Ruby Methods

This is a long lecture. It will take 2 sessions to get through it all.

Prerequisites

Ruby variables

You need to know Ruby variables to write Ruby methods.

What you'll learn

- Ruby methods
- Test-driven development (TDD)
- Ruby blocks
- Ruby Enumerators and Ranges
- Ruby modules
- Optional and default method arguments

We'll probably get to blocks and enumerators in the first session. We'll cover the rest in the 2nd session.

What is a method?

```
"".methods.sort
```

What is a method?

We've been using methods for a while. We started calling them "verbs", but Ruby calls them methods. We know how to generate a list of methods using a method called "`methods`". And we know how to sort that list.

But what *is* a method?

Method

```
"".size

"".method("size")
"".method("size").class
"".method("size").methods

"".method(:size)
```

Take String's `size` method for example. It returns the size of a String in characters.

We can use a method named "`method`" (singular) to get more information about the method itself.

Using our introspection skills, we can see that a method is a data type called `Method`. This data type even has it's own methods. Methods have methods!

We can also get information about methods using symbols instead of strings.

Creating Methods

```
def add_two(number)
  puts number + 2
end

add_two(1)
```

We can create our own methods in Ruby. Here, for example, I've created a method to add 2 to a number.

Once I've defined a new method like this, I can immediately start using that method within the same Ruby program.

Creating Methods

```
def add_two(number)
  puts number + 2
end

add_two(1)
```

A method definition starts with the word "`def`" and ends with "`end`". After `def` comes the method name. Then, in parentheses, comes a list of named arguments that the method accepts. When arguments are passed to the method, they are assigned to these variables.

The stuff in the middle is just a block of code, just like the code blocks we saw in `if`-statements.

Inside the code block, whatever argument you pass to the method when you call it (in this case, 1), is assigned to the variable `number` in the method definition.

Some terminology:
I've *defined a method* named "`add_two`". The "`add_two`" method *accepts 1 argument* named `number`. The method takes that argument and adds 2 to it.

Once defined, I can *call* (or *invoke*, or *execute*) the method "`add_two`" and *pass it the argument* 1.

Variable Scope

```
def add_two(number)
  puts number + 2
end

puts number
```

The code block within a method works exactly like the code block we saw in the `if`-statement with one exception. The code block has access to a variable named `number`. The variable is *injected* into the block (more new terminology).

Moreover, the injected variable `number` is not *accessible* outside the method.

This code, for example, generates an error. In the last line, Ruby doesn't know what the variable `number` is. `number` is not *accessible* from this point in the code. It's only *accessible* from within the method's code block.

Since the variable `number` is not available outside the method `add_two`, we describe the variable as *scoped* to the method. We'd say the *scope* of the variable doesn't extend beyond the code block.

Lots of new terminology today.

Method Advice

```
# Adds 2 to the number.
def add_two(number)
  number + 2
end

puts add_two(1)
```

Some advice for creating methods:

1. Just like variables, give your methods good, **descriptive names**. You may end up using them far away from where you've defined them. Name your methods so you can remember what they do when you see them used out of context.

2. **Always add a comment** above your method explaining what it does. For simple 1-line methods like this one, the comment is overkill. But once you start writing longer methods, you should summarize what the method does in a comment rather than forcing yourself to re-read all the code to figure out what it does. Don't force yourself to think too hard when you stumble upon this method later.

3. **Limit the amount of work** you do in a method. For example, I've slightly edited this method to move the `puts` call outside the method body. The method name is `add_two`, not `add_two_and_puts`. *All* it should do is add 2 to the number. If you're ever tempted to use the word "and" in a method name, that's a bad sign.

Btw, these "bad signs" are called "code smells". A "code smell" is the term we use to describe when your "coder conscience" is telling you something's wrong. You see some code that matches a pattern, and you don't like it.

Recognizing a pattern that leads you to a minor refactoring isn't a code smell. Some refactorings are purely for readability purposes. And readability is relative.

A code smell is stronger than that. Smelly code is code that you suspect is going to cause a problem later. It's code that you know you're going to have to rewrite later.

Why do you know you'll have to rewrite it later? Because experience. Because experience tells you that you're probably going to use this method in a way that doesn't require outputting text. Experience tells you that having a `puts` there is going to be annoying. You'll be annoyed seeing some output displayed every time to want to use this method. Because `add`ing`_two` and `puts`ing just don't seem like they should *always* go together. Most times you'll just want to want `add_two` to something, without generating any output.

Because experience.

Defensive Programming

```
# Adds 2 to the number.
def add_two(number)
  (number + 2) unless number.nil?
end

puts add_two(nil)
```

Writing a method requires us to take other people into consideration. The methods we write today will likely be used by other people on our team tomorrow.

The homework introduced the `gets` method as a way to interact with a user in a Ruby program. Up until now, you've probably assumed your user would be polite. We shouldn't assume that. In fact, that's a dangerous assumption.

Instead, assume all users are dumb or *evil*. This comes naturally to many anti-social computer-types, but try anyway. Write code with the expectation that the person using it will be too dumb to understand what's going on or will be actively trying to break your program (perhaps for personal gain).

(Side note: Now is a good time to consider rewriting some of your homework projects.)

The most common way to mess around with one of your methods is to pass it **unexpected arguments**.

For example, passing a `nil` argument to our `add_two` method will break it. You can't add `nil` and 2.

So what do we do about it?

One solution is to just let the code break. Hopefully the user will see the Ruby error

and understand what to do. Probably not, but one can dream.

A better solution is to be proactively defensive, like in this example. Check if the `number` is `nil`. If it is, skip the line of code that would otherwise break. It's better to do nothing than let your users see an ugly and cryptic Ruby error.

Defensive Programming

```
# Adds 2 to the number.
def add_two(number)
  if number.class == Fixnum
    number + 2
  end
end
```

Unfortunately, checking for `nil` only solves one part of the problem. What if the argument is an `Array` or a `Hash`. Those arguments will break the program as well. Do we just keep adding checks for every type of data in Ruby?

Let's step back and rethink this. Instead of listing every negative example (every argument that could break our method), let's focus on the positive examples (the arguments that *won't* break our code).

In other words, what's the *minimum requirement* for the `number` argument to get this method to work?

`number` must be a Fixnum. Anything else is probably going to break our method. So this code kinda works.

Except there's a bug. By restricting the input to `Fixnum`s, we've inadvertently dropped support for other numbers in Ruby, like `Float`s and `Bignum`s. Those numbers don't work anymore. But they should. We need to widen the net.

It's a careful balancing act. How do you let in *all* of the good arguments while ignoring *all* of the bad ones?

Duck Typing

```
# Adds 2 to the number.
def add_two(number)
  if number.respond_to? :+
    number + 2
  end
end
```

The “Ruby way” to validate an argument to a method is to use the `responds_to?` method. I say “the Ruby way” because other programming languages handle this problem differently.

What we’d like to say in our code is, if the `number` argument “responds to” the `+` method, then it should be safe to use it.

How does `respond_to?` work? The `respond_to?` method ends with a question mark, so it should return a boolean (and it does). It accepts as an argument a method name. You can specify the name as a symbol or string. We’re using a symbol here.

`respond_to?` returns `true` if the method you’re looking for exists. Otherwise it returns `false`.

This type of defensive programming validation uses a technique called “duck typing”. Instead of checking if the `number` variable is a “duck”, we will just assume “If it looks like a duck and quacks like a duck, then it’s a duck”.

We don’t care if `number` is a `Fixnum`, `Bignum`, or `Float`. What we *really* care about is that it supports the `+` method. If it does, we can use it without generating a Ruby error.

Defensive Programming

```
# Adds 2 to the number.
def add_two(number)
  if number.respond_to? :+
    if number.respond_to? :push
      number.push 2
    else
      number + 2
    end
  end
end
```

You can usually stop after doing a “duck type” check. But we’re still not done fixing this method.

Arrays support the + method. But you can’t add 2 to an Array. So what do we do?

We can just let Arrays cause errors

Or we can try to fix our method, like this. When someone passes an array to `add_two`, we can assume that what the user probably *means* is to push a 2 onto the array. That’s our best guess at the user’s *intent*.

This code is getting pretty messy. We had to write so many extra lines of code to make sure dumb and evil users wouldn’t break our program. Now whenever our method sees something it doesn’t expect, it makes a best effort to do something useful. If it can’t, it doesn’t do anything.

Test-Driven Development

```
puts add_two(1)
puts add_two(1.0)
puts add_two(nil)
puts add_two({})
puts add_two([])
puts add_two(true)
```

Programming defensively can get confusing. Do you remember all the different ways we can use our add_two method? Here are 6 different ways that work. It would be nice to keep track of these tests somewhere other than our own heads.

TDD

Red

Break your method

Green

Fix your method

Refactor

Rewrite your method.

Test-driven development (TDD) is a coding technique that helps us keep track of all of these test cases. It requires us to start every new task by writing some code that doesn't work. Then we write enough code to fix the bug. Then, if necessary, we refactor.

So there are three steps.

Step 1: The "red" step
Write some code that breaks your method.

Step 2: The "green" step.
Fix your method so it doesn't break.

Step 3: The "refactor" step.
If necessary, refactor your method to make it more readable, maintainable, or faster.

TDD

```
def test
    puts add_two(1)
    puts add_two(1.0)
    puts add_two(nil)
    puts add_two({})
    puts add_two([])
    puts add_two(true)
end
```

When using TDD, it’s common to collect all your test cases in a method. The test method tests all the different cases we can think of. This test passes when you can call the “test” method without triggering an error.

TDD

```
puts add_two(“”)
```

We haven’t tried calling our method with a String. So let’s add that to the test cases and see what happens.

The test fails! Oh no! We’ll have to fix that in our project work.

TDD as Documentation

```
puts add_two(1)
puts add_two(1.0)
puts add_two(nil)
puts add_two({})
puts add_two([])
puts add_two(true)
puts add_two("")
```

Writing tests are a great way to help you remember what your method actually does.

TDD as Documentation

```
# Add 2 to the number.
def add_two(number)
  ...
end
```

Recall that the only comment we added to the method was “Add 2 to the number”. That leaves the user with a lot of questions. What if the number isn’t actually a number? How does this method work for those arguments?

TDD as Documentation

```
puts add_two(1)
puts add_two(1.0)
puts add_two(nil)
puts add_two({})
puts add_two("")
puts add_two([])
puts add_two(false)
```

Looking at the test results, you can get a much more complete idea of what the method does.

Comments, while nice to have, can get out of date. We added a bunch of lines to our method, but we forgot to update the comment.

A test can't be out of date. As long as you run it regularly, it will tell you exactly how the method works.

Method Pairs

```
def add_two
  …
end

def test_add_to
 …
end
```

We'll be talking about testing a lot more later. Suffice to say that, at this point, I expect you all to write all methods in pairs. One is the method itself. The other is a test method.

Method Returns

```
def one
  1
end

def one
  0+1
end
```

Methods return whatever the last line of code returns. So if we wanted to create a method that returns 1, we would just make sure the last line was an expression that evaluated to one. An expression can be a plain value or a method call. Any of the Ruby phrases we introduced before we starting talking about variables can be used.

This is called an implicit return.

Method Returns

```
def find_ten
    i = 0
    loop do
        if i == 10
            return i
        end
        i += 1
    end
end
```

Methods also support explicit returns. If you use the return keyword, you can exit a code block immediately and return a value. In this case, for example, we start counting from 0 and stop when we count to 10. When we reach 10, we return it, which both breaks the loop and exits the method entirely.

Explicit returns are great when you're, for example, creating a method that searches an array for something. Using return allows you to break the search off early and stop the method, which will save you some time if the item you're looking for appears near the beginning.

Blocks

```
begin
    puts "Hello world"
end
```

Up to now we keep talking about this thing called a code block. if-statements allow you to pick between two different code blocks. while-statements allow you to loop over the same code block multiple times. And code-blocks themselves can control how Ruby flows through them using keywords like break and return.

Blocks

```
block = begin
    puts “Hello world!”
    0
end

puts block
```

Methods are essentially named code blocks. They assign a block to a variable. puts-ing the block here outputs the result of executing the block, not the block object itself.

The point is that blocks have two personalities. They can be methods, code that does something, or data, code that holds something.

We’ve seen how to use blocks as methods. But Ruby really shines when we begin to use blocks as data - for example, when we start using blocks in places where we’d use variables, like as an argument to another method.

The Block Argument

```
5.times do
  puts "Hello"
end
```

One of the things Ruby does really well that other languages don't is the way it handles the concept of code blocks with split-personalities. All languages have the ability to easily define code blocks as methods. But very few have such an elegant way of using code blocks as data.

You may have seen in the documentation that certain Ruby methods accept a block as an argument. This is what that looks like.

Here we see that the number 5 has a method named "times". That method accepts a code block as an argument. The block begins with do and ends with end. In classic "Ruby reads like English" style, this code will execute the code block 5 times.

The Block Argument

```
def say_hello(argument)
    puts "Hello #{argument}"
end

say_hello_method = method(:say_hello)
5.times &say_hello_method
```

You can use methods as code blocks, but it's awkward. In this example, when you type "say_hello", Ruby isn't sure if you mean "say_hello" the method or "say_hello()" the result you get when you execute the method. So you have to jump through some hoops to use the method name as an argument. This is complicated and not recommended.

The Block Argument

```
5.times do
  puts “Hello”
end
```

This is much simpler.

Block One-Liner

```
5.times { puts “Hello” }

5.times {
  puts “Hello”
  puts “World”
}
```

If the code block is small enough, you can take advantage of this one-line refactoring. This is inspired by C, which uses curly brackets to indicate the beginning and end of code blocks.

Note that, in the Ruby documentation, blocks arguments are shown using this shortened syntax.

Some people use curly brackets even if the code block is multiple lines. That works as well and it saves you from having to type “do” and “end”. This allows you to boost your productivity by reducing the amount of keypresses you need to get the job done. Senior devs use tricks like this, but it can sometimes make your code harder to read. I’ll leave the decision to use this (or not use this) up to you.

Block Variables

```
5.times do |number|
    puts "Hello #{number}"
end

puts number
```

Like methods, blocks can accept scoped variables. Unlike methods, the syntax for introducing variables into a block is different.

The “times” method, for example, sends a variable into the block. If we don’t declare the variable, it’ll be ignored. We don’t have to use it if we don’t want to.

But if we want to use the argument being passed in, we need a place to put it. We create one by declaring a variable between bars. That allows us to use the variable in our block.

Like with methods, this variable is scoped to the block. If we try to access it outside the block, we’ll generate an error.

Block Variables

```
5.times { |number|
    puts "Hello #{number}"
}
```

This is what the code looks like with the shorthand block syntax.

Also note, in classic Array style, that the number starts counting at 0. So 5.times passes 0,1,2,3,4 into the block, not 1,2,3,4,5.

Enumerators

```
5.times.class
([].methods - 5.times.methods).count

5.times.to_a == [0,1,2,3,4]

5.times[2]
5.times << 5
```

5.times, on it's own, returns a new data type we haven't seen before: an Enumerator.

An Enumerator is very much like an Array, but lighter. It's similar to the relationship between Strings and Symbols. Enumerators have 63 fewer methods than Arrays.

What 5.times does, behind the scenes, is create something like an Array. In fact, you can convert the 5.times Enumerator into an Array and see that it's equal to an Array of 5 numbers, counting from 0 to 4.

But the Enumerator does not support indexing or adding new items. Symbols are built for naming things. Enumerators are built for enumerating lists, one-at-a-time.

Enumerators and Ranges

```
5.times.to_a == (0..4).to_a

(0..4).to_a == [0,1,2,3,4]

(0...4).to_a == [0,1,2,3]
```

How does “times” convert a single number into an Array? It creates something called a Range, a close sibling of the Enumerator.

A range is a quick way to take two numbers and quickly generate a list starting from one and ending at the other.

So 0-dot-dot-4 is a quick way to create an Array-like object containing [0,1,2,3,4].

Add another dot, 0-dot-dot-dot-4, and you create a Range that extends up to, but not including, the last number.

Ranges

```
(1..100).to_a

("a".."z").to_a

(:a..:z).to_a
```

Ranges are fantastic ways to quickly generate long lists of items. Count up to 100? No problem.

Any data type that supports the “next” method can be used in a range, including letters and symbols. So you can generate all the letters of the alphabet easily.

Enumerators and Blocks

```
[1,2,3,4,5].each { |number|
  puts "Counted to #{number}..."
}

(1..5).each {|i| puts i }
```

Enumerators and blocks combine to form my favorite feature of the Ruby language. Remember when I said that almost everything on the web is an Array combined with an if-statement?

Google: a list of search results *if* they match my search term
Twitter: a list of tweets *if* I'm following the tweeter
Facebook: a list of activities *if* I'm friends with the Facebooker

The place in Ruby where Arrays and booleans meet is the Enumerator block.

First, meet the method "each". "each" returns an Enumerator that yields each item in an Array to a block. So, in this case, each number from 1 to 5 is yielded to the block. Of course, I could do the same thing with less code using a range.

Enumerators

```
(1..5).each {|num|
  if num.even?
    puts "Even"
  else
    puts "Odd"
  end
}
```

Within the code block I can add an if-statement. So here I can print “Even” if the number is even?. Otherwise I can print “odd”.

Obsolete For Loops

```
for num in (1..5)
  puts num
end

(1..5).each { |num|
  puts num
}
```

For probably the last time, let me show you the Ruby for-loop. As you can see, the for loop does pretty much the same thing an Enumerator does. I can loop from 1 to 5. Or I can loop over every item in an Array, Range, or Enumerator.

So for-loops are redundant. But there's an even better reason why they're not used in Ruby.

Enumerators

```
(1..5).each {|num|
  if num.even?
    puts "Even"
  else
    puts "Odd"
  end
}
```

The main reason the `for`-loop is obsolete is because there are a bunch of other useful methods that look exactly like `each`, but with extra features.

For example, there's an `each`-like Enumerator method we can use to refactor this code.

This code is enumerating over a list of numbers (1 through 5). For every even number it outputs "Even". For every odd number it outputs "Odd". Pretty simple.

Map

```
result = (1..5).map do |num|
  if num.even?
    "Even"
  else
    "Odd"
  end
end

puts result
```

This is another way to do the same thing using the `map` method.

The `map` method enumerates over the range, one item at a time, and *replaces* each item in the list with the results of the code block. The resulting array is stored in the variable `result`.

At the end I `puts` the `result` array to get the same effect from before - output “Even” for all even numbers and “Odd” for all odd numbers.

Compared to the last slide, I’ve refactored the repeated `puts` from the code block. My code is a little DRYer.

Now that the code block is simpler, an even better refactoring becomes apparent.

Map

```
puts (1..5).map do |num|
  num.even? ? "Even" : "Odd"
end
```

The `if`/`then` in the code block is now small enough to be refactored into a single line small enough to fit on this slide.

I've refactored the previous program (2 slides ago) from 7 lines of code (LoCs) down to 3 LoCs.

It could even be 1 LoC if I replaced the `do`/`end` with curly brackets and had enough space on this slide.

So `map` allows me to *transform* one array into another array.

TODO: example showing how these things can be chained together

Select

```
puts (1..5).select { |num|
  num.even?
}

puts (1..5).reject { |num|
  num.even?
}
```

Instead of *transforming* an array, what if I want to *filter* the array.

Say that, given a list of numbers, I just want to output the list of even numbers. No problem.

I can use an Enumerator method named `select`. It returns only those items in the list that cause the block of code to return `true`.

btw: `find_all` and `select` do the same thing. If `find_all` is easier for you to remember, use it. Either you want to “select all the items that match the condition” or “find all the items that match the condition” - whichever is more readable to you.

The opposite of `select` is `reject`. So if I only want the odd numbers, I can just replace `select` with `reject`.

Notice that I’m now combining the concepts of lists and boolean logic without ever writing an `if`-statement. Ruby is doing that work for me, resulting in smaller and potentially more readable code.

Like I said before, most of the internet is filtered lists of items: lists + some boolean logic. Enumerating over a list of items and checking a boolean expression (e.g. 3 slides ago) is a very common pattern. Ruby acknowledges these patterns and

introduces shortcuts to make developers' lives (your lives) easier. Developer happiness.

Any? and All?

```
puts [1,2,3].any? {|n| n.even? }
puts [1,2,3].all? {|n| n.even? }
puts [1,2,3].one? {|n| n.even? }
puts [1,2,3].none? {|n| n.zero? }
```

If I want to check if any item in the list is even, I can use `any?`

If I want to check if all items in the list are even, I can use `all?`

If I want to check if only one of the items in the list are even, I can use `one?`

If I want to check if none of the items in the list are zero, I can use `none?`

In each these examples, I've been saved from having to write an `if`-statement, which makes my programs shorter and possibly easier to read. Are "any items even"? Are "all items even"? Is "one item even"? Are "no items even"?

There are a bunch of these kinds of `Enumerator` methods. And they all follow the same pattern. Any list (e.g. an `Array` or `Range`), a method, and a block of code.

So the main reason why for-loops are obsolete in Ruby? Ruby coders are so used to using this powerful pattern for processing lists of items that they forget the for-loop syntax even exists.

Methods for Methods

```
def puts_block
  puts yield
end

puts_block { "Hello world!" }
```

Would you like to write a method that accept a block parameter?

The only thing you need to do is use the keyword “yield” where you’d like the block to be in your method. That’s it. You don’t even need to specify an argument list after the method name.

The reason why this is so simple is that Ruby only allows you to specify 1 block per method. So when you yield, Ruby only has one place to go.

Methods for Methods

```
def puts_block
    if block_given?
        puts yield
    else
        puts "No block given."
    end
end

puts_block { "Hello world!" }
puts_block
```

Would you like to make the block optional?

The block_given? method lets you know if a block has been passed as an argument. If it has, it return true.

Methods for Methods

```
def puts_hello_wyncode
    if block_given?
        puts yield "Wyncode"
    else
        puts "No block given."
    end
end

puts_hello_wyncode { |name| "Hello #{name}" }
```

Would you like to pass an argument to the block? Just attach it to the end of the yield.

Methods for Methods

```
def puts_hello_wyncode
    if block_given?
        puts yield "Hello", "Wyncode"
    else
        puts "No block given."
    end
end

puts_hello_wyncode { |greeting, name| "#{greeting} #
{name}!" }
```

Would you like to pass a multiple arguments to the block? Just attach them all to the end of the yield, separated by commas.

Writing a method that accepts a block isn't very common. But when it happens, it can be pretty awesome (as we'll see later when we get to Sinatra).

Modules

```
module AlphabetTesters
  def self.a?(letter)
      letter.downcase == "a"
  end

  def self.b?(letter)
      letter.downcase == "b"
  end
end
```

As you write more and more methods, you'll want to start organizing those methods in some way. A group of related methods can be packaged up into something called a Module. A module starts with the word "module", then the module name, and ends with the word "end". Inside the module is a code block containing method definitions. A module can also contain constants.

Each method definition must start with the word self. It's very repetitive and thus not-DRY, but it's there for a reason, which we'll find out later.

Modules

alphabet_testers.rb

```
module AlphabetTesters
   ...
end
```

Modules are usually located in their own files. So let's move the AlphabetTesters module to a new file called alphabet_testers.rb. Note that Module names are CamelCased (meaning that the first letter of each word is capitalized so it looks like the humps on a camels back)

File names are lowercased and underscored. Don't forget to end all your files with .rb.

Modules

```
require "./alphabet_testers.rb"

puts AlphabetTesters.a?("a")
```

Once you've got that module file created, it's time to show you, for the first time, how to relate Ruby code in two different files to one another.

The require method tells Ruby that the code in one file requires the code in another file. This method accepts a String representing the filesystem path to the file. In my Ruby file I want to require the code in the alphabet_testers.rb file located in the current directory. The dot at the beginning of the filename is shorthand for the current directory.

Once I've required the file, I can call methods defined in the module. Note that I have to prefix the method call with the module name. This prefix is called "namespacing". Within the AlphabetTesters module I've defined a space containing the names of my new methods.

Namespacing is useful because it prevents me from accidentally overwriting methods in the Ruby standard library.

Namespacing

```
module AlphabetTesters
  def self.print(something)
      puts "Printing this #{something}"
  end
end
```

Within our module, let's define a print method that doesn't do the same thing as the Ruby print method.

Namespacing

```
require "./alphabet_testers.rb"
print "a"
AlphabetTesters.print("a")
```

When I require the module, I can still use Ruby's print method. I didn't break it. If I want to use my new print method, I need to access it from the module's namespace.

So locating new methods in modules is a safer way of introducing new methods to the Ruby standard library.

Including

```
Kernel.puts "Hello"
puts "Hello"
```

I've already mentioned previously that the Kernel module adds methods like "print" and "puts" to the Ruby standard library. But when I use those methods, I don't have to prefix them with the module name. How did that happen?

Including

```
require "./alphabet_testers.rb"
include AlphabetTesters

puts b?("b")
```

If I include my module after I require the file it's in, I can use the module methods without the prefix.

But wait, not so fast...

Including

In alphabet_tester.rb:

```
def ~~self.~~b?(letter)
    letter.downcase == "b"
end
```

If I access the method through the namespace, I need to define the method with the word self. But if I want to access the method without the namespace, I need to define the method the old way, without the self.

Including

```
require "./alphabet_testers.rb"
include AlphabetTesters

puts b?("b")
puts AlphabetTesters.a?("A")
```

This will work now.

So including is inherently a little more risky than requiring. Including tells Ruby to essentially copy-and-paste the content of the module directly into the spot where I'm calling the "include" method. Anything defined in the module is now defined in my current workspace. Any methods that don't start with "self." can be used right away. Any methods that do start with "self." still require the namespace.

Including

```
def b?(letter)
    letter.downcase == "b"
end

def self.b?(letter)
    letter.downcase == "b"
end
```

Kernel.print and just plain print both work in Ruby because the Kernel module defines both print and self.print. So if we define both a b? and and a self.b? method...

Including

```
require "./alphabet_testers.rb"
puts AlphabetTesters.b?("b")

include AlphabetTesters
puts b?("b")
```

That allows us to use both the namespaced and non-namespaced versions of the b? method.

The require gives us access to the namespaced version of the method.
The include gives us access to the plain version of the method.

Default Arguments

```
def add_two(number)
  number + 2
end
```

Let's go back to plain old methods for one final lesson.

Recall the first version of our add_two method. It took a number passed in as an argument and added two to it. As part of the lesson on Defensive Programming, we had to add a bunch of code in case the number argument was something unexpected: for example nil or an Array or a String or a Hash, and so on. That made the simple process of adding 2 to something much more complicated if we care to not have our methods triggering errors.

However, there was one more way a dumb/evil user could still break our method.

Default Arguments

```
puts add_two
```

wrong number of arguments (0 for 1)

Someone could try to call our method with the wrong number of arguments, or even no arguments at all. This will trigger an error. If we care about this error, we should do something about it.

Default Arguments

```
def add_two(number = 0)
  number + 2
end
```

The solution is this. Ruby allows you to make method arguments optional. If you set the argument variable equal to a value in the method definition, then Ruby will use that value if the user doesn't specify one at all.

Default Arguments

```
puts add_two(1)

puts add_two
```

Now when a dumb/evil user tries to call our method with no arguments at all, we'll just assume he meant 0 and return 2.

Variable Length Args

```
puts add_two(1, 2)
```

wrong number of arguments (2 for 0..1)

But here comes that dastardly user again. Now he's trying to pass multiple arguments to our method, and that's causing an error as well.

Variable Length Args

```
def add_two(number = 0, *rest)
  number + 2
end
```

If we care about this error, we can update our method again. If the final variable in your argument list begins with an asterisk, then all the arguments passed to the method get stored in an Array with that name.

The argument doesn't have to be called rest, but that's a common name for the "rest of the arguments".

Variable Length Args

```
def add_two(number = 0, *rest)
  if rest.size > 0
    puts "Seriously? #{rest}"
  end
  number + 2
end
```

Since it's an Array, we can check it's size to determine if the user is messing with us. We can even get the superfluous arguments and do something with them. For example, we can mock the user for messing around with our method.

Optional Arguments

```
puts add_two(1)
puts add_two
puts add_two(1, 2)
```

Now all three of these tests pass.

Optional Arguments

```
[].push(1,2,3,4)
```

Both of these techniques, allowing arguments to be optional and supporting any number of arguments, are not just things you can add to your methods defensively. You may want to use these features in your methods.

For example, the Array.push method accepts any number of arguments. It uses the “rest” argument to capture all the arguments, no matter how many you specify, and add them to the Array. That’s a useful method. You may want to do something like that one day.

Optional Arguments

```
def add_two(number = 0,
  strings_are_ok=false,
  arrays_are_ok=false,
  nils_are_ok=false)
  number + 2
end
```

Maybe a list of arguments isn't good enough. Maybe you'd like to write a method with a bunch of options, all of which are optional.

Take our add_two method, for example. The full defensive version had all sorts of checks to see if the argument is an Array or a String. Maybe, instead of doing all those checks, you can let the user tell you whether he's ok if you return an error. By default you can assume unexpected data is not ok, so errors are always ok. But if the user says errors are not cool with him, you can use extra arguments to create an if-statement where you can try to figure out what to do.

This list of optional arguments can get pretty long. And you may have to constantly update it as you learn about new types of data. That's annoying.

Optional Arguments

```
def add_two(number = 0, options = {})
  number + 2
end
```

Instead, the Ruby convention is to put all those extra arguments in a hash. The hash, like the asterisk-rest argument, will capture all the additional arguments for the method. But, better than *rest, it'll not only store the values of the arguments, but also the names.

Optional Arguments

```
puts add_two(1)

puts add_two("1", {strings_ok: true})
puts add_two([1], {arrays_ok: true})

puts add_two("1", strings_ok: true)
puts add_two([1], arrays_ok: true)
```

Calling a method with an options hash looks like this. You make the final argument to the method a literal Hash object.

If the final argument to the method call is a Hash, don't forget that Ruby allows you to ignore the curly braces.

Optional Arguments

```
def add_two(number = 0, options = {})
  if options[:strings_ok]
    # support string arguments
  elsif options[:arrays_ok]
    # support Array arguments.
  else
    number + 2
  end
end
```

Then you have to update your method to support all the different options you'd like to support. This logic takes advantage of the fact that, if an argument isn't specified in the hash, the lookup returns nil, which is falsey.

Options Defaults

```
def add_two(number = 0, options = {})
  default_options = {strings_ok: false, arrays_ok: false}
  options = default_options.merge(options)
  # option if statements
end
```

If you'd like, you can create a hash of the default options to simulate having all the default options in the method definition. Then you can use the Hash.merge method, which will merge your defaults with the user's options. That means if the user has specified as option, use that one. Otherwise use the default.

In this case, however, since all the defaults are false, it's not necessary. Nil is already falsey.

But if you'd like to have a default option that's not falsey (false or nil), then a default_options hash is the way to go.

All Together Now

```
# Makes heroic efforts to add two to your argument.
def add_two(number = 0, options = {}, *rest)
  # merge with default options
  # keep or ignore the "rest" of the arguments
  # check the options
  # if the option is on, figure out what to do
    # e.g. check respond_to? and look for alternatives
  # just add two
end
```

All together, here's a fully-stacked method. This is a method that does everything right. It's making a maximum effort to make sure that users can't break it.

… and it's terribly designed.

You can be too defensive. Sometimes you have to just let errors happen. If you don't want to handle a particular case, just add a note to the comment letting the user know that weird stuff just isn't supported.

Let It Be

```
# Adds 2 to the number,
# which it assumes is numeric.
def add_two(number)
  puts number + 2
end
```

This method is just fine. It let's the user know that, if he tries any funny business, he's responsible for the consequences. Most methods start out this way. Over time they grow more complex as you begin to realize how dumb or evil your actual users are. But if you can keep things under control, this may be good enough.

Everyone's comfort level with defensive programming is different.

Just A Little

```
# Adds 2 to the number,
# which it assumes is numeric.
def add_two(number)
  puts number + 2 unless number.nil?
end
```

At the very least, I prefer to check if my arguments are nil. Nils have a nasty habit of showing up in unexpected places. In my code, I would consider anything more than that, for a first draft, to be overkill.