

Share this:



Related

[Method Arguments In Ruby](#)

August 17, 2009

In "coding"

[Using Ruby Blocks And Rolling Your Own Iterators](#)

September 3, 2009

In "coding"

[A Wealth Of Ruby Loops And Iterators](#)

September 2, 2009

In "coding"

[ABOUT](#) [CONTACT](#)[FOLLOW ME ON TWITTER](#)

SKORKS

How A Ruby Case Statement Works And What You Can Do With It

August 24, 2009 By [Alan Skorkin](#) [25 Comments](#)

I found *case* statements in Ruby pretty interesting as they are capable of doing a little more than the equivalent constructs in other languages. We all know how a simple *case* statement works, we test on a condition that we give to a *case*

statement, we then walk through a set of possible matches each of which is contained in a *when* statement e.g.:

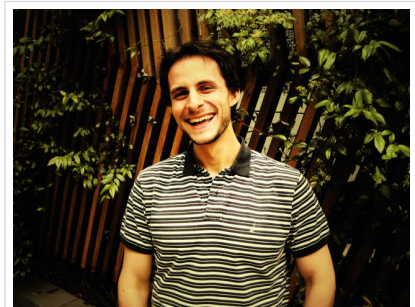
```
print "Enter your grade: "
grade = gets.chomp
case grade
when "A"
  puts 'Well done!'
when "B"
  puts 'Try harder!'
when "C"
  puts 'You need help!!!'
else
  puts "You just making it up!"
end
```

So far nothing special, the above works just the way you would expect. But, you can do more with a case statement in Ruby.

Multi-Value When And No-Value Case

Ruby allows you to supply multiple values to a when statement rather than just one e.g.:

```
print "Enter your grade: "
grade = gets.chomp
```



Receive Rss Updates

[RSS - Posts](#)

Receive Email Updates

SUBSCRIBE

Popular Posts

[How A Ruby Case Statement Works And What You Can Do With It](#)[The Difference Between A Developer, A Programmer And A Computer Scientist](#)[How To Quickly Generate A Large File On The Command Line \(With Linux\)](#)[Here Are Some Words That Rhyme With Orange!](#)[Method Arguments In Ruby](#)[Why Developers Never Use State Machines](#)[Bash Shortcuts For Maximum Productivity](#)[The Best Way To Pretty Print JSON On The Command-Line](#)[What Every Developer Should Know About URLs](#)

```
case grade
when "A", "B"
  puts 'You pretty smart!'
when "C", "D"
  puts 'You pretty dumb!!'
else
  puts "You can't even use a computer!"
end
```

Pretty cool, but nothing too revolutionary. It doesn't end there though you can use a *case* statement without giving it a value to match against, which allows a case statement to mimic the behavior of an if statement, e.g.:

```
print "Enter a string: "
some_string = gets.chomp
case
when some_string.match(/\d/)
  puts 'String has numbers'
when some_string.match(/[a-zA-Z]/)
  puts 'String has letters'
else
  puts 'String has no numbers or letters'
end
```

So, why would you use it instead of an if statement? Well, there is probably no reason, they are equivalent. But, it is good to be aware that you can do this, especially if you run into it in the wild, wouldn't want to be caught unawares.

It All Evaluates To An Object

You probably keep hearing this over and over, but everything in Ruby works with objects. Things are no different when it comes to *case* statements. A case statement will always return a single object, just like a method call. So, you can safely wrap a case statement in a method call and Ruby will have no problems with it. For example, the above version of the case statement can be re-written like this:

```
print "Enter a string: "
some_string = gets.chomp
puts case
when some_string.match(/\d/)
  'String has numbers'
when some_string.match(/[a-zA-Z]/)
  'String has letters'
else
  'String has no numbers or letters'
end
```

We are now wrapping the whole case statement in a *puts* method call, rather than doing it within each individual *when* statement. This works because no matter which when/else succeeds, the whole case statement returns the result of the last line that was executed (in our case it just always returns a string), and so the *puts* just writes out whatever string is returned.

How It All Works Under The Hood

You can almost consider *case/when* statements to be syntactic sugar for a method call. Every object in Ruby inherits a method called the case equality method, also known as the triple equals (*===*). You can think of it as an operator if it helps (the *===* operator), but we know that Ruby operators are just syntactic sugar for method calls. So whenever a *when* is trying to match a value (except when we are using a

no-value *case*) there is a method call behind the scenes to the `===` method/operator. We can therefore take our very first example and re-write it using if statements to be completely equivalent to a case statement:

```
print "Enter your grade: "
grade = gets.chomp
if "A" === grade
  puts 'Well done!'
elsif "B" === grade
  puts 'Try harder!'
elsif "C" === grade
  puts 'You need help!!!'
else
  puts "You just making it up!"
end
```

The implications of this are as follows. Any class you write can override the `===` method and therefore define it's own behavior for usage in a case statement. For built-in objects such as strings, the `===` operator/method is simply equivalent to the `==` method/operator. But you don't have to be restricted to this. If you want your class to have the ability to participate in a case statement in some specialized way all you need to do is something like this:

```
class Vehicle
  attr_accessor :number_of_wheels

  def initialize(number_of_wheels)
    @number_of_wheels = number_of_wheels
  end

  def ==(another_vehicle)
    self.number_of_wheels == another_vehicle.number_of_wheels
  end
end

four_wheeler = Vehicle.new 4
two_wheeler = Vehicle.new 2

print "Enter number of wheel for vehicle: "
vehicle = Vehicle.new gets.chomp.to_i

puts case vehicle
when two_wheeler
  'Vehicle has the same number of wheels as a two-wheeler!'
when four_wheeler
  'Vehicle has the same number of wheels as a four-wheeler!'
else
  "Don't know of a vehicle with that wheel arrangement!"
end
```

In this case even though we are matching directly on the vehicle object in our *case/when* statement, behind the scenes a method call is made to the `===` operator and so the real match is made on the number of wheels attribute that the vehicle object has (as per what is defined in the `==` method).

This idea of hiding method calls behind syntax that doesn't look like method calls, seems to be rather common in Ruby. Personally I like it, you can do some rather elegant things, but the building blocks are still just a bunch of method calls. When

it comes to the *case* statement though, I hardly ever use it when I program in Java for example, but it looks to me like it can be somewhat more handy in Ruby especially with the ability to define custom *case* behavior for your own objects.

Image by [dahliascakes](#)

Share this:



25 Comments

Skorks

Login ▾

Recommend 2

Share

Sort by Best ▾



Join the discussion...



naudster • 4 years ago

I don't use "case" very often, and always forget the syntax when I do want to use it, so google keeps bringing me back here. Thanks for the thorough explanation!

25 ^ | ▾ • Reply • Share ›



Zhongpeng Lin • 5 years ago

It seems like case statement couldn't handle multiple assignment. When I tried to do this:

```
next_type, index = case type
when 25
  10, 0
when 10
  5, 1
when 5
  1, 2
when 1
  nil, 3
end
```

I got syntax errors. Am I doing it wrong?

1 ^ | ▾ • Reply • Share ›



Sjors Branderhorst → Zhongpeng Lin • 5 years ago

you are not doing it wrong: return as [10,0] instead of 10,0 -- parallel assignment doesn't obey principle of least surprise inside case statements

6 ^ | ▾ • Reply • Share ›



lucas • 8 years ago

I was writing a state machine in ruby the other day using a case statement to fork off the work to various methods based on which :symbol was assigned to @state

```
case @state
when :state_1
  state_1(some_arg)
when :state_2
  state_2(some_arg)
when :state_3
  state_3(some_arg)
when :state_4
  state_4(some_arg)
else
  raise "invalid state"
end
```

Then I had it pointed out to me that the whole thing was unnecessary and could better be written as:

```
self.send(@state, some_arg)
```

```
some_str @state, some_arg
```

Ruby is awesome like that :)

1 ^ | v • Reply • Share ›



Alan Skorkin → lucas • 8 years ago

How cool is it :, as I dig deeper into Ruby I keep discovering things that just blow me away sometimes, not that Ruby doesn't have it's share of issues, but still :).

^ | v • Reply • Share ›



namelessjon • 8 years ago

Your example with the regexes can actually be written as:

```
case some_string
when /\d/
'String has numbers'
when /[a-zA-Z]/
'String has letters'
else
'String has no numbers or letters'
end
```

1 ^ | v • Reply • Share ›



Alan Skorkin → namelessjon • 8 years ago

Yes, very true.

I wrote it the way I did to illustrate the fact that you can use a case to mimic an if by not providing the case with a value.

^ | v • Reply • Share ›



Adesola • 5 years ago

pls, i try running this code but it's nt working, could anyone fix it for me, I'm still learning ruby

```
puts "\nWelcome to the vacation calculator!\n\n"
print "How many years have you worked for the company? \n\n: "
answer = STDIN.gets
answer.chop!
answer = answer.to_i
puts case
when (answer.between?(1, 5))
puts "You are entitled to 1 week of vacation per year."
when (answer.between?(6, 10))
puts "You are entitled to 2 weeks of vacation per year."
when (answer.between?(11, 30))
puts "You are entitled to 3 weeks of vacation per year."
else
You are entitled to 5 weeks of vacation per year."
```

end

^ | v • Reply • Share ›



Stephen A. Wilson → Adesola • 4 years ago

you have puts in each of your cases, when you have the entire thing in a puts statement. additionally, you did not say which value to use for the cases.

```
puts case answer
when between?(1,5)
"You are entitled to 1 week of vacation per year."
when between?(6, 10)
"You are entitled to 2 weeks of vacation per year."
when between?(11, 30)
"You are entitled to 3 weeks of vacation per year."
else
"You are entitled to 5 weeks of vacation per year."
end
```

sidenote:

How many years have you worked for the company? 0

You are entitled to 5 weeks of vacation per year.

3 ^ | v • Reply • Share ›



Joshua Toyota ➔ Adesola • 3 years ago

or you can do this:

```
puts case true
when answer.between?(1,5) # true === answer.between?(1,5)
  '1wk/yr'
# ...
end
^ | v • Reply • Share ›
```



Satya • 5 years ago

Thanks, Alan. I'm using this form:

```
some_config = case COLLEGE
when 'univ of bar', 'Univ of baz'
  true
else
  false
end
```

Well, that's trivial but I use it for other less trivial config stuff. I maintain software that's customized for various clients.

^ | v • Reply • Share ›



Suresh Nambiar • 6 years ago

Slightly modified version...

Provides a function to convert a dateString to date.

```
def server_format_date ( strdate )
  strdt = strdate.chomp.strip
  case strdt.gsub(/\D/, "-")
  when /\d{8}/ then Date.strptime(strdt, "%d%m%Y")
  when /\d{7}/ then Date.strptime('o' + strdt, "%d%m%Y")
  when /\d{6}/ then Date.strptime(strdt, "%d%m%y")
  when /\d{5}/ then Date.strptime('o' + strdt, "%d%m%y")
  when /\d{4}/ then Date.strptime(strdt[0] + "-" + strdt[1] + "-" + strdt[2..3],
    "%d-%m-%y")
  when /\d{2}-\d{2}-\d{4}/ then Date.strptime(strdt, "%d-%m-%Y")
  when /\d{2}-\d{2}-\d{2}/ then Date.strptime(strdt, "%d-%m-%y")
  when /\d{4}-\d{2}-\d{2}/ then Date.strptime(strdt, "%Y-%m-%d")
  end
end
```

[see more](#)

^ | v • Reply • Share ›



Suresh Nambiar • 6 years ago

```
def server_format_date ( strdate )
  strdt = strdate.chomp
  case strdt.gsub(/\D/, "-")
  when /\d{8}/ then Date.strptime(strdt, "%d%m%Y")
  when /\d{7}/ then Date.strptime('o' + strdt, "%d%m%Y")
  when /\d{6}/ then Date.strptime(strdt, "%d%m%y")
  when /\d{5}/ then Date.strptime('o' + strdt, "%d%m%y")
  when /\d{4}/ then Date.strptime(strdt[0] + "-" + strdt[1] + "-" + strdt[2..3],
    "%d-%m-%y")
  when /\d{2}-\d{2}-\d{4}/ then Date.strptime(strdt, "%d-%m-%Y")
  when /\d{2}-\d{2}-\d{2}/ then Date.strptime(strdt, "%d-%m-%y")
  when /\d{4}-\d{2}-\d{2}/ then Date.strptime(strdt, "%Y-%m-%d")
  end
end
```

Test Cases :-

```
server_format_date ( '2011-3-4' )
```

[see more](#)

^ | v • Reply • Share ›



Chris • 6 years ago

Thanks, dude! From problem to solution within 30 seconds of the google. Thanks!

^ | v • Reply • Share ›



Mitch • 6 years ago

Of course, this means that the case statement is linear in time rather than constant as it would be in other languages. That's the original point of case statements, after all.

^ | v • Reply • Share ›



Alwyn → Mitch • 5 years ago

I agree. In other languages my main reason for using case/switch would be performance due to it being constant time. If this is not the case in Ruby, are there any other good reasons beyond simplifying logic? For performance I guess one can set up a Hash with each value being a lambda.

^ | v • Reply • Share ›



Aleksandar Kostadinov → Alwyn • 2 years ago

nicer code look

^ | v • Reply • Share ›



Akhil Bansal • 7 years ago

Interesting, I didn't know it.

^ | v • Reply • Share ›



FrihD • 8 years ago

Also, "when some_class" tests the class of the object passed in the case statement. e.g.

```
klass = Class.new
case klass.new
when klass
  p "hurrah"
end
```

^ | v • Reply • Share ›



Alan Skorkin → FrihD • 8 years ago

Hmm, I am not precisely sure what you mean, could you elaborate?

1 ^ | v • Reply • Share ›



John F. Miller → Alan Skorkin • 8 years ago

Another Way of saying this is that by default `Class#===` checks whether `target#kind_of?` evaluates to true for the given class. This is very useful when dealing with a method that can take many different types of arguments:

```
def foo(input)
  case input
  when String
    JSON.parse input
  when Hash
    input
  when Numeric
    {:value=>input}
  when NilClass
    {}
  end
end
```

#

...
end

^ | v • Reply • Share ›



Alan Skorkin → John F. Miller • 8 years ago

That clarifies it even more. Definitely some great use cases for case :).

^ | v • Reply • Share ›



FrihD → Alan Skorkin • 8 years ago

Instead of having a lot of "is_a? Foo" you can do:

```
case obj
when Foo
do_something
when Bar
do_anotherthing
end
```

It can be useful when you implement e.g. a Request class with many subclasses.

(but at some point, a hash becomes easier to manage). It is also useful when you want to get rid of duck_typing in a method or simulate method overloading.

^ | v • Reply • Share ›



Alan Skorkin → FrihD • 8 years ago

Ahh yes I get it now, my apologies for being a little slow :).

That is definitely very handy.

^ | v • Reply • Share ›



smeally → FrihD • 7 years ago


Hey there this is a great post

^ | v • Reply • Share ›

ALSO ON SKORKS


The Best Way To Pretty Print JSON On The Command-Line

18 comments • 4 years ago •

 **Bill Dueber** — Two other options: Perl's JSON::XS ships with a json_xs binary that will do pretty-printing of JSON ...


Ruby & (Ampersand) Parameter Demystified

10 comments • 4 years ago •

 **Alex Popov** — I try to avoid leaving useless "Thank you" comments, but I really need a way to express my ...


A Closure Is Not Always A Closure In Ruby




14 comments • 4 years ago •

 **mrdias** — I have faced a similar problem in the past, this can mean that your dsl is missing some abstraction. ...

Ruby – Why U No Have Nested Exceptions?

13 comments • 4 years ago •

 **headius** — I have filed an issue with MRI in the "Common Ruby" project to hopefully get Exception#cause logic ...

 [Subscribe](#)  [Add Disqus to your site](#) [Add Disqus](#) [Add](#)  [Privacy](#)