Using Blocks in Ruby: A Brain-Friendly Report

- Do heavy lifting easily with blocks
- Get more done with less code
- Bend your mind around blocks and methods
- Avoid embarrassing mistakes

Jay McGavren
Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com
Using Blocks in Ruby

Wouldn’t it be dreamy if there were a book on Ruby that didn’t throw blocks, modules, and exceptions at you all at once? I guess it’s just a fantasy...

Jay McGavren
A block is a chunk of code that you associate with a method call. While the method runs, it can invoke (execute) the block one or more times. Methods and blocks work in tandem to process your data. Blocks are a way of encapsulating or packaging statements up and using them wherever you need them. They turn up all over Ruby code.

**Blocks are mind-bending stuff. But stick with it!**

Even if you’ve programmed in other languages, you’ve probably never seen anything like blocks. But stick with it, because the payoff is big.

Imagine if, for all the methods you have to write for the rest of your career, someone else wrote half of the code for you. For free. They’d write all the tedious stuff at the beginning and end, and just leave a little blank space in the middle for you to insert your code, the clever code, the code that runs your business.

If we told you that blocks can give you that, you’d be willing to do whatever it takes to learn them, right?

Well, here’s what you’ll have to do: be patient, and persistent. We’re here to help. We’ll look at each concept repeatedly, from different angles. We’ll provide exercises for practice. Make sure to do them, because they’ll help you understand and remember how blocks work.

A few hours of hard work now are going to pay dividends for the rest of your Ruby career, we promise. Let’s get to it!
Defining a method that takes blocks

Blocks and methods work in tandem. In fact, you can’t have a block without also having a method to accept it. So, to start, let’s define a method that works with blocks.

(On this page, we’re going to show you how to use an ampersand, &, to accept a block, and the call method to call that block. This isn’t the quickest way to work with blocks, but it does make it more obvious what’s going on. We’ll show you yield, which is more commonly used, in a few pages!)

Since we’re just starting off, we’ll keep it simple. The method will print a message, invoke the block it received, and print another message.

```ruby
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
```

This method takes a block as a parameter!

If you place an ampersand before the last parameter in a method definition, Ruby will expect a block to be attached to any call to that method. It will take the block, convert it to an object, and store it in that parameter.

```ruby
def my_method(&my_block)
  ...
end
```

When you call this method with a block, it will be stored in “my_block”.

Remember, a block is just a chunk of code that you pass into a method. To execute that code, stored blocks have a call instance method that you can call on them. The call method invokes the block’s code.

```ruby
def my_method(&my_block)
  ...
end
```

No ampersand; that’s only used when you’re defining the parameter.

Okay, we know, you still haven’t seen an actual block, and you’re going crazy wondering what they look like. Now that the setup’s out of the way, we can show you...
Your first block

Are you ready? Here it comes: your first glimpse of a Ruby block.

A block must always follow a method call.

Start of the block

my_method do
  puts "We're in the block!"
end

Block body

End of the block

There it is! Like we said, a block is just a chunk of code that you pass to a method. We invoke my_method, which we just defined, and then place a block immediately following it. The method will receive the block in its my_block parameter.

- The start of the block is marked with the keyword do, and the end is marked by the keyword end.
- The block body consists of one or more lines of Ruby code between do and end. You can place any code you like here.
- When the block is called from the method, the code in the block body will be executed.
- After the block runs, control returns to the method that invoked it.

So we can call my_method and pass it the above block. The method will receive the block as a parameter, my_block, so we can refer to the block inside the method.

...and here’s the output we’d see:

We're in the method, about to invoke your block!
We're in the block!
We're back in the method!
Flow of control between a method and block

We declared a method named `my_method`, called it with a block, and got this output:

```ruby
my_method do
  puts "We're in the block!"
end
```

Let’s break down what happened in the method and block, step by step.

1. The first `puts` statement in `my_method`’s body runs.

The method:
```ruby
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
```

The block:
```ruby
do
  puts "We're in the block!"
end
```

2. The `my_block.call` expression runs, and control is passed to the block. The `puts` expression in the block’s body runs.

```ruby
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
do
  puts "We're in the block!"
end
```

3. When the statements within the block body have all run, control returns to the method. The second call to `puts` within `my_method`’s body runs, and then the method returns.

```ruby
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
do
  puts "We're in the block!"
end
  puts "We're in the block!"
end
def my_method(&my_block)
  puts "We're in the method, about to invoke your block!"
  my_block.call
  puts "We're back in the method!"
end
do
  puts "We're in the block!"
end
  puts "We're back in the method!"
end
```

4. Blocks
Calling the same method with different blocks

You can pass many different blocks to a single method.

We can pass different blocks to the method we just defined, and do different things:

```ruby
my_method do
  puts "It's a block party!"
end
```

We're in the method, about to invoke your block!
It's a block party!
We're back in the method!

```ruby
my_method do
  puts "Wooooo!"
end
```

We're in the method, about to invoke your block!
Wooooo!
We're back in the method!

The code in the method is always the same, but you can change the code you provide in the block.
# Calling a block multiple times

A method can invoke a block as many times as it wants.

This method is just like our previous one, except that it has *two* `my_block.call` expressions:

```ruby
def twice(&my_block)
  puts "In the method, about to call the block!"
  my_block.call
  puts "Back in the method, about to call the block again!"
  my_block.call
  puts "Back in the method, about to return!"
end

twice do
  puts "Woooo!"
end
```

The method name is appropriate: as you can see from the output, the method does indeed call our block twice!

1. Statements in the method body run until the first `my_block.call` expression is encountered. The block is then run. When it completes, control returns to the method.

2. The method body resumes running. When the second `my_block.call` expression is encountered, the block is run again. When it completes, control returns to the method so that any remaining statements there can run.

Block parameters

We learned back in Chapter 2 that when defining a Ruby method, you can specify that it will accept one or more parameters:

```ruby
def print_parameters(p1, p2)
  puts p1, p2
end
```

You’re probably also aware that you can pass arguments when calling the method that will determine the value of those parameters.

```ruby
print_parameters("one", "two")
```

In a similar vein, a method can pass one or more arguments to a block. Block parameters are similar to method parameters; they’re values that are passed in when the block is run, and that can be accessed within the block body.

Arguments to call get forwarded on to the block:

You can have a block accept one or more parameters from the method by defining them between vertical bar (|) characters at the start of the block:

```ruby
def give(&my_block)
  my_block.call("2 turtle doves", "1 partridge")
end
```

```ruby
give do |present1, present2|
  puts "My method gave to me..."
  puts present1, present2
end
```

So, when we call our method and provide a block, the arguments to call are passed into the block as parameters, which then get printed. When the block completes, control returns to the method, as normal.

you are here ▶
Using the “yield” keyword

So far, we’ve been treating blocks like an argument to our methods. We’ve been declaring an extra method parameter that takes a block as an object, then using the `call` method on that object.

```ruby
def twice(&my_block)
  my_block.call
  my_block.call
end
```

We mentioned that this wasn’t the easiest way to accept blocks, though. Now, let’s learn the less obvious but more concise way: the `yield` keyword.

The `yield` keyword will find and invoke the block a method was called with—there’s no need to declare a parameter to accept the block.

This method is functionally equivalent to the one above:

```ruby
def twice
  yield
  yield
end
```

Just like with `call`, we can also give one or more arguments to `yield`, which will be passed to the block as parameters. Again, these methods are functionally equivalent:

```ruby
def give(&my_block)
  my_block.call("2 turtle doves", "1 partridge")
end

def give
  yield "2 turtle doves", "1 partridge"
end
```

Conventional Wisdom

Declaring a `&block` parameter is useful in a few rare instances (which are beyond the scope of this book). But now that you understand what the `yield` keyword does, you should just use that in most cases. It’s cleaner and easier to read.
Block formats

So far, we’ve been using the do...end format for blocks. Ruby has a second block format, though: “curly brace” style. You’ll see both formats being used “in the wild,” so you should learn to recognize both.

Aside from do and end being replaced with curly braces, the syntax and functionality are identical.

And just as do...end blocks can accept parameters, so can curly-brace blocks:

```ruby
def take_this
  yield "present"
end

take_this do |thing|
  puts "do/end block got #{thing}" end

take_this { |thing| puts "braces block got #{thing}" } # do/end block got present
```

By the way, you’ve probably noticed that all our do...end blocks span multiple lines, but our curly-brace blocks all appear on a single line. This follows another convention that much of the Ruby community has adopted. It’s valid syntax to do it the other way:

```ruby
breaks convention!
take_this { |thing|
  puts "braces: got #{thing}" }

breaks convention (and is really ugly)!
take_this do |thing| puts "do/end: got #{thing}" end
```

But not only is that out of line with the convention, it’s really ugly.
Tonight’s talk: A method and a block talk about how they became associated with each other.

**Method:**

Thanks for coming, Block! I called you here tonight so we could educate people on how blocks and methods work together. I’ve had people ask me exactly what you contribute to the relationship, and I think we can clear those questions up for everyone.

So most parts of a method’s job are pretty clearly defined. My task, for example, is to loop through each item in an array.

Sure! It’s a task lots of developers need done; there’s a lot of demand for my services. But then I encounter a problem: what do I do with each of those array elements? Every developer needs something different! And that’s where blocks come in…

I know another method that does nothing but open and close a file. He’s very good at that part of the task. But he has no clue what to do with the contents of the file…

I handle the general work that’s needed on a wide variety of tasks…

**Block:**

Sure, Method! I’m here to help whenever you call.

Right. Not a very glamorous job, but an important one.

Precisely. Every developer can write their own block that describes exactly what they need done with each element in the array…

…and so he calls on a block, right? And the block prints the file contents, or updates them, or whatever else the developer needs done. It’s a great working relationship!

And I handle the logic that’s specific to an individual task.
Here are three Ruby method definitions, each of which takes a block:

```ruby
def call_block(&block)
  puts 1
  block.call
  puts 3
end

def call_twice
  puts 1
  yield
  yield
  puts 3
end

def pass_parameters_to_block
  puts 1
  yield
  yield 9, 3
  puts 3
end
```

And here are several calls to the above methods. Match each method call to the output it produces.

(We've done the first one for you.)

A. `call_block(&block)
  puts 1
  block.call
  puts 3`

B. `call_block { puts 2 }
  ......
  end`

C. `call_twice { puts 2 }
  ......
  end`

D. `call_twice do
  puts "two"
  ......
  end`

E. `pass_parameters_to_block do |param1, param2|
  puts param1 + param2
  ......
  end`

F. `pass_parameters_to_block do |param1, param2|
  puts param1 / param2
  ......
  end`
Here are three Ruby method definitions, each of which takes a block:

<table>
<thead>
<tr>
<th>Method Definition</th>
<th>Method Call</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def call_block(&amp;block)</code> <code>puts 1</code> <code>block.call</code> <code>puts 3</code> <code>end</code></td>
<td><code>call_block do</code> <code>puts 2</code> <code>end</code></td>
<td>1 2 3</td>
</tr>
<tr>
<td><code>def call_twice</code> <code>puts 1</code> <code>yield</code> <code>yield</code> <code>puts 3</code> <code>end</code></td>
<td><code>call_twice { puts &quot;two&quot; }</code></td>
<td>1 2 3</td>
</tr>
<tr>
<td><code>def pass_parameters_to_block</code> <code>puts 1</code> <code>yield param1, param2</code> <code>puts param1 + param2</code> <code>end</code></td>
<td>`pass_parameters_to_block do</td>
<td>param1, param2</td>
</tr>
<tr>
<td></td>
<td>`pass_parameters_to_block do</td>
<td>param1, param2</td>
</tr>
</tbody>
</table>

And here are several calls to the above methods. Match each method call to the output it produces.
The "each" method

We had a lot to learn in order to get here: how to write a block, how a method calls a block, how a method can pass parameters to a block. And now, it’s finally time to take a good, long look at the method that will let us get rid of that repeated loop code in our total, refund, and show_discounts methods. It’s an instance method that appears on every Array object, and it’s called each.

You’ve seen that a method can yield to a block more than once, with different values each time:

```ruby
def my_method
  yield 1
  yield 2
  yield 3
end

my_method { |param| puts param }
```

The each method uses this feature of Ruby to loop through each of the items in an array, yielding them to a block, one at a time.

```ruby
"a", "b", "c".each { |param| puts param }
```

If we were to write our own method that works like each, it would look very similar to the code we’ve been writing all along:

```ruby
class Array
  def each
    index = 0
    while index < self.length
      yield self[index]
      index += 1
    end
  end
end
```

Remember, “self” refers to the current object—in this case, the current array.

Then move to the next element, just like before.

We loop through each element in the array, just like in our total, refund, and show_discounts methods. The key difference is that instead of putting code to process the current array element in the middle of the loop, we use the yield keyword to pass the element to a block.
The "each" method, step-by-step

We’re using the each method and a block to process each of the items in an array:

```
"a", "b", "c"].each { |param| puts param }
```

Let’s go step-by-step through each of the calls to the block and see what it’s doing.

1. For the first pass through the while loop, index is set to 0, so the first element of the array gets yielded to the block as a parameter. In the block body, the parameter gets printed. Then control returns to the method, index gets incremented, and the while loop continues.

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

2. Now, on the second pass through the while loop, index is set to 1, so the second element in the array will be yielded to the block as a parameter. As before, the block body prints the parameter, control then returns to the method, and the loop continues.

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

3. After the third array element gets yielded to the block for printing and control returns to the method, the while loop ends, because we’ve reached the end of the array. No more loop iterations means no more calls to the block; we’re done!

```
def each
  index = 0
  while index < self.length
    yield self[index]
    index += 1
  end
end
```

That’s it! We’ve found a method that can handle the repeated looping code, and yet allows us to run our own code in the middle of the loop (using a block). Let’s put it to use!
DRYing up our code with “each” and blocks

Our invoicing system requires us to implement these three methods. All three of them have nearly identical code for looping through the contents of an array.

It’s been difficult to get rid of that duplication, though, because all three methods have different code in the middle of that loop.

```ruby
def total(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount += prices[index]
    index += 1
  end
  amount
end

def refund(prices)
  amount = 0
  index = 0
  while index < prices.length
    amount -= prices[index]
    index += 1
  end
  amount
end

def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end
```

This line in the middle differs, though…

Highlighted lines are duplicated among the three methods.

But now we’ve finally mastered the `each` method, which loops over the elements in an array and passes them to a block for processing.

```
["a", "b", "c"].each { |param| puts param }
```

Let’s see if we can use `each` to refactor our three methods and eliminate the duplication.

- Given an array of prices, add them all together and return the total.
- Given an array of prices, subtract each price from the customer’s account balance.
- Given an array of prices, reduce each item’s price by 1/3, and print the savings.
DRYing up our code with “each” and blocks (continued)

First up for refactoring is the total method. Just like the others, it contains code for looping over prices stored in an array. In the middle of that looping code, total adds the current price to a total amount.

The each method looks like it will be perfect for getting rid of the repeated looping code! We can just take the code in the middle that adds to the total, and place it in a block that’s passed to each.

```
index = 0
while index < prices.length
  amount += prices[index]
  index += 1
end

prices.each { |price| amount += price }
```

Let’s redefine our total method to utilize each, then try it out.

```
def total(prices)
  amount = 0
  prices.each do |price|
    amount += price
  end
  amount
end

prices = [3.99, 25.00, 8.99]
puts format("%.2f", total(prices))  # 37.98
```

Perfect! There’s our total amount. The each method worked!
DRYing up our code with “each” and blocks (continued)

For each element in the array, each passes it as a parameter to the block. The code in the block adds the current array element to the amount variable, and then control returns back to each.

```
def each
    index = 0
    while index < self.length
        yield self[index]
        index += 1
    end
end
```

We’ve successfully refactored the total method!

But before we move on to the other two methods, let’s take a closer look at how that amount variable interacts with the block.

Refactored

- Given an array of prices, add them all together and return the total.
- Given an array of prices, subtract each price from the customer’s account balance.
- Given an array of prices, reduce each item’s price by 1/3, and print the savings.
Blocks and variable scope

We should point something out about our new total method. Did you notice that we use the amount variable both inside and outside the block?

As you may remember from Chapter 2, the scope of local variables defined within a method is limited to the body of that method. You can’t access variables that are local to the method from outside the method.

The same is true of blocks, if you define the variable for the first time inside the block.

But, if you define a variable before a block, you can access it inside the block body. You can also continue to access it after the block ends!
Blocks and variable scope (continued)

Since Ruby blocks can access variables declared outside the block body, our `total` method is able to use `each` with a block to update the `amount` variable.

We can call `total` like this:

```ruby
total([3.99, 25.00, 8.99])
```

The `amount` variable is set to 0, and then `each` is called on the array. Each of the values in the array is passed to the block. Each time the block is called, `amount` is updated:

1. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```

   Updated from 0 to 3.99

2. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```

   Updated from 3.99 to 28.99

3. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```

   Updated from 28.99 to 37.98

When the `each` method completes, `amount` is still set to that final value, 37.98. It’s that value that gets returned from the method.
Using “each” with the “refund” method

We’ve revised the total method to get rid of the repeated loop code. We need to do the same with the refund and show_discounts methods, and then we’ll be done!

The process of updating the refund method is very similar to the process we used for total. We simply take the specialized code from the middle of the generic loop code, and move it to a block that’s passed to each.

```ruby
def refund(prices)
    amount = 0
    index = 0
    while index < prices.length
        amount -= prices[index]
        index += 1
    end
    amount
end
```

Much cleaner, and calls to the method still work just the same as before!

```ruby
prices = [3.99, 25.00, 8.99]
puts format("%.2f", refund(prices))

# -37.98
```

Within the call to each and the block, the flow of control looks very similar to what we saw in the total method:

1. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```

2. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```

3. def each
   ```ruby
   index = 0
   while index < self.length
     yield self[index]
     index += 1
   end
   end
   ```
Using “each” with our last method

One more method, and we’re done! Again, with show_discounts, it’s a matter of taking the code out of the middle of the loop and moving it into a block that’s passed to each.

```ruby
def show_discounts(prices)
  index = 0
  while index < prices.length
    amount_off = prices[index] / 3.0
    puts format("Your discount: $%.2f", amount_off)
    index += 1
  end
end
```

Again, as far as users of your method are concerned, no one will notice you’ve changed a thing!

```ruby
prices = [3.99, 25.00, 8.99]
show_discounts(prices)
```

Here’s what the calls to the block look like:

1. `def each
   index = 0
   while index < self.length
     yield self[index]
     amount_off = price / 3.0
     puts format("Your discount: $%.2f", amount_off)
     index += 1
   end
 end

2. `def each
   index = 0
   while index < self.length
     yield self[index]
     amount_off = price / 3.0
     puts format("Your discount: $%.2f", amount_off)
     index += 1
   end
 end

3. `def each
   index = 0
   while index < self.length
     yield self[index]
     amount_off = price / 3.0
     puts format("Your discount: $%.2f", amount_off)
     index += 1
   end
 end

From here…

To here!

Here’s what the calls to the block look like:
We’ve gotten rid of the repetitive loop code!

We’ve done it! We’ve refactored the repetitive loop code out of our methods! We were able to move the portion of the code that differed into blocks, and rely on a method, each, to replace the code that remained the same!
Pool Puzzle

Your job is to take code snippets from the pool and place them into the blank lines in the code. Don’t use the same snippet more than once, and you won’t need to use all the snippets. Your goal is to make code that will run and produce the output shown.

def pig_latin(words)
    original_length = 0
    __________ = 0
    words.____ do ______
        puts "Original word: #{word}"
        __________ += word.length
        letters = word.chars
        first_letter = letters.shift
        new_word = "#{letters.join}#{first_letter}ay"
        puts "Pig Latin word: #{________}"
        __________ += new_word.length
    end
    puts "Total original length: #{________}"
    puts "Total Pig Latin length: #{new_length}"
end

my_words = ["blocks", "totally", "rock"]
pig_latin(______)
def pig_latin(words)

    original_length = 0
    new_length = 0

    words.each do |word|
        puts "Original word: #{word}"
        original_length += word.length
        letters = word.chars
        first_letter = letters.shift
        new_word = "#{letters.join}#{first_letter}ay"
        puts "Pig Latin word: #{new_word}"
        new_length += new_word.length
    end

    puts "Total original length: #{original_length}"
    puts "Total Pig Latin length: #{new_length}"
end

my_words = ["blocks", "totally", "rock"]
pig_latin(my_words)

Output:

<table>
<thead>
<tr>
<th>Original word: blocks</th>
<th>Pig Latin word: locksbay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original word: totally</td>
<td>Pig Latin word: otallytay</td>
</tr>
<tr>
<td>Original word: rock</td>
<td>Pig Latin word: ockray</td>
</tr>
<tr>
<td>Original total length: 17</td>
<td>Total Pig Latin length: 23</td>
</tr>
</tbody>
</table>
Utilities and appliances, blocks and methods

Imagine two electric appliances: a mixer and a drill. They have very different jobs: one is used for baking, the other for carpentry. And yet they have a very similar need: electricity.

Now, imagine a world where, any time you wanted to use an electric mixer or drill, you had to wire your appliance into the power grid yourself. Sounds tedious (and fairly dangerous), right?

That’s why, when your house was built, an electrician came and installed *power outlets* in every room. The outlets provide the same utility (electricity) through the same interface (an electric plug) to very different appliances.

The electrician doesn’t know the details of how your mixer or drill works, and he doesn’t care. He just uses his skills and training to get the current safely from the electric grid to the outlet.

Likewise, the designers of your appliances don’t have to know how to wire a home for electricity. They only need to know how to take power from an outlet and use it to make their devices operate.

You can think of the author of a method that takes a block as being kind of like an electrician. They don’t know how the block works, and they don’t care. They just use their knowledge of a problem (say, looping through an array’s elements) to get the necessary data to the block.

You can think of calling a method with a block as being kind of like plugging an appliance into an outlet. Like the outlet supplying power, the block parameters offer a safe, consistent interface for the method to supply data to your block. Your block doesn’t have to worry about how the data got there, it just has to process the parameters it’s been handed.

```
def wire
  yield "current"
end
```

```
wire { |power| puts "Using #{power} to turn drill bit" }
wire { |power| puts "Using #{power} to spin mixer" }
```

Not every appliance uses electricity, of course; some require other utilities. There are stoves and furnaces that require gas. There are automatic sprinklers and spray nozzles that use water.

Just as there are many kinds of utilities to supply many kinds of appliances, there are many methods in Ruby that supply data to blocks. The `each` method was just the beginning. Blocks, also sometimes known as lambdas, are crucial components of Ruby. They are used in loops, in functions that have to run code at some future time (known as callbacks), and other contexts.
Learn from experts. Find the answers you need.

Sign up for a **10-day free trial** to get **unlimited access** to all of the content on Safari, including Learning Paths, interactive tutorials, and curated playlists that draw from thousands of ebooks and training videos on a wide range of topics, including data, design, DevOps, management, business—and much more.

Start your free trial at: oreilly.com/safari

(No credit card required.)