Ruby Errors

Prerequisites

Ruby Classes

What you'll learn

- Ruby Exceptions
- How to be a hero
- When to give up
- Backtracing

Initialize Error

```
class Table
    attr_accessor :num_legs

    def initialize(num_legs)
        @num_legs = num_legs
    end
end
```

I've been using phrases like "generating errors" or "returning errors" when referring to buggy Ruby code, but I haven't actually shown you how that works yet.

You may be thinking: if I want my method to *return* an error, I should just *return* some kind of error object. Not exactly.

To explain what I mean, let's revisit our friend the Table class.

Suppose, given this class definition, that some dumb/evil user tries to pass you a negative number as the number of legs. What should we do to prevent that?

Initialize Error

```
def initialize(num_legs)
  if num_legs > 0
    @num_legs = num_legs
  else
    nil
  end
end
```

Maybe we should just return `nil` if the `num_legs` argument is invalid. That seemed to work in the past.

Initialize Error

```
table = Table.new(-1)
p table.num_legs
```

That doesn't work. I can still create a table by passing -1 legs. I still get *something* in my `table` variable. It just has a `nil` number of legs. Because, if I do nothing, `nil` is the default value.

This is not what I want. I don't want a table with `nil` legs.

Initialize Error

```
def initialize(num_legs)
  if num_legs > 0
    @num_legs = num_legs
  else
    @num_legs = 4
  end
end
```

I can use a default value if the `num_legs` is negative, but that begs the question.

What should be the default value? 4? 1? What if I don't know the answer to that question?

Initialize Error

```
def initialize(num_legs)
  if num_legs > 0
    @num_legs = num_legs
  else
    Exception.new
  end
end
```

Maybe I Googled and learned that the error class in Ruby is called `Exception`. What if I return one of these `Exception` objects?

Initialize Error

```
table = Table.new(-1)
puts table.num_legs
```

Nope. That *still* doesn't work. I still get a table with a `nil` number of legs.

It turns out that it doesn't matter what `initialize` returns. I'll always get a table object.

Somehow I need to stop `initialize` from returning anything at all.

Initialize Error

```
def initialize(num_legs)
  if num_legs > 0
    @num_legs = num_legs
  else
    raise "Invalid number of legs."
  end
end
```

The answer is a new method called `raise` (from the `Kernel` module, so it's available everywhere). When I use the `raise` method, I can pass it an optional error message string.

Initialize Error

```
table = Table.new(-1)
```

RuntimeError: Invalid number of legs

This gives me what I need. This line of code doesn't execute correctly. I don't have a table object when it's done. The `table` variable does not have a value. Instead, I get see error in the output.

Initialize Error

```
table = Table.new(-1)
puts "Hello world"
```

RuntimeError: Invalid number of legs

In fact, it works *too* well.

The line that comes right after the bad line of code doesn't run. Since creating a table raised an error, all progress through the program halts.

I can put whatever I want after that first line, even total gibberish. Ruby will never get to it.

Initialize Error

```
def initialize(num_legs)
    raise "Halt!"
    if num_legs > 0
        @num_legs = num_legs
    else
        raise "Invalid number of legs"
    end
end
```

Within the method, the same is true for the `raise` method itself. Anything after `raise` is totally ignored.

This code, for example, won't allow me to create a table under *any* circumstances. `Table.new` will *always* raise an error. There's no way around it.

Initialize Error

```
def initialize(num_legs)
    raise "Halt!"
    complete nonsense
    if num_legs > 0
        @num_legs = num_legs
    else
        raise "Invalid number of legs"
    end
end
```

And, as we've seen before, when Ruby ignores code, it really ignores it. I can put complete nonsense after the `raise` and Ruby will never read it. Ruby won't even tell me if a subsequent line has a syntax error.

Error Recovery

```
def add_two(number)
    unless number.respond_to? :+
        raise "Invalid argument"
    end
    number + 2
end
```

So `raise` gives me what I want, but it's a bit harsh. The whole program dies. No further lines of code are evaluated. How do I recover from this error?

Let's revisit the `add_two` method we defined earlier. Instead of being overly defensive and making a heroic effort to support all sorts of arguments, let's just raise an error if we can't find a + method.

This is actually a really common way of handling invalid argument types. Just complain that the argument is invalid and halt the program.

Error Recovery

```
puts add_two({})
```

RuntimeError: Invalid argument

Now passing an invalid type of data to my method raises an error and halts the program.

Error Recovery

```
# some previous code

puts add_two({})

# some more code (the never runs)
```

Raising an error is fatal to my entire program. I'll run some prior lines of code, then I'll run a method that raises an error. Then boom, my program is done. Game over. The entire program halts and exits and I get an error in my output.

How do I handle these errors in code if Ruby doesn't allow me to run any more code after the error happens? What if I want to display something friendlier than a Ruby error message to my users?

Error Recovery

```
begin
  puts add_two({})
end
```

The first step is to quarantine this potentially fatal code. Let's wrap it in a block.

This code is potentially fatal to my entire program. Once we have it isolated, we need a hero to rescue us from this disaster.

Error Rescue

```
begin
  puts add_two({})
rescue
  puts "Sorry!"
end
```

This is our code hero.

Once some potentially fatal code is isolated, we can add a `rescue` block to the end to take special measures if and when something bad happens within our quarantine area.

In this case, the `rescue` block recovers control of the halting program, revives it, and outputs a friendly “Sorry!” message to the user.

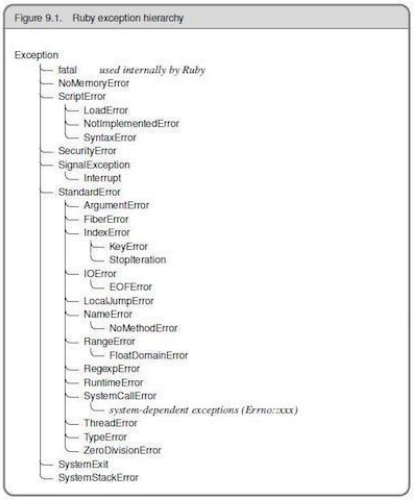
Error Rescue

```
begin
  puts add_two({})
rescue
  puts "Sorry!"
end
puts "And we're back!"
```

Once a block is rescued from an error, Ruby will abort the block, execute the rescue block, and continue reading lines of code.

My program has been rescued from it’s terrible fate.

Error Types



When you use raise with just an error message string, you actually create a `RuntimeError` object. There are a bunch of other types of errors as well.

This is a picture of the object hierarchy for errors. I'm not sure if this is readable, but the point is that there are lots.
[This is also available in the documentation for Ruby Exception. Search for "Ruby errors".]

The `Exception` class is the root of this object hierarchy. Everything inherits from there.

Error Types

```
def add_two(number)
    unless number.respond_to? :+
        raise ArgumentError, "Invalid argument"
    end
    number + 2
end
```

I'm not going to go over all the different `Exception` types. Frankly, I don't see them used very often. Just using "`raise`" is good enough for most people.

But, technically speaking, this particular type of error is, by convention, an `ArgumentError`. `ArgumentError`s are typically used to indicate that there's some problem with the arguments being passed into a method, which is exactly what's going on here. This method has a problem with arguments that don't support the + method.

Error Types

```
def add_two(number)
    if not number.respond_to? :+
        raise ArgumentError, "Invalid argument"
    elsif number == 0
        raise "I just don't like 0"
    end
    number + 2
end
```

Why would you want to raise different error types? Let's say that, just for fun, we want to raise an `ArgumentError` for bad data types, but a regular `RuntimeError` if the argument is 0 (just because we don't like the number 0).

Error Type Rescue

```
begin
  puts add_two(0)
rescue ArgumentError
  puts "Sorry! My bad."
rescue
  puts "What?!"
end
```

When a method returns different kinds of errors, we can define different kinds of error recovery. By putting the `Exception` class name after the rescue, we can define a block that will only respond to those types of errors.

So, for example, we can apologize if there's an argument error and whine if there's any other kind of error.

Error Type Rescue

```
begin
  puts add_two({})
rescue ArgumentError => e
  puts "You: #{e.message}. Me: Sorry!"
rescue
  puts "What?!"
end
```

If we want, we can even examine the error itself. If you add an arrow (called a “hashrocket”) followed by any variable name (typically “e” or “ex” for error or exception), that variable will be assigned to the rescued `Exception`. You can then use that variable in your recovery block.

Error Backtrace

```
begin
  puts add_two(0)
rescue ArgumentError
  puts "Sorry!"
rescue => e
  puts "What?!"
  puts e.backtrace
end
```

Other than the get the message, one of the more useful things you can do with `Exception` objects is figure out where they came from. Specifically, you can figure out what line of code raised the error. That information is stored in an `Array` named "`backtrace`".

Error Backtrace

wyncode.rb:3:in `add_two'
wyncode.rb:11:in `<main>'

This is a sample backtrace.

Each String in the backtrace lists a file name, a line number, and a context. I'm working in a file named wyncode.rb. The error ended up at line 11 in the file, but started at line 3, within the `add_two` method.

Error Backtrace

```
def a
    b
end

def b
    c
end
```

```
def c
    d
end

def d
    raise "Boom!"
end
```

A backtrace is really useful because it can go deep into a program and tell you where an error was first raised.

For example, consider this code here. The method “a” calls the method “b”, which calls “c”, which calls “d”. And “d” raises an error. Most complicated programs looks like this, where one method calls another, which calls another, and so on - sometimes for 10s, even 100s of method calls.

Error Backtrace

```
begin
  a
rescue => e
  puts e.backtrace
end
```

Let’s examine the backtrace of the Exception raised by calling a.

Error Backtrace

```
wyncode.rb:14:in `d'
wyncode.rb:10:in `c'
wyncode.rb:6:in `b'
wyncode.rb:2:in `a'
wyncode.rb:18:in `<main>'
```

Reading from bottom-to-top, I can see that the error ended up in my main workspace in wyncode.rb, inside my code block. Before then it was caused by an error in the “a” method (at line 2), which was itself caused by an error in the “b” method (at line 6), which was caused by an error in the “c” method (at line 10), which, at the top of the backtrace, originated with an error in the “d” method (at line 14). So all the trouble began at line 14 in the “d” method.

Given this stack trace, I can fix the bug at the source, or I can choose to rescue the bug at any other point in the backtrace.

Error Backtrace

```
def b
    begin
        c
    rescue
        nil
    end
end
```

So if I update the “b” method, for example, to rescue any errors in c, like this...

Error Backtrace

```
def b
    c rescue nil
end
```

Which, in typical Ruby fashion, I can write shorthand in 1-line like this...

Error Backtrace

```
begin
  a
rescue => e
  puts e.backtrace
end
```

I can call my “a” method and nothing bad will happen.

Error Backtrace

```
def a
    b
end

def b
    c rescue nil
end
```

```
def c
    d
end

def d
    raise "Boom!"
end
```

Figuring out where it's appropriate to handle an error is an important part of software design. In general, you want to handle the error if, wherever you are, you're actually capable of doing something useful.

Rescuing Errors

```
begin
  1/0
rescue ZeroDivisionError
  puts "I can't divide by zero"
end
```

For example, mathematically it’s impossible to divide a number by 0. So, if someone attempts it, you can decide to warn the user that it’s a bad idea.

Rescuing Errors

```
begin
  exit
rescue SystemExit
  puts "Not so fast!"
end
```

Then there are more serious problems, like SystemExit. SystemExit is raised when a user is specifically asking to your program to stop and go away, such as by typing exit in the Ruby REPL. If you try to rescue this class of Exception somewhere, you're preventing the user from leaving, which is just annoying. There isn't anything useful you can say that will make that user feel good about what you're doing. And it forces the user to use more drastic measures to force your program to go away, which could cause damage. So just let it go.

Rescuing Errors

```
begin
  exit
rescue # assumed StandardError
  puts "Not so fast!"
end
```

By default, rescue will ignore the most serious errors. It will only catch Exceptions in the StandardError branch.

Rescuing Errors

```
rescue Exception
end
```

One final note: never write this line of code. Ever.

Exception is the root of the Exception branch. Every error, from the most mundane to the most serious, will get caught by this block of code. There's no block of code that you can write that is appropriate for this rescue block. These types of errors can include: "your computer is out of disk space", "your computer is on fire", and "it's the apocalypse". There's nothing you can do about it. Just let it go.

TODO:

- don't use errors for flow control,
- throw/catch vs. return vs. raise/rescue

```
catch :ball do
        0.upto(10).each do |i|
                puts i
                10.downto(0).each do |j|
                        puts j
                        0.upto(10).each do |k|
                                puts k
                                throw :ball if k.zero?
                        end
                end
        end
end
```