which tells the method to make an educated guess peeking at the first indented line, and fallback to a space if there is none.

```
"  foo".indent(2)        # => "    foo"
"foo\n\t\tbar".indent(2) # => "\t\tfoo\n\t\t\t\tbar"
"foo".indent(2, "\t")    # => "\t\tfoo"
```

While `indent_string` is typically one space or tab, it may be any string.

The third argument, `indent_empty_lines`, is a flag that says whether empty lines should be indented. Default is false.

```
"foo\n\nbar".indent(2)            # => "  foo\n\n  bar"
"foo\n\nbar".indent(2, nil, true) # => "  foo\n  \n  bar"
```

The `indent!` method performs indentation in-place.

Defined in `active_support/core_ext/string/indent.rb`.

## 5.10 Access

**5.10.1** `at(position)` Returns the character of the string at position `position`:

```
"hello".at(0)  # => "h"
"hello".at(4)  # => "o"
"hello".at(-1) # => "o"
"hello".at(10) # => nil
```

Defined in `active_support/core_ext/string/access.rb`.

**5.10.2** `from(position)` Returns the substring of the string starting at position `position`:

```
"hello".from(0)  # => "hello"
"hello".from(2)  # => "llo"
"hello".from(-2) # => "lo"
"hello".from(10) # => "" if < 1.9, nil in 1.9
```

Defined in `active_support/core_ext/string/access.rb`.

**5.10.3** `to(position)` Returns the substring of the string up to position `position`:

```
"hello".to(0)  # => "h"
"hello".to(2)  # => "hel"
"hello".to(-2) # => "hell"
"hello".to(10) # => "hello"
```

Defined in `active_support/core_ext/string/access.rb`.

**5.10.4** `first(limit = 1)` The call `str.first(n)` is equivalent to `str.to(n-1)` if `n` $> 0$, and returns an empty string for `n` $== 0$.

Defined in `active_support/core_ext/string/access.rb`.

**5.10.5** `last(limit = 1)` The call `str.last(n)` is equivalent to `str.from(-n)` if `n` $> 0$, and returns an empty string for `n` $== 0$.

Defined in `active_support/core_ext/string/access.rb`.

## 5.11 Inflections

**5.11.1** `pluralize` The method `pluralize` returns the plural of its receiver:

```
"table".pluralize     # => "tables"
"ruby".pluralize      # => "rubies"
"equipment".pluralize # => "equipment"
```

As the previous example shows, Active Support knows some irregular plurals and uncountable nouns. Built-in rules can be extended in `config/initializers/inflections.rb`. That file is generated by the `rails` command and has instructions in comments.

`pluralize` can also take an optional `count` parameter. If `count == 1` the singular form will be returned. For any other value of `count` the plural form will be returned:

```
"dude".pluralize(0) # => "dudes"
"dude".pluralize(1) # => "dude"
"dude".pluralize(2) # => "dudes"
```

Active Record uses this method to compute the default table name that corresponds to a model:

```
# active_record/model_schema.rb
def undecorated_table_name(class_name = base_class.name)
  table_name = class_name.to_s.demodulize.underscore
  pluralize_table_names ? table_name.pluralize : table_name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.2 `singularize`** The inverse of `pluralize`:

```
"tables".singularize    # => "table"
"rubies".singularize    # => "ruby"
"equipment".singularize # => "equipment"
```

Associations compute the name of the corresponding default associated class using this method:

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.3 `camelize`** The method `camelize` returns its receiver in camel case:

```
"product".camelize    # => "Product"
"admin_user".camelize # => "AdminUser"
```

As a rule of thumb you can think of this method as the one that transforms paths into Ruby class or module names, where slashes separate namespaces:

```
"backoffice/session".camelize # => "Backoffice::Session"
```

For example, Action Pack uses this method to load the class that provides a certain session store:

```
# action_controller/metal/session_management.rb
def session_store=(store)
  @@session_store = store.is_a?(Symbol) ?
    ActionDispatch::Session.const_get(store.to_s.camelize) :
    store
end
```

camelize accepts an optional argument, it can be :upper (default), or :lower. With the latter the first letter becomes lowercase:

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

That may be handy to compute method names in a language that follows that convention, for example JavaScript.

As a rule of thumb you can think of camelize as the inverse of underscore, though there are cases where that does not hold: "SSLError".underscore.camelize gives back "SslError". To support cases such as this, Active Support allows you to specify acronyms in config/initializers/inflections.rb:

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end

"SSLError".underscore.camelize # => "SSLError"
```

camelize is aliased to camelcase.

Defined in active_support/core_ext/string/inflections.rb.

**5.11.4 underscore** The method underscore goes the other way around, from camel case to paths:

```
"Product".underscore   # => "product"
"AdminUser".underscore # => "admin_user"
```

Also converts "::" back to "/":

```
"Backoffice::Session".underscore # => "backoffice/session"
```

and understands strings that start with lowercase:

```
"visualEffect".underscore # => "visual_effect"
```

underscore accepts no argument though.

Rails class and module autoloading uses underscore to infer the relative path without extension of a file that would define a given missing constant:

```
# active_support/dependencies.rb
def load_missing_constant(from_mod, const_name)
  ...
  qualified_name = qualified_name_for from_mod, const_name
  path_suffix = qualified_name.underscore
  ...
end
```

As a rule of thumb you can think of `underscore` as the inverse of `camelize`, though there are cases where that does not hold. For example, `"SSLError".underscore.camelize` gives back `"SslError"`.

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.5 `titleize`** The method `titleize` capitalizes the words in the receiver:

```
"alice in wonderland".titleize # => "Alice In Wonderland"
"fermat's enigma".titleize     # => "Fermat's Enigma"
```

`titleize` is aliased to `titlecase`.

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.6 `dasherize`** The method `dasherize` replaces the underscores in the receiver with dashes:

```
"name".dasherize         # => "name"
"contact_data".dasherize # => "contact-data"
```

The XML serializer of models uses this method to dasherize node names:

```
# active_model/serializers/xml.rb
def reformat_name(name)
  name = name.camelize if camelize?
  dasherize? ? name.dasherize : name
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.7 `demodulize`** Given a string with a qualified constant name, `demodulize` returns the very constant name, that is, the rightmost part of it:

```
"Product".demodulize                        # => "Product"
"Backoffice::UsersController".demodulize    # => "UsersController"
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"
"::Inflections".demodulize                  # => "Inflections"
"".demodulize                               # => ""
```

Active Record for example uses this method to compute the name of a counter cache column:

```
# active_record/reflection.rb
def counter_cache_column
  if options[:counter_cache] == true
    "#{active_record.name.demodulize.underscore.pluralize}_count"
  elsif options[:counter_cache]
    options[:counter_cache]
  end
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.8** `deconstantize` Given a string with a qualified constant reference expression, `deconstantize` removes the rightmost segment, generally leaving the name of the constant's container:

```
"Product".deconstantize                         # => ""
"Backoffice::UsersController".deconstantize     # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

Active Support for example uses this method in `Module#qualified_const_set`:

```
def qualified_const_set(path, value)
  QualifiedConstUtils.raise_if_absolute(path)

  const_name = path.demodulize
  mod_name = path.deconstantize
  mod = mod_name.empty? ? self : qualified_const_get(mod_name)
  mod.const_set(const_name, value)
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.9** `parameterize` The method `parameterize` normalizes its receiver in a way that can be used in pretty URLs.

```
"John Smith".parameterize # => "john-smith"
"Kurt G^^c3^^b6del".parameterize # => "kurt-godel"
```

In fact, the result string is wrapped in an instance of `ActiveSupport::Multibyte::Chars`.
Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.10** `tableize` The method `tableize` is `underscore` followed by `pluralize`.

```
"Person".tableize      # => "people"
"Invoice".tableize     # => "invoices"
"InvoiceLine".tableize # => "invoice_lines"
```

As a rule of thumb, `tableize` returns the table name that corresponds to a given model for simple cases. The actual implementation in Active Record is not straight `tableize` indeed, because it also demodulizes the class name and checks a few options that may affect the returned string.
Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.11** `classify` The method `classify` is the inverse of `tableize`. It gives you the class name corresponding to a table name:

```
"people".classify        # => "Person"
"invoices".classify      # => "Invoice"
"invoice_lines".classify # => "InvoiceLine"
```

The method understands qualified table names:

```
"highrise_production.companies".classify # => "Company"
```

Note that `classify` returns a class name as a string. You can get the actual class object invoking `constantize` on it, explained next.

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.12 `constantize`** The method `constantize` resolves the constant reference expression in its receiver:

```
"Fixnum".constantize # => Fixnum

module M
  X = 1
end
"M::X".constantize # => 1
```

If the string evaluates to no known constant, or its content is not even a valid constant name, `constantize` raises `NameError`.

Constant name resolution by `constantize` starts always at the top-level `Object` even if there is no leading "::".

```
X = :in_Object
module M
  X = :in_M

  X                 # => :in_M
  "::X".constantize # => :in_Object
  "X".constantize   # => :in_Object (!)
end
```

So, it is in general not equivalent to what Ruby would do in the same spot, had a real constant be evaluated.

Mailer test cases obtain the mailer being tested from the name of the test class using `constantize`:

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.sub(/Test$/, '').constantize
rescue NameError => e
  raise NonInferrableMailerError.new(name)
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.13 `humanize`** The method `humanize` tweaks an attribute name for display to end users.

Specifically performs these transformations:

- Applies human inflection rules to the argument.
- Deletes leading underscores, if any.
- Removes a "_id" suffix if present.

- Replaces underscores with spaces, if any.
- Downcases all words except acronyms.
- Capitalizes the first word.

The capitalization of the first word can be turned off by setting the +:capitalize+ option to false (default is true).

```
"name".humanize                          # => "Name"
"author_id".humanize                     # => "Author"
"author_id".humanize(capitalize: false) # => "author"
"comments_count".humanize                # => "Comments count"
"_id".humanize                           # => "Id"
```

If "SSL" was defined to be an acronym:

```
'ssl_error'.humanize # => "SSL error"
```

The helper method `full_messages` uses `humanize` as a fallback to include attribute names:

```
def full_messages
  map { |attribute, message| full_message(attribute, message) }
end

def full_message
  ...
  attr_name = attribute.to_s.tr('.', '_').humanize
  attr_name = @base.class.human_attribute_name(attribute, default: attr_name)
  ...
end
```

Defined in `active_support/core_ext/string/inflections.rb`.

**5.11.14** `foreign_key` The method `foreign_key` gives a foreign key column name from a class name. To do so it demodulizes, underscores, and adds "_id":

```
"User".foreign_key           # => "user_id"
"InvoiceLine".foreign_key    # => "invoice_line_id"
"Admin::Session".foreign_key # => "session_id"
```

Pass a false argument if you do not want the underscore in "_id":

```
"User".foreign_key(false) # => "userid"
```

Associations use this method to infer foreign keys, for example `has_one` and `has_many` do this:

```
# active_record/associations.rb
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

Defined in `active_support/core_ext/string/inflections.rb`.

### 5.12 Conversions

**5.12.1** `to_date`, `to_time`, `to_datetime` The methods `to_date`, `to_time`, and `to_datetime` are basically convenience wrappers around `Date._parse`:

```
"2010-07-27".to_date              # => Tue, 27 Jul 2010
"2010-07-27 23:37:00".to_time     # => Tue Jul 27 23:37:00 UTC 2010
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` receives an optional argument `:utc` or `:local`, to indicate which time zone you want the time in:

```
"2010-07-27 23:42:00".to_time(:utc)   # => Tue Jul 27 23:42:00 UTC 2010
"2010-07-27 23:42:00".to_time(:local) # => Tue Jul 27 23:42:00 +0200 2010
```

Default is `:utc`.
Please refer to the documentation of `Date._parse` for further details.
The three of them return `nil` for blank receivers.
Defined in `active_support/core_ext/string/conversions.rb`.

## 6 Extensions to `Numeric`

### 6.1 Bytes

All numbers respond to these methods:

```
bytes
kilobytes
megabytes
gigabytes
terabytes
petabytes
exabytes
```

They return the corresponding amount of bytes, using a conversion factor of 1024:

```
2.kilobytes   # => 2048
3.megabytes   # => 3145728
3.5.gigabytes # => 3758096384
-4.exabytes   # => -4611686018427387904
```

Singular forms are aliased so you are able to say:

```
1.megabyte # => 1048576
```

Defined in `active_support/core_ext/numeric/bytes.rb`.

## 6.2 Time

Enables the use of time calculations and declarations, like `45.minutes + 2.hours + 4.years`.

These methods use Time#advance for precise date calculations when using from_now, ago, etc. as well as adding or subtracting their results from a Time object. For example:

```
# equivalent to Time.current.advance(months: 1)
1.month.from_now

# equivalent to Time.current.advance(years: 2)
2.years.from_now

# equivalent to Time.current.advance(months: 4, years: 5)
(4.months + 5.years).from_now
```

## 6.3 Formatting

Enables the formatting of numbers in a variety of ways.

Produce a string representation of a number as a telephone number:

```
5551234.to_s(:phone)
# => 555-1234
1235551234.to_s(:phone)
# => 123-555-1234
1235551234.to_s(:phone, area_code: true)
# => (123) 555-1234
1235551234.to_s(:phone, delimiter: " ")
# => 123 555 1234
1235551234.to_s(:phone, area_code: true, extension: 555)
# => (123) 555-1234 x 555
1235551234.to_s(:phone, country_code: 1)
# => +1-123-555-1234
```

Produce a string representation of a number as currency:

```
1234567890.50.to_s(:currency)                 # => $1,234,567,890.50
1234567890.506.to_s(:currency)                # => $1,234,567,890.51
1234567890.506.to_s(:currency, precision: 3)  # => $1,234,567,890.506
```

Produce a string representation of a number as a percentage:

```
100.to_s(:percentage)
# => 100.000%
100.to_s(:percentage, precision: 0)
# => 100%
1000.to_s(:percentage, delimiter: '.', separator: ',')
# => 1.000,000%
302.24398923423.to_s(:percentage, precision: 5)
# => 302.24399%
```

Produce a string representation of a number in delimited form:

```
12345678.to_s(:delimited)                     # => 12,345,678
12345678.05.to_s(:delimited)                  # => 12,345,678.05
12345678.to_s(:delimited, delimiter: ".")     # => 12.345.678
12345678.to_s(:delimited, delimiter: ",")     # => 12,345,678
12345678.05.to_s(:delimited, separator: " ")  # => 12,345,678 05
```

Produce a string representation of a number rounded to a precision:

```
111.2345.to_s(:rounded)                     # => 111.235
111.2345.to_s(:rounded, precision: 2)       # => 111.23
13.to_s(:rounded, precision: 5)             # => 13.00000
389.32314.to_s(:rounded, precision: 0)      # => 389
111.2345.to_s(:rounded, significant: true)  # => 111
```

Produce a string representation of a number as a human-readable number of bytes:

```
123.to_s(:human_size)            # => 123 Bytes
1234.to_s(:human_size)           # => 1.21 KB
12345.to_s(:human_size)          # => 12.1 KB
1234567.to_s(:human_size)        # => 1.18 MB
1234567890.to_s(:human_size)     # => 1.15 GB
1234567890123.to_s(:human_size)  # => 1.12 TB
```

Produce a string representation of a number in human-readable words:

```
123.to_s(:human)               # => "123"
1234.to_s(:human)              # => "1.23 Thousand"
12345.to_s(:human)             # => "12.3 Thousand"
1234567.to_s(:human)           # => "1.23 Million"
1234567890.to_s(:human)        # => "1.23 Billion"
1234567890123.to_s(:human)     # => "1.23 Trillion"
1234567890123456.to_s(:human)  # => "1.23 Quadrillion"
```

Defined in `active_support/core_ext/numeric/conversions.rb`.

# 7 Extensions to `Integer`

## 7.1 `multiple_of?`

The method `multiple_of?` tests whether an integer is multiple of the argument:

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

Defined in `active_support/core_ext/integer/multiple.rb`.

### 7.2 ordinal

The method `ordinal` returns the ordinal suffix string corresponding to the receiver integer:

```
1.ordinal    # => "st"
2.ordinal    # => "nd"
53.ordinal   # => "rd"
2009.ordinal # => "th"
-21.ordinal  # => "st"
-134.ordinal # => "th"
```

Defined in `active_support/core_ext/integer/inflections.rb`.

### 7.3 ordinalize

The method `ordinalize` returns the ordinal string corresponding to the receiver integer. In comparison, note that the `ordinal` method returns **only** the suffix string.

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
53.ordinalize   # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize  # => "-21st"
-134.ordinalize # => "-134th"
```

Defined in `active_support/core_ext/integer/inflections.rb`.

## 8 Extensions to BigDecimal

### 8.1 to_s

The method `to_s` is aliased to `to_formatted_s`. This provides a convenient way to display a BigDecimal value in floating-point notation:

```
BigDecimal.new(5.00, 6).to_s  # => "5.0"
```

### 8.2 to_formatted_s

Te method `to_formatted_s` provides a default specifier of "F". This means that a simple call to `to_formatted_s` or `to_s` will result in floating point representation instead of engineering notation:

```
BigDecimal.new(5.00, 6).to_formatted_s  # => "5.0"
```

and that symbol specifiers are also supported:

```
BigDecimal.new(5.00, 6).to_formatted_s(:db)  # => "5.0"
```

Engineering notation is still supported:

```
BigDecimal.new(5.00, 6).to_formatted_s("e")  # => "0.5E1"
```

# 9 Extensions to Enumerable

## 9.1 sum

The method sum adds the elements of an enumerable:

```
[1, 2, 3].sum # => 6
(1..100).sum  # => 5050
```

Addition only assumes the elements respond to +:

```
[[1, 2], [2, 3], [3, 4]].sum    # => [1, 2, 2, 3, 3, 4]
%w(foo bar baz).sum             # => "foobarbaz"
{a: 1, b: 2, c: 3}.sum # => [:b, 2, :c, 3, :a, 1]
```

The sum of an empty collection is zero by default, but this is customizable:

```
[].sum    # => 0
[].sum(1) # => 1
```

If a block is given, sum becomes an iterator that yields the elements of the collection and sums the returned values:

```
(1..5).sum {|n| n * 2 } # => 30
[2, 4, 6, 8, 10].sum    # => 30
```

The sum of an empty receiver can be customized in this form as well:

```
[].sum(1) {|n| n**3} # => 1
```

Defined in active_support/core_ext/enumerable.rb.

## 9.2 index_by

The method index_by generates a hash with the elements of an enumerable indexed by some key.

It iterates through the collection and passes each element to a block. The element will be keyed by the value returned by the block:

```
invoices.index_by(&:number)
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

Keys should normally be unique. If the block returns the same value for different elements no collection is built for that key. The last item will win.

Defined in active_support/core_ext/enumerable.rb.

### 9.3 many?

The method many? is shorthand for collection.size > 1:

```
<% if pages.many? %>
  <%= pagination_links %>
<% end %>
```

If an optional block is given, many? only takes into account those elements that return true:

```
@see_more = videos.many? {|video| video.category == params[:category]}
```

Defined in active_support/core_ext/enumerable.rb.

### 9.4 exclude?

The predicate exclude? tests whether a given object does **not** belong to the collection. It is the negation of the built-in include?:

```
to_visit << node if visited.exclude?(node)
```

Defined in active_support/core_ext/enumerable.rb.

## 10 Extensions to Array

### 10.1 Accessing

Active Support augments the API of arrays to ease certain ways of accessing them. For example, to returns the subarray of elements up to the one at the passed index:

```
%w(a b c d).to(2) # => %w(a b c)
[].to(7)          # => []
```

Similarly, from returns the tail from the element at the passed index to the end. If the index is greater than the length of the array, it returns an empty array.

```
%w(a b c d).from(2)  # => %w(c d)
%w(a b c d).from(10) # => []
[].from(0)           # => []
```

The methods second, third, fourth, and fifth return the corresponding element (first is built-in). Thanks to social wisdom and positive constructiveness all around, forty_two is also available.

```
%w(a b c d).third # => c
%w(a b c d).fifth # => nil
```

Defined in active_support/core_ext/array/access.rb.

## 10.2 Adding Elements

**10.2.1** `prepend` This method is an alias of `Array#unshift`.

```
%w(a b c d).prepend('e')  # => %w(e a b c d)
[].prepend(10)            # => [10]
```

Defined in `active_support/core_ext/array/prepend_and_append.rb`.

**10.2.2** `append` This method is an alias of `Array#<<`.

```
%w(a b c d).append('e')  # => %w(a b c d e)
[].append([1,2])         # => [[1,2]]
```

Defined in `active_support/core_ext/array/prepend_and_append.rb`.

## 10.3 Options Extraction

When the last argument in a method call is a hash, except perhaps for a `&block` argument, Ruby allows you to omit the brackets:

```
User.exists?(email: params[:email])
```

That syntactic sugar is used a lot in Rails to avoid positional arguments where there would be too many, offering instead interfaces that emulate named parameters. In particular it is very idiomatic to use a trailing hash for options.

If a method expects a variable number of arguments and uses `*` in its declaration, however, such an options hash ends up being an item of the array of arguments, where it loses its role.

In those cases, you may give an options hash a distinguished treatment with `extract_options!`. This method checks the type of the last item of an array. If it is a hash it pops it and returns it, otherwise it returns an empty hash.

Let's see for example the definition of the `caches_action` controller macro:

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  ...
end
```

This method receives an arbitrary number of action names, and an optional hash of options as last argument. With the call to `extract_options!` you obtain the options hash and remove it from `actions` in a simple and explicit way.

Defined in `active_support/core_ext/array/extract_options.rb`.

## 10.4 Conversions

**10.4.1** `to_sentence` The method `to_sentence` turns an array into a string containing a sentence that enumerates its items:

```
%w().to_sentence                # => ""
%w(Earth).to_sentence           # => "Earth"
%w(Earth Wind).to_sentence      # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

This method accepts three options:

- `:two_words_connector`: What is used for arrays of length 2. Default is " and ".
- `:words_connector`: What is used to join the elements of arrays with 3 or more elements, except for the last two. Default is ",".
- `:last_word_connector`: What is used to join the last items of an array with 3 or more elements. Default is ", and".

The defaults for these options can be localized, their keys are:

| Option | I18n key |
|---|---|
| `:two_words_connector` | `support.array.two_words_connector` |
| `:words_connector` | `support.array.words_connector` |
| `:last_word_connector` | `support.array.last_word_connector` |

Defined in `active_support/core_ext/array/conversions.rb`.

**10.4.2** `to_formatted_s` The method `to_formatted_s` acts like `to_s` by default.

If the array contains items that respond to `id`, however, the symbol `:db` may be passed as argument. That's typically used with collections of Active Record objects. Returned strings are:

```
[].to_formatted_s(:db)            # => "null"
[user].to_formatted_s(:db)        # => "8456"
invoice.lines.to_formatted_s(:db) # => "23,567,556,12"
```

Integers in the example above are supposed to come from the respective calls to `id`.

Defined in `active_support/core_ext/array/conversions.rb`.

**10.4.3** `to_xml` The method `to_xml` returns a string containing an XML representation of its receiver:

```
Contributor.limit(2).order(:rank).to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors type="array">
```

```
#   <contributor>
#     <id type="integer">4356</id>
#     <name>Jeremy Kemper</name>
#     <rank type="integer">1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id type="integer">4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank type="integer">2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

To do so it sends `to_xml` to every item in turn, and collects the results under a root node. All items must respond to `to_xml`, an exception is raised otherwise.

By default, the name of the root element is the underscorized and dasherized plural of the name of the class of the first item, provided the rest of elements belong to that type (checked with `is_a?`) and they are not hashes. In the example above that's "contributors".

If there's any element that does not belong to the type of the first one the root node becomes "objects":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </object>
#   <object>
#     <author>Joshua Peek</author>
#     <authored-timestamp type="datetime">2009-09-02T16:44:36Z</
authored-timestamp>
#     <branch>origin/master</branch>
#     <committed-timestamp type="datetime">2009-09-02T16:44:36Z</
committed-timestamp>
#     <committer>Joshua Peek</committer>
#     <git-show nil="true"></git-show>
#     <id type="integer">190316</id>
#     <imported-from-svn type="boolean">false</imported-from-svn>
#     <message>Kill AMo observing wrap_with_notifications since ARes was only using
it</message>
#     <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#   </object>
# </objects>
```

If the receiver is an array of hashes the root element is by default also "objects":

```
[{a: 1, b: 2}, {c: 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </object>
#   <object>
#     <c type="integer">3</c>
#   </object>
# </objects>
```

If the collection is empty the root element is by default "nil-classes". That's a gotcha, for example the root element of the list of contributors above would not be "contributors" if the collection was empty, but "nil-classes". You may use the `:root` option to ensure a consistent root element.

The name of children nodes is by default the name of the root node singularized. In the examples above we've seen "contributor" and "object". The option `:children` allows you to set these node names.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder via the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder:

```
Contributor.limit(2).order(:rank).to_xml(skip_types: true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>
#     <name>Jeremy Kemper</name>
#     <rank>1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id>4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank>2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

Defined in `active_support/core_ext/array/conversions.rb`.

## 10.5 Wrapping

The method `Array.wrap` wraps its argument in an array unless it is already an array (or array-like).

Specifically:

- If the argument is `nil` an empty list is returned.
- Otherwise, if the argument responds to `to_ary` it is invoked, and if the value of `to_ary` is not `nil`, it is returned.
- Otherwise, an array with the argument as its single element is returned.

```
Array.wrap(nil)       # => []
Array.wrap([1, 2, 3]) # => [1, 2, 3]
Array.wrap(0)         # => [0]
```

This method is similar in purpose to `Kernel#Array`, but there are some differences:

- If the argument responds to `to_ary` the method is invoked. `Kernel#Array` moves on to try `to_a` if the returned value is `nil`, but `Array.wrap` returns `nil` right away.
- If the returned value from `to_ary` is neither `nil` nor an `Array` object, `Kernel#Array` raises an exception, while `Array.wrap` does not, it just returns the value.
- It does not call `to_a` on the argument, though special-cases `nil` to return an empty array.

The last point is particularly worth comparing for some enumerables:

```
Array.wrap(foo: :bar) # => [{:foo=>:bar}]
Array(foo: :bar)      # => [[:foo, :bar]]
```

There's also a related idiom that uses the splat operator:

```
[*object]
```

which in Ruby 1.8 returns `[nil]` for `nil`, and calls to `Array(object)` otherwise. (Please if you know the exact behavior in 1.9 contact fxn.)

Thus, in this case the behavior is different for `nil`, and the differences with `Kernel#Array` explained above apply to the rest of `objects`.

Defined in `active_support/core_ext/array/wrap.rb`.

## 10.6 Duplicating

The method `Array.deep_dup` duplicates itself and all objects inside recursively with Active Support method `Object#deep_dup`. It works like `Array#map` with sending `deep_dup` method to each object inside.

```
array = [1, [2, 3]]
dup = array.deep_dup
dup[1][2] = 4
array[1][2] == nil   # => true
```

Defined in `active_support/core_ext/object/deep_dup.rb`.

## 10.7 Grouping

**10.7.1** `in_groups_of(number, fill_with = nil)` The method `in_groups_of` splits an array into consecutive groups of a certain size. It returns an array with the groups:

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

or yields them in turn if a block is passed:

```
<% sample.in_groups_of(3) do |a, b, c| %>
  <tr>
    <td><%= a %></td>
    <td><%= b %></td>
    <td><%= c %></td>
  </tr>
<% end %>
```

The first example shows `in_groups_of` fills the last group with as many `nil` elements as needed to have the requested size. You can change this padding value using the second optional argument:

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

And you can tell the method not to fill the last group passing `false`:

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

As a consequence `false` can't be a used as a padding value.
Defined in `active_support/core_ext/array/grouping.rb`.

**10.7.2** `in_groups(number, fill_with = nil)` The method `in_groups` splits an array into a certain number of groups. The method returns an array with the groups:

```
%w(1 2 3 4 5 6 7).in_groups(3)
# => [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

or yields them in turn if a block is passed:

```
%w(1 2 3 4 5 6 7).in_groups(3) {|group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

The examples above show that `in_groups` fills some groups with a trailing `nil` element as needed. A group can get at most one of these extra elements, the rightmost one if any. And the groups that have them are always the last ones.

You can change this padding value using the second optional argument:

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
# => [["1", "2", "3"], ["4", "5", "0"], ["6", "7", "0"]]
```

And you can tell the method not to fill the smaller groups passing `false`:

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
# => [["1", "2", "3"], ["4", "5"], ["6", "7"]]
```

As a consequence `false` can't be a used as a padding value.
Defined in `active_support/core_ext/array/grouping.rb`.

**10.7.3** `split(value = nil)` The method `split` divides an array by a separator and returns the resulting chunks.

If a block is passed the separators are those elements of the array for which the block returns true:

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }
# => [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

Otherwise, the value received as argument, which defaults to `nil`, is the separator:

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)
# => [[0], [-5], [], ["foo", "bar"]]
```

Observe in the previous example that consecutive separators result in empty arrays.
Defined in `active_support/core_ext/array/grouping.rb`.

# 11 Extensions to `Hash`

## 11.1 Conversions

**11.1.1** `to_xml` The method `to_xml` returns a string containing an XML representation of its receiver:

```
{"foo" => 1, "bar" => 2}.to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <hash>
#   <foo type="integer">1</foo>
#   <bar type="integer">2</bar>
# </hash>
```

To do so, the method loops over the pairs and builds nodes that depend on the *values*. Given a pair `key`, `value`:

- If `value` is a hash there's a recursive call with `key` as `:root`.
- If `value` is an array there's a recursive call with `key` as `:root`, and `key` singularized as `:children`.
- If `value` is a callable object it must expect one or two arguments. Depending on the arity, the callable is invoked with the `options` hash as first argument with `key` as `:root`, and `key` singularized as second argument. Its return value becomes a new node.

- If `value` responds to `to_xml` the method is invoked with `key` as `:root`.
- Otherwise, a node with `key` as tag is created with a string representation of `value` as text node. If `value` is `nil` an attribute "nil" set to "true" is added. Unless the option `:skip_types` exists and is true, an attribute "type" is added as well according to the following mapping:

```
XML_TYPE_NAMES = {
  "Symbol"     => "symbol",
  "Fixnum"     => "integer",
  "Bignum"     => "integer",
  "BigDecimal" => "decimal",
  "Float"      => "float",
  "TrueClass"  => "boolean",
  "FalseClass" => "boolean",
  "Date"       => "date",
  "DateTime"   => "datetime",
  "Time"       => "datetime"
}
```

By default the root node is "hash", but that's configurable via the `:root` option.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder with the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder.

Defined in `active_support/core_ext/hash/conversions.rb`.

## 11.2 Merging

Ruby has a built-in method `Hash#merge` that merges two hashes:

```
{a: 1, b: 1}.merge(a: 0, c: 2)
# => {:a=>0, :b=>1, :c=>2}
```

Active Support defines a few more ways of merging hashes that may be convenient.

**11.2.1 `reverse_merge` and `reverse_merge!`** In case of collision the key in the hash of the argument wins in `merge`. You can support option hashes with default values in a compact way with this idiom:

```
options = {length: 30, omission: "..."}.merge(options)
```

Active Support defines `reverse_merge` in case you prefer this alternative notation:

```
options = options.reverse_merge(length: 30, omission: "...")
```

And a bang version `reverse_merge!` that performs the merge in place:

```
options.reverse_merge!(length: 30, omission: "...")
```

Take into account that `reverse_merge!` may change the hash in the caller, which may or may not be a good idea.

Defined in `active_support/core_ext/hash/reverse_merge.rb`.

**11.2.2 reverse_update** The method reverse_update is an alias for reverse_merge!, explained above. Note that reverse_update has no bang.

Defined in active_support/core_ext/hash/reverse_merge.rb.

**11.2.3 deep_merge and deep_merge!** As you can see in the previous example if a key is found in both hashes the value in the one in the argument wins.

Active Support defines Hash#deep_merge. In a deep merge, if a key is found in both hashes and their values are hashes in turn, then their *merge* becomes the value in the resulting hash:

```
{a: {b: 1}}.deep_merge(a: {c: 2})
# => {:a=>{:b=>1, :c=>2}}
```

The method deep_merge! performs a deep merge in place.

Defined in active_support/core_ext/hash/deep_merge.rb.

## 11.3 Deep duplicating

The method Hash.deep_dup duplicates itself and all keys and values inside recursively with Active Support method Object#deep_dup. It works like Enumerator#each_with_object with sending deep_dup method to each pair inside.

```
hash = { a: 1, b: { c: 2, d: [3, 4] } }

dup = hash.deep_dup
dup[:b][:e] = 5
dup[:b][:d] << 5

hash[:b][:e] == nil      # => true
hash[:b][:d] == [3, 4]   # => true
```

Defined in active_support/core_ext/object/deep_dup.rb.

## 11.4 Working with Keys

**11.4.1 except and except!** The method except returns a hash with the keys in the argument list removed, if present:

```
{a: 1, b: 2}.except(:a) # => {:b=>2}
```

If the receiver responds to convert_key, the method is called on each of the arguments. This allows except to play nice with hashes with indifferent access for instance:

```
{a: 1}.with_indifferent_access.except(:a)  # => {}
{a: 1}.with_indifferent_access.except("a") # => {}
```

There's also the bang variant except! that removes keys in the very receiver.

Defined in active_support/core_ext/hash/except.rb.

**11.4.2 `transform_keys` and `transform_keys!`** The method `transform_keys` accepts a block and returns a hash that has applied the block operations to each of the keys in the receiver:

```
{nil => nil, 1 => 1, a: :a}.transform_keys { |key| key.to_s.upcase }
# => {"" => nil, "A" => :a, "1" => 1}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.transform_keys { |key| key.to_s.upcase }
# The result could either be
# => {"A"=>2}
# or
# => {"A"=>1}
```

This method may be useful for example to build specialized conversions. For instance `stringify_keys` and `symbolize_keys` use `transform_keys` to perform their key conversions:

```
def stringify_keys
  transform_keys { |key| key.to_s }
end
...
def symbolize_keys
  transform_keys { |key| key.to_sym rescue key }
end
```

There's also the bang variant `transform_keys!` that applies the block operations to keys in the very receiver.

Besides that, one can use `deep_transform_keys` and `deep_transform_keys!` to perform the block operation on all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_transform_keys { |key|
key.to_s.upcase }
# => {""=>nil, "1"=>1, "NESTED"=>{"A"=>3, "5"=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

**11.4.3 `stringify_keys` and `stringify_keys!`** The method `stringify_keys` returns a hash that has a stringified version of the keys in the receiver. It does so by sending `to_s` to them:

```
{nil => nil, 1 => 1, a: :a}.stringify_keys
# => {"" => nil, "a" => :a, "1" => 1}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.stringify_keys
# The result could either be
# => {"a"=>2}
# or
# => {"a"=>1}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionView::Helpers::FormHelper` defines:

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")
  options = options.stringify_keys
  options["type"] = "checkbox"
  ...
end
```

The second line can safely access the "type" key, and let the user to pass either `:type` or "type".

There's also the bang variant `stringify_keys!` that stringifies keys in the very receiver.

Besides that, one can use `deep_stringify_keys` and `deep_stringify_keys!` to stringify all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_stringify_keys
# => {""=>nil, "1"=>1, "nested"=>{"a"=>3, "5"=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

**11.4.4 `symbolize_keys` and `symbolize_keys!`** The method `symbolize_keys` returns a hash that has a symbolized version of the keys in the receiver, where possible. It does so by sending `to_sym` to them:

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys
# => {1=>1, nil=>nil, :a=>"a"}
```

Note in the previous example only one key was symbolized.

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.symbolize_keys
# The result could either be
# => {:a=>2}
# or
# => {:a=>1}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionController::UrlRewriter` defines

```
def rewrite_path(options)
  options = options.symbolize_keys
  options.update(options[:params].symbolize_keys) if options[:params]
  ...
end
```

The second line can safely access the `:params` key, and let the user to pass either `:params` or "params".

There's also the bang variant `symbolize_keys!` that symbolizes keys in the very receiver.

Besides that, one can use `deep_symbolize_keys` and `deep_symbolize_keys!` to symbolize all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, "nested" => {"a" => 3, 5 => 5}}.deep_symbolize_keys
# => {nil=>nil, 1=>1, nested:{a:3, 5=>5}}
```

Defined in `active_support/core_ext/hash/keys.rb`.

**11.4.5 `to_options` and `to_options!`** The methods `to_options` and `to_options!` are respectively aliases of `symbolize_keys` and `symbolize_keys!`.

Defined in `active_support/core_ext/hash/keys.rb`.

**11.4.6 `assert_valid_keys`** The method `assert_valid_keys` receives an arbitrary number of arguments, and checks whether the receiver has any key outside that white list. If it does `ArgumentError` is raised.

```
{a: 1}.assert_valid_keys(:a)  # passes
{a: 1}.assert_valid_keys("a") # ArgumentError
```

Active Record does not accept unknown options when building associations, for example. It implements that control via `assert_valid_keys`.

Defined in `active_support/core_ext/hash/keys.rb`.

## 11.5 Working with Values

**11.5.1 `transform_values` && `transform_values!`** The method `transform_values` accepts a block and returns a hash that has applied the block operations to each of the values in the receiver.

```
{ nil => nil, 1 => 1, :x => :a }.transform_values { |value| value.to_s.upcase }
# => {nil=>"", 1=>"1", :x=>"A"}
```

There's also the bang variant `transform_values!` that applies the block operations to values in the very receiver.

Defined in `active_support/core_text/hash/transform_values.rb`.

## 11.6 Slicing

Ruby has built-in support for taking slices out of strings and arrays. Active Support extends slicing to hashes:

```
{a: 1, b: 2, c: 3}.slice(:a, :c)
# => {:c=>3, :a=>1}

{a: 1, b: 2, c: 3}.slice(:b, :X)
# => {:b=>2} # non-existing keys are ignored
```

If the receiver responds to `convert_key` keys are normalized:

```
{a: 1, b: 2}.with_indifferent_access.slice("a")
# => {:a=>1}
```

Slicing may come in handy for sanitizing option hashes with a white list of keys.

There's also `slice!` which in addition to perform a slice in place returns what's removed:

```
hash = {a: 1, b: 2}
rest = hash.slice!(:a) # => {:b=>2}
hash                   # => {:a=>1}
```

Defined in `active_support/core_ext/hash/slice.rb`.

## 11.7 Extracting

The method `extract!` removes and returns the key/value pairs matching the given keys.

```
hash = {a: 1, b: 2}
rest = hash.extract!(:a) # => {:a=>1}
hash                     # => {:b=>2}
```

The method `extract!` returns the same subclass of Hash, that the receiver is.

```
hash = {a: 1, b: 2}.with_indifferent_access
rest = hash.extract!(:a).class
# => ActiveSupport::HashWithIndifferentAccess
```

Defined in `active_support/core_ext/hash/slice.rb`.

## 11.8 Indifferent Access

The method `with_indifferent_access` returns an `ActiveSupport::HashWithIndifferentAccess` out of its receiver:

```
{a: 1}.with_indifferent_access["a"] # => 1
```

Defined in `active_support/core_ext/hash/indifferent_access.rb`.

## 11.9 Compacting

The methods `compact` and `compact!` return a Hash without items with `nil` value.

```
{a: 1, b: 2, c: nil}.compact # => {a: 1, b: 2}
```

Defined in `active_support/core_ext/hash/compact.rb`.

# 12 Extensions to Regexp

## 12.1 multiline?

The method multiline? says whether a regexp has the /m flag set, that is, whether the dot matches newlines.

```
%r{.}.multiline?  # => false
%r{.}m.multiline? # => true

Regexp.new('.').multiline?                    # => false
Regexp.new('.', Regexp::MULTILINE).multiline? # => true
```

Rails uses this method in a single place, also in the routing code. Multiline regexps are disallowed for route requirements and this flag eases enforcing that constraint.

```
def assign_route_options(segments, defaults, requirements)
  ...
  if requirement.multiline?
    raise ArgumentError, "Regexp multiline option not allowed in routing requirements: #
{requirement.inspect}"
  end
  ...
end
```

Defined in active_support/core_ext/regexp.rb.

# 13 Extensions to Range

## 13.1 to_s

Active Support extends the method Range#to_s so that it understands an optional format argument. As of this writing the only supported non-default format is :db:

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"

(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

As the example depicts, the :db format generates a BETWEEN SQL clause. That is used by Active Record in its support for range values in conditions.

Defined in active_support/core_ext/range/conversions.rb.

## 13.2 include?

The methods Range#include? and Range#=== say whether some value falls between the ends of a given instance:

```
(2..3).include?(Math::E) # => true
```

Active Support extends these methods so that the argument may be another range in turn. In that case we test whether the ends of the argument range belong to the receiver themselves:

```
(1..10).include?(3..7)  # => true
(1..10).include?(0..7)  # => false
(1..10).include?(3..11) # => false
(1...9).include?(3..9)  # => false

(1..10) === (3..7)  # => true
(1..10) === (0..7)  # => false
(1..10) === (3..11) # => false
(1...9) === (3..9)  # => false
```

Defined in `active_support/core_ext/range/include_range.rb`.

### 13.3 overlaps?

The method `Range#overlaps?` says whether any two given ranges have non-void intersection:

```
(1..10).overlaps?(7..11)  # => true
(1..10).overlaps?(0..7)   # => true
(1..10).overlaps?(11..27) # => false
```

Defined in `active_support/core_ext/range/overlaps.rb`.

## 14 Extensions to Proc

### 14.1 bind

As you surely know Ruby has an `UnboundMethod` class whose instances are methods that belong to the limbo of methods without a self. The method `Module#instance_method` returns an unbound method for example:

```
Hash.instance_method(:delete) # => #<UnboundMethod: Hash#delete>
```

An unbound method is not callable as is, you need to bind it first to an object with `bind`:

```
clear = Hash.instance_method(:clear)
clear.bind({a: 1}).call # => {}
```

Active Support defines `Proc#bind` with an analogous purpose:

```
Proc.new { size }.bind([]).call # => 0
```

As you see that's callable and bound to the argument, the return value is indeed a `Method`.

To do so `Proc#bind` actually creates a method under the hood. If you ever see a method with a weird name like `__bind_1256598120_237302` in a stack trace you know now where it comes from.

Action Pack uses this trick in `rescue_from` for example, which accepts the name of a method and also a proc as callbacks for a given rescued exception. It has to call them in either case, so a bound method is returned by `handler_for_rescue`, thus simplifying the code in the caller:

```
def handler_for_rescue(exception)
  _, rescuer = Array(rescue_handlers).reverse.detect do |klass_name, handler|
    ...
  end

  case rescuer
  when Symbol
    method(rescuer)
  when Proc
    rescuer.bind(self)
  end
end
```

Defined in `active_support/core_ext/proc.rb`.

# 15 Extensions to `Date`

## 15.1 Calculations

All the following methods are defined in `active_support/core_ext/date/calculations.rb`.

The following calculation methods have edge cases in October 1582, since days 5..14 just do not exist. This guide does not document their behavior around those days for brevity, but it is enough to say that they do what you would expect. That is, `Date.new(1582, 10, 4).tomorrow` returns `Date.new(1582, 10, 15)` and so on. Please check `test/core_ext/date_ext_test.rb` in the Active Support test suite for expected behavior.

**15.1.1** `Date.current` Active Support defines `Date.current` to be today in the current time zone. That's like `Date.today`, except that it honors the user time zone, if defined. It also defines `Date.yesterday` and `Date.tomorrow`, and the instance predicates `past?`, `today?`, and `future?`, all of them relative to `Date.current`.

When making Date comparisons using methods which honor the user time zone, make sure to use `Date.current` and not `Date.today`. There are cases where the user time zone might be in the future compared to the system time zone, which `Date.today` uses by default. This means `Date.today` may equal `Date.yesterday`.

**15.1.2 Named dates** 15.1.2.1 `prev_year`, `next_year`

In Ruby 1.9 `prev_year` and `next_year` return a date with the same day/month in the last or next year:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_year              # => Fri, 08 May 2009
d.next_year              # => Sun, 08 May 2011
```

If date is the 29th of February of a leap year, you obtain the 28th:

```
d = Date.new(2000, 2, 29) # => Tue, 29 Feb 2000
d.prev_year               # => Sun, 28 Feb 1999
d.next_year               # => Wed, 28 Feb 2001
```

prev_year is aliased to last_year.

15.1.2.2 prev_month, next_month

In Ruby 1.9 prev_month and next_month return the date with the same day in the last or next month:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_month             # => Thu, 08 Apr 2010
d.next_month             # => Tue, 08 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2000, 5, 31).prev_month # => Sun, 30 Apr 2000
Date.new(2000, 3, 31).prev_month # => Tue, 29 Feb 2000
Date.new(2000, 5, 31).next_month # => Fri, 30 Jun 2000
Date.new(2000, 1, 31).next_month # => Tue, 29 Feb 2000
```

prev_month is aliased to last_month.

15.1.2.3 prev_quarter, next_quarter

Same as prev_month and next_month. It returns the date with the same day in the previous or next quarter:

```
t = Time.local(2010, 5, 8) # => Sat, 08 May 2010
t.prev_quarter             # => Mon, 08 Feb 2010
t.next_quarter             # => Sun, 08 Aug 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Time.local(2000, 7, 31).prev_quarter  # => Sun, 30 Apr 2000
Time.local(2000, 5, 31).prev_quarter  # => Tue, 29 Feb 2000
Time.local(2000, 10, 31).prev_quarter # => Mon, 30 Oct 2000
Time.local(2000, 11, 31).next_quarter # => Wed, 28 Feb 2001
```

prev_quarter is aliased to last_quarter.

15.1.2.4 beginning_of_week, end_of_week

The methods beginning_of_week and end_of_week return the dates for the beginning and end of the week, respectively. Weeks are assumed to start on Monday, but that can be changed passing an argument, setting thread local Date.beginning_of_week or config.beginning_of_week.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.beginning_of_week          # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week                # => Sun, 09 May 2010
d.end_of_week(:sunday)       # => Sat, 08 May 2010
```

beginning_of_week is aliased to at_beginning_of_week and end_of_week is aliased to at_end_of_week.

15.1.2.5 monday, sunday

The methods monday and sunday return the dates for the previous Monday and next Sunday, respectively.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.monday                     # => Mon, 03 May 2010
d.sunday                     # => Sun, 09 May 2010

d = Date.new(2012, 9, 10)    # => Mon, 10 Sep 2012
d.monday                     # => Mon, 10 Sep 2012

d = Date.new(2012, 9, 16)    # => Sun, 16 Sep 2012
d.sunday                     # => Sun, 16 Sep 2012
```

15.1.2.6 prev_week, next_week

The method next_week receives a symbol with a day name in English (default is the thread local Date .beginning_of_week, or config.beginning_of_week, or :monday) and it returns the date corresponding to that day.

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.next_week              # => Mon, 10 May 2010
d.next_week(:saturday)   # => Sat, 15 May 2010
```

The method prev_week is analogous:

```
d.prev_week              # => Mon, 26 Apr 2010
d.prev_week(:saturday)   # => Sat, 01 May 2010
d.prev_week(:friday)     # => Fri, 30 Apr 2010
```

prev_week is aliased to last_week.

Both next_week and prev_week work as expected when Date.beginning_of_week or config.beginning _of_week are set.

15.1.2.7 beginning_of_month, end_of_month

The methods beginning_of_month and end_of_month return the dates for the beginning and end of the month:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month     # => Sat, 01 May 2010
d.end_of_month           # => Mon, 31 May 2010
```

beginning_of_month is aliased to at_beginning_of_month, and end_of_month is aliased to at_end_of _month.

15.1.2.8 beginning_of_quarter, end_of_quarter

The methods beginning_of_quarter and end_of_quarter return the dates for the beginning and end of the quarter of the receiver's calendar year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter   # => Thu, 01 Apr 2010
d.end_of_quarter         # => Wed, 30 Jun 2010
```

beginning_of_quarter is aliased to at_beginning_of_quarter, and end_of_quarter is aliased to at_end_of_quarter.

15.1.2.9 beginning_of_year, end_of_year

The methods beginning_of_year and end_of_year return the dates for the beginning and end of the year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year      # => Fri, 01 Jan 2010
d.end_of_year            # => Fri, 31 Dec 2010
```

beginning_of_year is aliased to at_beginning_of_year, and end_of_year is aliased to at_end_of_year.

**15.1.3 Other Date Computations** 15.1.3.1 years_ago, years_since

The method years_ago receives a number of years and returns the same date those many years ago:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

years_since moves forward in time:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2012, 2, 29).years_ago(3)     # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3)   # => Sat, 28 Feb 2015
```

15.1.3.2 months_ago, months_since

The methods months_ago and months_since work analogously for months:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2) # => Wed, 30 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2010, 4, 30).months_ago(2)    # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

15.1.3.3 weeks_ago

The method weeks_ago works analogously for weeks:

```
Date.new(2010, 5, 24).weeks_ago(1)    # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)    # => Mon, 10 May 2010
```

15.1.3.4 advance

The most generic way to jump to other days is advance. This method receives a hash with keys :years, :months, :weeks, :days, and returns a date advanced as much as the present keys indicate:

```
date = Date.new(2010, 6, 6)
date.advance(years: 1, weeks: 2)  # => Mon, 20 Jun 2011
date.advance(months: 2, days: -2) # => Wed, 04 Aug 2010
```

Note in the previous example that increments may be negative.

To perform the computation the method first increments years, then months, then weeks, and finally days. This order is important towards the end of months. Say for example we are at the end of February of 2010, and we want to move one month and one day forward.

The method `advance` advances first one month, and then one day, the result is:

```
Date.new(2010, 2, 28).advance(months: 1, days: 1)
# => Sun, 29 Mar 2010
```

While if it did it the other way around the result would be different:

```
Date.new(2010, 2, 28).advance(days: 1).advance(months: 1)
# => Thu, 01 Apr 2010
```

**15.1.4 Changing Components** The method `change` allows you to get a new date which is the same as the receiver except for the given year, month, or day:

```
Date.new(2010, 12, 23).change(year: 2011, month: 11)
# => Wed, 23 Nov 2011
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
Date.new(2010, 1, 31).change(month: 2)
# => ArgumentError: invalid date
```

**15.1.5 Durations** Durations can be added to and subtracted from dates:

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

**15.1.6 Timestamps** The following methods return a `Time` object if possible, otherwise a `DateTime`. If set, they honor the user time zone.

15.1.6.1 `beginning_of_day`, `end_of_day`

The method `beginning_of_day` returns a timestamp at the beginning of the day (00:00:00):

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Mon Jun 07 00:00:00 +0200 2010
```

The method `end_of_day` returns a timestamp at the end of the day (23:59:59):

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Mon Jun 07 23:59:59 +0200 2010
```

`beginning_of_day` is aliased to `at_beginning_of_day`, `midnight`, `at_midnight`.

15.1.6.2 `beginning_of_hour`, `end_of_hour`

The method `beginning_of_hour` returns a timestamp at the beginning of the hour (hh:00:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_hour # => Mon Jun 07 19:00:00 +0200 2010
```

The method `end_of_hour` returns a timestamp at the end of the hour (hh:59:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_hour # => Mon Jun 07 19:59:59 +0200 2010
```

`beginning_of_hour` is aliased to `at_beginning_of_hour`.

15.1.6.3 `beginning_of_minute`, `end_of_minute`

The method `beginning_of_minute` returns a timestamp at the beginning of the minute (hh:mm:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_minute # => Mon Jun 07 19:55:00 +0200 2010
```

The method `end_of_minute` returns a timestamp at the end of the minute (hh:mm:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_minute # => Mon Jun 07 19:55:59 +0200 2010
```

`beginning_of_minute` is aliased to `at_beginning_of_minute`.

`beginning_of_hour`, `end_of_hour`, `beginning_of_minute` and `end_of_minute` are implemented for `Time` and `DateTime` but **not** `Date` as it does not make sense to request the beginning or end of an hour or minute on a `Date` instance.

15.1.6.4 `ago`, `since`

The method `ago` receives a number of seconds as argument and returns a timestamp those many seconds ago from midnight:

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

Similarly, `since` moves forward:

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

#### 15.1.7 Other Time Computations

### 15.2 Conversions

## 16 Extensions to DateTime

DateTime is not aware of DST rules and so some of these methods have edge cases when a DST change is going on. For example seconds_since_midnight might not return the real amount in such a day.

### 16.1 Calculations

All the following methods are defined in active_support/core_ext/date_time/calculations.rb.

The class DateTime is a subclass of Date so by loading active_support/core_ext/date/calculations.rb you inherit these methods and their aliases, except that they will always return datetimes:

```
yesterday
tomorrow
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
```

The following methods are reimplemented so you do **not** need to load active_support/core_ext/date/calculations.rb for these ones:

```
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
ago
since (in)
```

On the other hand, advance and change are also defined and support more options, they are documented below.

The following methods are only implemented in active_support/core_ext/date_time/calculations.rb as they only make sense when used with a DateTime instance:

```
beginning_of_hour (at_beginning_of_hour)
end_of_hour
```

### 16.1.1 Named Datetimes

16.1.1.1 DateTime.current

Active Support defines DateTime.current to be like Time.now.to_datetime, except that it honors the user time zone, if defined. It also defines DateTime.yesterday and DateTime.tomorrow, and the instance predicates past?, and future? relative to DateTime.current.

### 16.1.2 Other Extensions

16.1.2.1 seconds_since_midnight

The method seconds_since_midnight returns the number of seconds since midnight:

```
now = DateTime.current     # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight # => 73596
```

16.1.2.2 utc

The method utc gives you the same datetime in the receiver expressed in UTC.

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                # => Mon, 07 Jun 2010 23:27:52 +0000
```

This method is also aliased as getutc.

16.1.2.3 utc?

The predicate utc? says whether the receiver has UTC as its time zone:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

16.1.2.4 advance

The most generic way to jump to another datetime is advance. This method receives a hash with keys :years, :months, :weeks, :days, :hours, :minutes, and :seconds, and returns a datetime advanced as much as the present keys indicate.

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(years: 1, months: 1, days: 1, hours: 1, minutes: 1, seconds: 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

This method first computes the destination date passing :years, :months, :weeks, and :days to Date#advance documented above. After that, it adjusts the time calling since with the number of seconds to advance. This order is relevant, a different ordering would give different datetimes in some edge-cases. The example in Date#advance applies, and we can extend it to show order relevance related to the time bits.

If we first move the date bits (that have also a relative order of processing, as documented before), and then the time bits we get for example the following computation:

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(months: 1, seconds: 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

but if we computed them the other way around, the result would be different:

```
d.advance(seconds: 1).advance(months: 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

Since `DateTime` is not DST-aware you can end up in a non-existing point in time with no warning or error telling you so.

**16.1.3 Changing Components** The method `change` allows you to get a new datetime which is the same as the receiver except for the given options, which may include `:year`, `:month`, `:day`, `:hour`, `:min`, `:sec`, `:offset`, `:start`:

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(year: 2011, offset: Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

If hours are zeroed, then minutes and seconds are too (unless they have given values):

```
now.change(hour: 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

Similarly, if minutes are zeroed, then seconds are too (unless it has given a value):

```
now.change(min: 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
DateTime.current.change(month: 2, day: 30)
# => ArgumentError: invalid date
```

**16.1.4 Durations** Durations can be added to and subtracted from datetimes:

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

# 17 Extensions to Time

## 17.1 Calculations

All the following methods are defined in active_support/core_ext/time/calculations.rb.

Active Support adds to Time many of the methods available for DateTime:

```
past?
today?
future?
yesterday
tomorrow
seconds_since_midnight
change
advance
ago
since (in)
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
beginning_of_hour (at_beginning_of_hour)
end_of_hour
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
```

They are analogous. Please refer to their documentation above and take into account the following differences:

- change accepts an additional :usec option.

- Time understands DST, so you get correct DST calculations as in

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid",
...>

# In Barcelona, 2010/03/28 02:00 +0100 becomes 2010/03/28 03:00 +0200 due to DST.
t = Time.local(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(seconds: 1)
# => Sun Mar 28 03:00:00 +0200 2010
```

- If since or ago jump to a time that can't be expressed with Time a DateTime object is returned instead.

**17.1.1** `Time.current` Active Support defines `Time.current` to be today in the current time zone. That's like `Time.now`, except that it honors the user time zone, if defined. It also defines the instance predicates `past?`, `today?`, and `future?`, all of them relative to `Time.current`.

When making Time comparisons using methods which honor the user time zone, make sure to use `Time.current` instead of `Time.now`. There are cases where the user time zone might be in the future compared to the system time zone, which `Time.now` uses by default. This means `Time.now.to_date` may equal `Date.yesterday`.

**17.1.2** `all_day`, `all_week`, `all_month`, `all_quarter` **and** `all_year` The method `all_day` returns a range representing the whole day of the current time.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_day
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00
```

Analogously, `all_week`, `all_month`, `all_quarter` and `all_year` all serve the purpose of generating time ranges.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_week
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00
now.all_week(:sunday)
# => Sun, 16 Sep 2012 00:00:00 UTC +00:00..Sat, 22 Sep 2012 23:59:59 UTC +00:00
now.all_month
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00
now.all_quarter
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00
now.all_year
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00
```

### 17.2 Time Constructors

Active Support defines `Time.current` to be `Time.zone.now` if there's a user time zone defined, with fallback to `Time.now`:

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid",
...>
Time.current
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

Analogously to `DateTime`, the predicates `past?`, and `future?` are relative to `Time.current`.

If the time to be constructed lies beyond the range supported by `Time` in the runtime platform, usecs are discarded and a `DateTime` object is returned instead.

**17.2.1 Durations** Durations can be added to and subtracted from time objects:

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now + 1.year
#  => Tue, 09 Aug 2011 23:21:11 UTC +00:00
now - 1.week
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Time.utc(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

## 18 Extensions to `File`

### 18.1 `atomic_write`

With the class method `File.atomic_write` you can write to a file in a way that will prevent any reader from seeing half-written content.

The name of the file is passed as an argument, and the method yields a file handle opened for writing. Once the block is done `atomic_write` closes the file handle and completes its job.

For example, Action Pack uses this method to write asset cache files like `all.css`:

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

To accomplish this `atomic_write` creates a temporary file. That's the file the code in the block actually writes to. On completion, the temporary file is renamed, which is an atomic operation on POSIX systems. If the target file exists `atomic_write` overwrites it and keeps owners and permissions. However there are a few

cases where `atomic_write` cannot change the file ownership or permissions, this error is caught and skipped over trusting in the user/filesystem to ensure the file is accessible to the processes that need it.

Due to the chmod operation `atomic_write` performs, if the target file has an ACL set on it this ACL will be recalculated/modified.

Note you can't append with `atomic_write`.

The auxiliary file is written in a standard directory for temporary files, but you can pass a directory of your choice as second argument.

Defined in `active_support/core_ext/file/atomic.rb`.

# 19 Extensions to `Marshal`

## 19.1 `load`

Active Support adds constant autoloading support to `load`.

For example, the file cache store deserializes this way:

```
File.open(file_name) { |f| Marshal.load(f) }
```

If the cached data refers to a constant that is unknown at that point, the autoloading mechanism is triggered and if it succeeds the deserialization is retried transparently.

If the argument is an `IO` it needs to respond to `rewind` to be able to retry. Regular files respond to `rewind`.

Defined in `active_support/core_ext/marshal.rb`.

# 20 Extensions to `Logger`

## 20.1 `around_[level]`

Takes two arguments, a `before_message` and `after_message` and calls the current level method on the `Logger` instance, passing in the `before_message`, then the specified message, then the `after_message`:

```
logger = Logger.new("log/development.log")
logger.around_info("before", "after") { |logger| logger.info("during") }
```

## 20.2 `silence`

Silences every log level lesser to the specified one for the duration of the given block. Log level orders are: debug, info, error and fatal.

```
logger = Logger.new("log/development.log")
logger.silence(Logger::INFO) do
  logger.debug("In space, no one can hear you scream.")
  logger.info("Scream all you want, small mailman!")
end
```

### 20.3 datetime_format=

Modifies the datetime format output by the formatter class associated with this logger. If the formatter class does not have a `datetime_format` method then this is ignored.

```
class Logger::FormatWithTime < Logger::Formatter
  cattr_accessor(:datetime_format) { "%Y%m%d%H%m%S" }

  def self.call(severity, timestamp, progname, msg)
    "#{timestamp.strftime(datetime_format)} -- #{String === msg ? msg : msg.inspect}\n"
  end
end

logger = Logger.new("log/development.log")
logger.formatter = Logger::FormatWithTime
logger.info("<- is the current time")
```

Defined in `active_support/core_ext/logger.rb`.

## 21 Extensions to NameError

Active Support adds `missing_name?` to `NameError`, which tests whether the exception was raised because of the name passed as argument.

The name may be given as a symbol or string. A symbol is tested against the bare constant name, a string is against the fully-qualified constant name.

A symbol can represent a fully-qualified constant name as in `:"ActiveRecord::Base"`, so the behavior for symbols is defined for convenience, not because it has to be that way technically.

For example, when an action of `ArticlesController` is called Rails tries optimistically to use `ArticlesHelper`. It is OK that the helper module does not exist, so if an exception for that constant name is raised it should be silenced. But it could be the case that `articles_helper.rb` raises a `NameError` due to an actual unknown constant. That should be reraised. The method `missing_name?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

Defined in `active_support/core_ext/name_error.rb`.

## 22 Extensions to LoadError

Active Support adds `is_missing?` to `LoadError`, and also assigns that class to the constant `MissingSourceFile` for backwards compatibility.

Given a path name `is_missing?` tests whether the exception was raised due to that particular file (except perhaps for the ".rb" extension).

For example, when an action of `ArticlesController` is called Rails tries to load `articles_helper.rb`, but that file may not exist. That's fine, the helper module is not mandatory so Rails silences a load error. But it could be the case that the helper module does exist and in turn requires another library that is missing. In that case Rails must reraise the exception. The method `is_missing?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

Defined in `active_support/core_ext/load_error.rb`.

## 23 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.