-

The Challenge

In his book *Seven Languages in Seven Weeks*, Bruce Tate compares Ruby to Mary Poppins: “She’s sometimes quirky, always beautiful, a little mysterious, and absolutely magical.” If we really had to pick a character to compare with this language, we could hardly make a better choice. The comparison, of course, stems from the philosophy of Ruby’s design: the language should make programmers happy.

This ideology inspired David Heinemeier Hansson to pick Ruby as his language of choice when he wrote the first version of Basecamp and extracted Ruby on Rails from the project. Since the language became popular due to the success of Rails, it’s common to hear people misusing “Ruby” and “Rails” synonymously, or to miscategorize Ruby as a “web language”.

As experienced Rubyists know, the language is not in any way limited to web development, and can be used for almost anything: writing native smartphone apps, data-processing scripts, or as a tool to teach people to program.

Even though the language markets itself as being developer-friendly, it can provide a few surprises for programmers coming from the other languages, because it changes the semantics of many familiar concepts in unexpected ways.

Mastering Ruby can be hard for a few reasons, and the popularity of Rails is one of them. Namely, many developers learned just enough Ruby to start using Rails and stopped exploring the language after that. Due to Ruby's magical typing scheme, it can be hard for these developers to understand what is pure Ruby and what is an extension built into Rails. Given the many extensions Rails provides, the motivation to study the basics of the language is diminished.

Additional complexity lies in the multiple faces of Ruby. While it is primarily an object-oriented language, it also allows writing programs in the functional programming style, which can be leveraged for improved expressiveness and code readability. This means that Ruby provides many ways to do a single thing, and thus the "Class A" Ruby programmer needs to understand all of the different personalities of the language.

It doesn't help that today we've got multiple implementations of Ruby, including MRI, JRuby, Rubinius, and mruby. Thus, to make good decisions and judgments, Ruby developers need to be, at the very least, aware of the different implementations, and their strengths and weaknesses.

The following guide gives you some questions for inspiration when preparing an interview for top-notch Ruby developers. You should not assess candidates based purely on their ability to "correctly" answer each question. That would be missing the point, because not every top candidate will actually know all the details about the language, nor does knowing all the details guarantee you the best developer. Instead, use the questions to see how the candidate thinks about the problem at hand, how they work through difficult problems, and how they weigh alternatives and make judgment calls. For example, if the candidate doesn't know the answer, ask how they would get it, provided they had a computer with internet access and a Ruby interpreter installed. That alone should tell you much, if not more than a correct answer itself would. You should also check out Toptal's general guide on hiring candidates, published on our blog: *In Search of the Elite Few*.

Note that this guide focuses on pure Ruby. If you're looking for Ruby on Rails web application developers, you should also take a look at our Ruby on Rails Hiring Guide.

Welcome to Ruby

Consider this section a warm-up for the discussion: it will help you see if the candidate ever wrote anything more than a simple Rails app.

Q: What are the differences between classes and modules? What are the similarities?

Modules in Ruby have two purposes: namespacing and mixins.

Consider an application that processes maps with points of interest. Two programmers working on this app could come up with two different classes and give them the same name. Let's say the first one creates a class `Point`, which represents a point in a plane defined by *x* and *y* coordinates. The programmer then uses this class when rendering maps as images. The other programmer's `Point` class models a point of interest on the map and has attributes specifying the latitude, the longitude, and the description.

When a Ruby interpreter finds multiple class definitions with the same name, it assumes that the programmer's intent is to "reopen" the class defined before and add the new methods to the previous class. This could cause various types of issues down the road, so it's best to put unrelated parts of an app in different **namespaces** in order to reduce a chance of naming collisions. To do this in Ruby we can put separate behavior in different modules, e.g. `Rendering` and `Geo`:

```
module Rendering
  class Point < Struct.new(:x, :y)
  end
end

module Geo
  class Point < Struct.new(:lat, :long, :description)
  end
end
```

Now the full names of the classes would change to `Rendering::Point` and `Geo::Point` so there would be no conflict.

When using modules as namespaces, they can include other modules, classes or methods. Later, one can include the module in other contexts (different modules or classes) to remove the need to fully spell-out their names.

Classes can contain modules, other classes, and methods, but one can't include them in the same way they would include a module.

As **mixins**, modules often group methods that can be included into the other classes. However, we cannot create instances of modules. Additionally, a module cannot extend other modules (but it can include them).

Classes can combine methods, they can be instantiated, and they can be extended by other classes.

In Ruby, almost everything is an object, so classes and modules are objects as well. Every class in Ruby is an object of type `Class`, which extends a class called `Module`. Looking at this inheritance hierarchy, we can clearly see that modules provide just a subset of features provided by classes.

Q: How does Ruby look up a method to invoke?

Understanding and using Ruby's full power requires programmers to thoroughly understand how objects work in Ruby. This is important not only for wielding a great power like metaprogramming, but also to understand concepts that might look weird at first, like definition of the class methods.

Given this snippet of code:

```
mike = Actor.new(first_name: 'Mike', last_name: 'Myers')
mike.full_name
```

Where would Ruby look for a definition of the method `full_name`?

The first place where Ruby would look up the method is in the object's own *metaclass* or *eigenclass*. This is a class that backs every single object and that contains methods defined on that object directly, distinct from all other instances.

For example, the following definition adds a method `full_name` to the metaclass of object `mike`:

```
def mike.full_name
  "#{first_name} #{last_name}"
end
```

Since the metaclass is a class specific to this object, no other instances of class `Actor` would have this method.

If Ruby can't find a method in the object's metaclass, it would start looking for the method in the **ancestors** of the object's class (i.e. the ancestors of `Actor`).

So, what are the ancestors of a class? We can ask Ruby to tell us!

```
Actor.ancestors
# => [Actor, Object, Kernel, BasicObject]
```

We can see that the list of ancestors of any class in Ruby begins with the class itself, includes all ancestor classes (`Object`, ‘BasicObject’), but also modules that are included in any of the classes in the inheritance hierarchy (e.g. `Kernel`). This is something that a Ruby programmer should understand.

Since Ruby 2.0, it’s up to the programmer to decide where to place the modules in the ancestors list.

Let’s take a look at two examples. We’ll assume this is Ruby version 2.0 or above:

Example 1: Including the `FullName` module into the class `Actor`.

```
class Actor < Person
  include FullName
end
```

If we now look up the ancestors list of the class `Actor`, we’ll see that the module `FullName` appears between the `Actor` and the `Person` classes:

```
Actor.ancestors
# => [Actor, FullName, Person, Object, Kernel, BasicObject]
```

Example 2: Prepending the `FullName` module to the class `Actor`.

```
class Actor < Person
  prepend FullName
end
```

By prepending `FullName`, we’ve told Ruby to put the module before the class itself in the ancestors list:

```
Actor.ancestors
# => [FullName, Actor, Person, Object, Kernel, BasicObject]
```

If Ruby searches the entire ancestor list and can’t find the method by the given name, Ruby will internally send another message (method call) to the object: `method_missing?`. The system will repeat the lookup for this method, and will find it at least in the `Object` class (sooner, if the programmer has defined it in an earlier ancestor) and execute it.

Q: Does Ruby support multiple inheritance? Is it needed at all?

Like other modern languages, Ruby supports only single inheritance, and that’s quoted as a feature by Yukihiro Matsumoto, the language’s creator.

There’s no need for Ruby to support multiple inheritance, since everything that could be done with it can be achieved with duck-typing and modules (i.e., mixins).

Since Ruby is a dynamically-typed language, there’s no need for objects to be of a specific type in order for a program to run. Instead, Ruby leverages duck-typing: if an object quacks like duck, and walks like a duck, it’s a duck. That is, we can ask any object if it knows how to respond to a message. Or we can simply send a message and trust the object to figure out how to handle it.

Another use of multiple inheritance is to share code among different classes which cannot be modeled as a chain. To achieve that, we can write the code we want to share as a method in a module and include it into other classes as needed. The concept that allows grouping of methods to be included in multiple classes is the definition of a mixin.

We should note that including a module into a class adds it to the class’ ancestors list:

```
class Actor < Person
  include FullName
```

```
end

Actor.ancestors
# => [Actor, FullName, Person, Object, Kernel, BasicObject]
```

Which means that once the module has been mixed in, there's no difference between the methods that come from a superclass, and those coming from a mixin.

Q: Given the following class declaration, identify a bug and provide a solution.

```
class Worker
  attr_reader :data

  def initialize(d)
    data = d
  end

  def call
    # do a process that requires @data
  end

  private

  def data=(d)
    @data = clean(d)
  end

  def clean(d)
    # return sanitized data
  end
end
```

The problem is in the `initialize` method, which tries to assign the argument `d` using the private attribute writer `data=`. However, the setter won't get invoked, because Ruby will treat `data = d` as a local variable initialization. When Ruby encounters an identifier beginning with a lowercase character or an underscore on a left-hand side of an assignment operator, Ruby will create and initialize a local variable. Please note that this behavior is inconsistent with the way Ruby handles the same identifiers in other contexts: if the identifier does not reference a defined local variable, Ruby will try to call a method with the given name.

To make it clear that we want to call the writer method (i.e. the method ending with the `=` character), we need to prepend the name with `self`:

```
def initialize(d)
  self.data = d
end
```

This may seem counter-intuitive, given the rule that private methods cannot be called with explicit receiver (even if it's `self`). However, the rule comes with an exception: the private writer methods (i.e., methods ending with `=`) can be invoked with `self`.

Another way to fix the bug would be to directly assign the value to the instance variable `@data` in the initializer:

```
def initialize(d)
  @data = clean(d)
end
```

Seasoned Rubyist, or a seasoned programmer learning Ruby?

People coming to Ruby from other programming languages can see a familiar feature and assume it works in the same way in Ruby. Unfortunately, that's not always the case, and there are a few concepts in Ruby that can be surprising to developers who have experience with some other language. The following questions aim to separate experienced Rubyists from people who blindly follow familiar concepts and apply them to Ruby.

Q: Explain the difference between `throw`/`catch` and `raise`/`rescue` in Ruby.

Like most modern Object-Oriented Programming (OOP) languages, Ruby has a mechanism for signaling exceptional situations and handling those exceptions in other blocks of code. To signal an error, a Ruby programmer would use the `raise` method, which aborts the execution of the current block of code and unwinds the call stack looking for a `rescue` statement that can handle the raised exception.

In that sense, the `raise` method performs the same job as the `throw` statement does in C++ and languages inspired by it, whereas `rescue` corresponds to the `catch` block. These operations should be used for signaling the exceptional state and handling it, they should not be used for normal flow control.

However, Ruby also has `throw` and `catch` methods, which can put newcomers off balance, because they *are* used for flow control, and should *not* be used for error handling.

You can think of `throw` as a GOTO statement and `catch` as a method that sets up a label. However, in contrast to traditional GOTO, which can jump to any other point in the program, `throw` can only be used inside a `catch` block, as we'll demonstrate momentarily.

It's important to understand the syntactical differences: `rescue` is a keyword, not a method we could override (`raise` is a method, though). We can combine multiple `rescue` statements one after another to handle different types of errors, and we always put the `rescue` keyword after a block that will, potentially, raise an exception.

For instance:

```
begin
  response = Net::HTTP.get(uri)
rescue Timeout::Error
  puts "A timeout occurred, please try later."
rescue Net::HTTPBadResponse
  puts "Server returned a malformed response."
rescue
  puts "An unknown error occurred."
end
```

On the other hand, `catch` is a method that accepts two arguments: a symbol that can be caught (a "label"), and a block inside which we can use `throw`. The `throw` method can also accept two arguments: the first is mandatory, and should be a symbol that matches the `catch` label to which the program should jump; the second is optional and will become the return value of the `catch` statement. The `throw/catch` pair can be useful when we need to break out of a nested loop, like in this example of looking for a free product:

```
free_product = catch(:found) do
  shops.each do |shop|
    shop.products.each do |product|
      if product.free?
        throw :found, product
      end
    end
  end
end

if free_product
  puts "Found a free product: #{free_product}!"
end
```

Q: Is there a difference between the Boolean `&&`/`||` operators and the written versions `and`/`or`?

There is definitely a big difference between these two. Even though `&&` and `and` (or `or` and `||`) are semantically equal (both being short-circuit Boolean operators), they have different operator precedence.

The English operators `and` and `or` have very small precedence. Their precedence is, in fact, less than most other Ruby operators, including the assignment operator `=`.

Considering that these operators are *the* Boolean operators in Python, seasoned Pythonistas can misuse them in Ruby, which could produce unexpected logical errors. Consider this idiom that's very often used to access an attribute of an object that may not be initialized:

```
actors = [ Actor.new(first_name: 'Mike', last_name: 'Myers') ]
name = actors[0] && actors[0].name
# => "Mike"
puts name
# => Mike
```

If the `actors` array is empty, or contains `nil` as a first element, the Boolean statement will immediately resolve to this value, and the `name` variable will be `nil`. If the first element of the array is truthy, however, we will then send the message `name` to it, and assign the result to the variable `name`.

However, if we used the English version of the operator, we'd get a different result:

```
actors = [ Actor.new(first_name: 'Mike', last_name: 'Myers') ]
name = actors[0] and actors[0].name
# => "Mike"
puts name
# => #<Actor:0x007fa07a1b4a00>
```

We can see that the result of the expression in the second line and the value assigned to the variable in the end are different. Because `=` binds tighter than `and`, the operands were grouped around the assignment first, and only then around the `and` operator. We can illustrate this with parentheses:

```
(name = actors[0]) and actors[0].name
```

The reason to actually have both operators in the language is to allow the use of `and` and `or` as control flow commands in Perl style.

Q: Given below is a snippet of code that calculates a total discounted value among all products that have the discount of 30% or more in all shops. Rewrite this snippet using basic functional operations (map/filter/reduce/flatten/compact) if we assume `shops` is an array of `Shop` objects loaded in the memory, where each `shop` references a the list of `products` in the memory. Which version would you keep in the production-ready code?

```
total_discount = 0

shops.each do |shop|
  shop.products.each do |product|
    if product.discount >= 0.3
      total_discount += product.discount * product.price
    end
  end
end
```

Since we want to transform a list into a single value, this seems to be a job for the `reduce` method (or `inject`, an alias of `reduce`):

```
total_the discount = shops.reduce(0) { |total, shop|
  total + shop.products.reduce(0) { |subtotal, product|
    if product.discount >= 0.3
```

```
      subtotal + product.discount * product.price
    else
      subtotal
    end
  }
}
```

The whole calculation is now a single expression and there's no need to initialize the accumulator variable in a separate line.

However, this approach might be somewhat harder to read, as we've got nested `reduce` operations. To simplify the code, we could first create a list of all products and then reduce it by chaining operations:

```
total_discount = shops.
  flat_map { |s| s.products }.
  reduce(0) { |total, product|
    if product.discount >= 0.3
      total + product.discount * product.price
    else
      total
    end
  }
```

It's possible to further improve readability if we are willing to sacrifice some performance by splitting different kinds of operations into separate blocks:

```
total_discount = shops.
  flat_map { |s| s.products }.
  select { |p| p.discount >= 0.3 }.
  map { |p| p.discount * p.price }.
  reduce(0) { |sum, d| sum + d }
```

It is important to repeat that we've hindered the performance with this change: the new code will iterate multiple times through the same list of products. If this code is not in a performance-critical path, this decrease could be justified by the improvements in readability.

Algorithmically speaking, both operations are linear in time, so for a very large number of products, the style of the code may not be of crucial importance.

On the other hand, we've gained code that declaratively lists steps in the processing of the initial list, similarly to an SQL query.

A Ruby guru should know how to reason about the change, measure the performance of both solutions and make a decision relevant to the context. Your goal here should not be to wait for the "correct" answer, but instead to listen to the programmer to see how she or he judges the various options.

Q: What's the difference between `public`, `protected` and `private` visibility in Ruby?

Method visibility specifiers is an area that often trips up newcomers to Ruby who are familiar with almost any other OOP language.

Public visibility is the default in Ruby, and it behaves just like it does in any other language. Public methods can be invoked by any object that can access the method's object.

However, `private` visibility is different. In Ruby, `private` methods can be directly called only if we don't explicitly specify the message receiver.

Consider this implementation of finding the *n*-th Fibonacci number using recursion with memoization:

```
class MemoizedFibonacci
  def initialize
    @memo = {}
  end

  def get(n)
    @memo[n] ||= calculate(n)
  end

  private

  def calculate(n)
    return 1 if n <= 1
    get(n-1) + get(n-2)
  end
end
```

We have a public method `get` and a private method `calculate`. The public method first checks if the result was previously calculated by looking it up in the `@memo` hash. If it wasn't, it will call the `calculate` method and store the result in the hash.

Let's try to make a slight modification to the `get` method:

```
def get(n)
  @memo[n] ||= self.calculate(n)
end
```

Newcomers know that `self` in Ruby is equivalent to `this` in languages like C++, C# and Java, so they are led to believe this change would have no effect. However, we've now added the explicit receiver, `self`, to the message `calculate`. Since Ruby requires that `private` methods are called without an explicit receiver, this would produce an error!

Another unexpected side effect of this rule is that private methods can be called from subclasses. In fact, many things we consider "keywords" in Ruby, are nothing but private methods in the module `Kernel`, which is included by the `Object` class and which are therefore inherited by every Ruby object. For example, when we raise an exception using `raise`, we're actually calling a private method of a superclass!

Ruby is an open-minded language, so it doesn't even let developers lock down their privates, so to speak.

This leaves the `protected` visibility. It behaves like `public`, but with an exception: protected methods can be called only from the methods of the same class or any of its subclasses. You will notice that in this sense, `protected` behaves in a similar manner to the `private` visibility of other languages. This means that `protected` method visibility should be used in classes which want to hide their state, but allow calling the method from other methods in the class that need it to produce copies or implement comparisons.

The Big Picture

Q: Name at least three different implementations of Ruby. Discuss the differences among them.

The standard Ruby implementation is called **MRI** (short for Matz's Ruby Interpreter) or **CRuby**. As the names suggests, it was written by Yukihiro "Matz" Matsumoto in C. Being the primary implementation of Ruby by its author, it can't be a surprise that it grows together with the language. Even though there is an ISO specification of the language (ISO/IEC 30170:2012), the spec was already obsolete with the release of Ruby 2.0. Thus, all new language features will first appear in MRI, and then they may get implemented in other interpreters. Being written in C, MRI can interoperate with other C code, and run gems written in C as well.

One of the most commonly cited problems with this implementation is the lack of support for true parallelization, since MRI Ruby supports only green threads and depends on a Global Interpreter Lock (GIL).

Rubinius is another implementation, based on LLVM, and written in C++ and Ruby. It improves concurrency by using native threads and a just-in-time (JIT) compiler. Additionally, the core library is mostly written in Ruby itself, making it easier to understand the internals, especially to folks not very comfortable with C.

The coupling between MRI and the language makes it hard to keep other implementations of Ruby up to date, which means they may lack features found in the recent versions of MRI. One of the first things the Rubinius team made was RubySpec: a runnable specification of the Ruby language, which would allow developers of the language's forks to check their implementation against the "standard". Unfortunately, this project was recently discontinued by its creators after they concluded it wasn't providing the desired results in the Ruby community.

Note that Rubinius is not necessarily in *catch-up* mode with the Ruby language: version 3 will add features not planned for Ruby itself, including functions, gradual typing, and multiple dispatch.

Those who need to interface Java code with Ruby may find **JRuby** helpful. Like Rubinius, it offers improved concurrency options by relying on native threads, and JIT compilation of bytecode into machine code. As a bonus, it provides interoperability with existing Java Virtual Machine (JVM) code (you can use native Java classes and libraries) and the possibility to run Ruby on any JVM.

Since JRuby relies upon JVM, it means it can't use Ruby gems written in pure C. While this implementation improves runtime speeds as a rule, it introduces slow starting time of applications.

Another noteworthy implementation is **mruby**: an embeddable subset of the Ruby ISO standard. Matz himself is leading its development, with the goal of enabling Ruby to run as an embedded language inside existing apps and games, providing scriptability and automation, and thus challenging Lua.

You can read much more about other Ruby implementations in our blog post about Ruby interpreters and runtimes.

Wrap-Up

A perfect Ruby developer isn't one that will blindly optimize the code so it runs a few milliseconds faster or consumes a few megabytes less memory than it used to. The perfect candidate will know how to achieve that when performance is of essence, but will be able to recommend alternatives if it would sacrifice too much readability and maintainability. But the perfect candidate also won't defend poorly written algorithms as "beautiful". In Ruby, one could say, perfection is in compromises and good judgment calls. Remember that when talking to your candidates, and listen to their reasoning.

Happy interviewing!

Hire Ruby developers now
See also: Toptal's growing, community-driven list of essential Ruby interview questions.

- Trusted by:
- Hewlett Packard Enterprise
- airbnb
- J.P.Morgan
- Emirates