Mastering

Ruby Exceptions

Starr Horne

Honeybadger.io

Contents

Chapter 1

Introduction

You've seen exceptions. Perhaps you've used them in your code. But how much do you know beyond the basic syntax?

As one of the founders of Honeybadger, an exception monitoring service for Ruby, I've spent a lot more time looking at and thinking about exceptions than your average developer. I've had a chance to study the intricate details of how Ruby's exception system works and how you can use it to your advantage.

The thing that strikes me over and over is how deep the rabbit hole goes. Exceptions seem so cut and dry. Every Ruby developer knows more-or-less how to handle them. But exceptions also have a deeper, more subtle side that gives the developer more control than they would have ever imagined.

I'd like this book to be useful to developers of all experience levels. I've structured it so that it progresses from basic to advanced. We'll start out with a chapter providing the most beginner-friendly introduction to exceptions that I could come up with. From there we'll introduce mid-level concepts like how to create your own exceptions. Finally, we'll end with information on how to extend Ruby's exception system in both sane and insane ways.

If you have any questions or any suggestions for improvement, please don't hesitate to reach out to me via email starr@honeybadger.io or twitter @StarrHorne

Chapter 2

What Are Exceptions?

Exceptions are Ruby's way of dealing with unexpected events.

If you've ever made a typo in your code, causing your program to crash with a message like `SyntaxError` or `NoMethodError` then you've seen exceptions in action.

When you raise an exception in Ruby, the world stops and your program starts to shut down. If nothing stops the process, your program will eventually exit with an error message.

Here's an example. In the code below, we try to divide by zero. This is impossible, so Ruby raises an exception called `ZeroDivisionError`. The program quits and prints an error message.

```
1 / 0
# Program crashes and outputs: "ZeroDivisionError: divided by 0"
```

Crashing programs tend to make our users angry. We normally want to stop this shutdown process and react to the error intelligently.

This is called "rescuing," "handling," or "catching" an exception. They all mean the same thing. This is how you do it in Ruby:

```
begin
  # Any exceptions in here...
```

```
  1/0
rescue
  # ...will cause this code to run
  puts "Got an exception, but I'm responding intelligently!"
  do_something_intelligent()
end


# This program does not crash.
# Outputs: "Got an exception, but I'm responding intelligently!"
```

The exception still happens, but it doesn't cause the program to crash because it was "rescued." Instead of exiting, Ruby runs the code in the rescue block, which prints out a message.

This is nice, but it has one big limitation. It tells us "something went wrong," without letting us know *what* went wrong.

All of the information about what went wrong is going to be contained in an exception object.

Exception Objects

Exception objects are normal Ruby objects. They hold all of the data about "what happened" for the exception you just rescued.

To get the exception object, use the rescue syntax below.

```
# Rescues all errors, an puts the exception object in `e`
rescue => e


# Rescues only ZeroDivisionError and puts the exception object in `e`
rescue ZeroDivisionError => e
```

In the second example above, `ZeroDivisionError` is the class of the object in `e`. All of the "types" of exceptions we've talked about are really just class names.

Exception objects also hold useful debug data. Let's take a look at the exception object for our `ZeroDivisionError`.

```ruby
begin
  # Any exceptions in here...
  1/0
rescue ZeroDivisionError => e
  puts "Exception Class: #{ e.class.name }"
  puts "Exception Message: #{ e.message }"
  puts "Exception Backtrace: #{ e.backtrace }"
end

# Outputs:
# Exception Class: ZeroDivisionError
# Exception Message: divided by 0
# Exception Backtrace: ...backtrace as an array...
```

Most exception objects will provide you with at least three pieces of data:

1. The type of exception, given by the class name.
2. An exception message
3. A backtrace

Raising Your Own Exceptions

So far we've only talked about rescuing exceptions. You can also trigger your own exceptions. This process is called "raising" and you do it by calling the `raise` method.

When you raise your own exceptions, you get to pick which type of exception to use. You also get to set the message.

Here's an example:

```ruby
begin
  # raises an ArgumentError with the message "you messed up!"
  raise ArgumentError.new("You messed up!")
rescue ArgumentError => e
  puts e.message
end
```

```
# Outputs: You messed up!
```

As you can see, we're creating a new error object (an `ArgumentError`) with a custom message ("You messed up!") and passing it to the `raise` method.

This being Ruby, `raise` can be called in several ways:

```
# This is my favorite because it's so explicit
raise RuntimeError.new("You messed up!")

# ...produces the same result
raise RuntimeError, "You messed up!"

# ...produces the same result. But you can only raise
# RuntimeErrors this way
raise "You messed up!"
```

Making Custom Exceptions

Ruby's built-in exceptions are great, but they don't cover every possible use case.

What if you're building a user system and want to raise an exception when the user tries to access an off-limits part of the site? None of Ruby's standard exceptions fit, so your best bet is to create a new kind of exception.

To make a custom exception, just create a new class that inherits from `StandardError`.

```
class PermissionDeniedError < StandardError
end

raise PermissionDeniedError.new()
```

Because `PermissionDeniedError` is a class, you can add attributes and methods to it like you would with any other class. Let's add an attribute called "action":

```ruby
class PermissionDeniedError < StandardError

  attr_reader :action

  def initialize(message, action)
    # Call the parent's constructor to set the message
    super(message)

    # Store the action in an instance variable
    @action = action
  end

end

# Then, when the user tries to delete something they don't
# have permission to delete, you might do something like this:
raise PermissionDeniedError.new("Permission Denied", :delete)
```

The Class Hierarchy

We just made a custom exception by subclassing `StandardError`, which itself subclasses `Exception`.

In fact, if you look at the class hierarchy of any exception in Ruby, you'll find it eventually leads back to `Exception`. Here, I'll prove it to you. These are most of Ruby's built-in exceptions, displayed hierarchically:

```
Exception
 NoMemoryError
 ScriptError
   LoadError
   NotImplementedError
   SyntaxError
 SignalException
   Interrupt
 StandardError
   ArgumentError
```

```
    IOError
      EOFError
    IndexError
    LocalJumpError
    NameError
      NoMethodError
    RangeError
      FloatDomainError
    RegexpError
    RuntimeError
    SecurityError
    SystemCallError
    SystemStackError
    ThreadError
    TypeError
    ZeroDivisionError
  SystemExit
```

You don't have to memorize all of these, of course. I'm showing them to you because this idea of hierarchy is very important. Here's why:

Rescuing errors of a specific class also rescues errors of its child classes.

For example, when you `rescue StandardError`, you not only rescue exceptions with class `StandardError` but those of its children as well. If you look at the chart, you'll see this includes `ArgumentError`, `IOError`, and many more.

Because all exceptions inherit from `Exception` you can rescue via `rescue Exception`. This, however, would be a very bad idea.

Rescuing All Exceptions (The Wrong Way)

Here is some code you never want to submit to a code review:

```ruby
# Don't do this
begin
```

```
  do_something()
rescue Exception => e
  ...
end
```

The code above will rescue every exception. **Don't do it!** It'll break your program in weird ways.

That's because Ruby uses exceptions for things other than errors. It also uses them to handle messages from the operating system called "Signals." If you've ever pressed "ctrl-c" to exit a program, you've used a signal. By suppressing all exceptions, you also suppress those signals.

There are also some kinds of exceptions — like syntax errors — that really should cause your program to crash. If you suppress them, you'll never know when you make typos or other mistakes.

Rescuing All Errors (The Right Way)

Go back and look at the class hierarchy chart and you'll see that all of the errors you'd want to rescue are children of `StandardError`.

That means that if you want to rescue "all errors" you should rescue `StandardError`.

```
begin
  do_something()
rescue StandardError => e
  # Only your app's exceptions are swallowed. Things like SyntaxError are left a
end
```

In fact, if you don't specify an exception class, Ruby assumes you mean `StandardError`.

```
begin
  do_something()
rescue => e
  # This is the same as rescuing StandardError
end
```

Rescuing Specific Errors (The Best Way)

Now that you know how to rescue all errors, you should know that it's usually a bad idea, a code smell, considered harmful, etc.

Blanked rescues are usually a sign that you were too lazy to figure out which specific exceptions needed to be rescued. And they will almost always come back to haunt you.

So take the time and do it right. Rescue specific exceptions.

```ruby
begin
  do_something()
rescue Errno::ETIMEDOUT => e
  // This will only rescue Errno::ETIMEDOUT exceptions
end
```

You can even rescue multiple kinds of exceptions in the same rescue block, so no excuses. :)

```ruby
begin
  do_something()
rescue Errno::ETIMEDOUT, Errno::ECONNREFUSED => e
  // This will rescue both Errno::ETIMEDOUT and Errno::ECONNREFUSED exceptions
end
```

Chapter 3

Advanced Rescue & Raise

Much of the hidden power of Ruby's exception system is contained in the humble `rescue` and `raise` syntax. In this chapter I'm going to show you how to harness that power. If the beginning of the chapter is old-news to you, stick with it! By the end, we'll be covering new ground.

Full Rescue Syntax

Simple begin-rescue-end clauses work well for most cases. But occasionally you have a situation that requires a more sophisticated approach. Below is an example of Ruby's full rescue syntax.

```ruby
begin
  ...
rescue TooHotError => too_hot
  # This code is run if a TooHotError occurs
rescue TooColdError => too_cold
  # This code is run if a TooColdError occurs
else
  # This code is run if no exception occurred at all
ensure
  # This code is always run, regardless of whether an exception occurred
end
```

As you can see we've introduced a few new keywords:

- `else` - is executed when no exceptions occur at all.

- `ensure` - is *always* executed, even if exceptions did occur. This is really useful for actions like closing files when something goes wrong.
- `rescue` - you know what rescue, is, but I just wanted to point out that we're using *multiple* rescues to provide different behavior for different exceptions.

I find that the full rescue syntax is most useful when working with disk or network IO. For example, at Honeybadger we do uptime monitoring — requesting your web page every few minutes and letting you know when it's down. Our logic looks something like this:

```ruby
begin
  request_web_page
rescue TimeoutError
  retry_request
rescue
  send_alert
else
  record_success
ensure
  close_network_connection
else
```

Here, we respond to timeout exceptions by retrying. All other exceptions trigger alerts. If no error occurred, we save the success in the database. Finally, regardless of what else occurred, we always close the network connection.

Method Rescue Syntax

If you find yourself wrapping an entire method in a rescue clause, Ruby provides an alternate syntax that is a bit prettier:

```ruby
def my_method
  ...
rescue
  ...
else
  ...
ensure
  ...
end
```

This is particularly useful if you want to return a "fallback" value when an exception occurs. Here's an example:

```ruby
def data
  JSON.parse(@input)
rescue JSON::JSONError
  {}
end
```

The `retry` Keyword

The `retry` keyword allows you to re-run everything between `begin` and `rescue`. This can be useful when you're working with things like flaky external APIs. The following example will try an api request three times before giving up.

```ruby
counter = 0
begin
  counter += 1
  make_api_request
rescue
  retry if counter <= 3
end
```

Ruby doesn't have a built-in mechanism to limit the number of retries. If you don't implement one yourself — like the `counter` variable in the above example — the result will be an infinite loop.

Reraising Exceptions

If you call `raise` with no arguments, while inside of a rescue block, Ruby will re-raise the original rescued exception.

```ruby
begin
  ...
rescue => e
  raise if e.message == "Fubar"
end
```

To the outside world a re-raised exception is indistinguishable from the original.

Changing Exceptions

It's common to rescue one exception and raise another. For example, `ActionView` rescues all exceptions that occur in your ERB templates and re-raises them as `ActionView::TemplateError` exceptions.

```ruby
begin
  render_template
rescue
  raise ActionView::TemplateError
end
```

In older versions of Ruby (pre 2.1) the original exception was discarded when using this technique. Newer versions of Ruby make the original available to you via a nested exception system which we will cover in detail in Chapter 4.

Exception Matchers

In this book, you've seen several examples where we rescue a specific type of exception. Here's another one:

```ruby
begin
  ...
rescue StandardError
end
```

This is good enough 99.99% of the time. But every so often you may find yourself needing to rescue exceptions based on something other than "type." Exception matchers give you this flexibility. When you define an exception matcher class, you can decide at runtime which exceptions should be rescued.

The following example shows you how to rescue all exceptions where the message begins with the string "FOOBAR".

```ruby
class FoobarMatcher
  def self.===(exception)
    # rescue all exceptions with messages starting with FOOBAR
    exception.message =~ /^FOOBAR/
  end
end

begin
  raise EOFError, "FOOBAR: there was an eof!"
rescue FoobarMatcher
  puts "rescued!"
end
```

The Mechanism

To understand how exception matcher classes work, you first need to understand how `rescue` decides which exceptions to rescue.

In the code below we've told Ruby to rescue a `RuntimeError`. But how does ruby know that a given exception is a `RuntimeError`?

```ruby
begin
  ...
rescue RuntimeError
end
```

You might think that Ruby simply checks the exception's class via `is_a?`:

```ruby
exception.is_a?(RuntimeError)
```

But the reality is a little more interesting. Ruby uses the `===` operator. Ruby's triple-equals operator doesn't have anything to do with testing equality. Instead, `a === b` answers the question "is b inherently part of a?".

```ruby
(1..100) === 3        # True
String === "hi"       # True
/abcd/ === "abcdefg"  # True
```

All classes come with a `===` method, which returns true if an object is an instance of said class.

```ruby
def self.===(o)
  self.is_a?(o)
end
```

When we rescue `RuntimeError` ruby tests the exception object like so:

```ruby
RuntimeError === exception
```

Because `===` is a normal method, we can implement our own version of it. This means we can easily create custom "matchers" for our rescue clauses.

Let's write a matcher that matches every exception:

```ruby
class AnythingMatcher
  def self.===(exception)
    true
  end
end

begin
  ...
rescue AnythingMatcher
end
```

By using === the Ruby core team has made it easy and safe to override the
default behavior. If they had used is_a?, creating exception matchers would
be much more difficult and dangerous.

Syntactic Sugar

This being Ruby, it's possible to create a much prettier way to dynamically
catch exceptions. Instead of manually creating matcher classes, we can write
a method that does it for us:

```ruby
def exceptions_matching(&block)
  Class.new do
    def self.===(other)
      @block.call(other)
    end
  end.tap do |c|
    c.instance_variable_set(:@block, block)
  end
end


begin
  raise "FOOBAR: We're all doomed!"
rescue exceptions_matching { |e| e.message =~ /^FOOBAR/ }
  puts "rescued!"
end
```

Advanced Raise

In Chapter 1 we covered a few of the ways you can call raise. For example
each of the following lines will raise a RuntimeError.

```ruby
raise
raise "hello"
raise RuntimeError, "hello"
raise RuntimeError.new("hello")
```

If you look at the Ruby documentation for the `raise` method, you'll see something weird. The final variant — my personal favorite — isn't explicitly mentioned.

```ruby
raise RuntimeError.new("hello")
```

To understand why, let's look at a few sentences of the documentation:

> With a single String argument, raises a RuntimeError with the string as a message. Otherwise, the first parameter should be the name of an Exception class (or an object that returns an Exception object when sent an exception message).

The important bit is at the very end: "or an object that returns an Exception object when sent an exception message."

This means that if `raise` doesn't know what to do with the argument you give it, it'll try to call the `exception` method of that object. If it returns an exception object, then that's what will be raised.

And all exception objects have a method `exception` which by default returns `self`.

```ruby
e = Exception.new
e.eql? e.exception # True
```

Raising Non-Exceptions

If we provide an `exception` method, any object can be raised as an exception.

Imagine you have a class called `HTMLSafeString` which contains a string of text. You might want to make it possible to pass one of these "safe" strings to `raise` just like a normal string. To do this we simply add an `exception` method that creates a new `RuntimeError`.

```ruby
class HTMLSafeString
  def initialize(value)
```

```ruby
    @value = value
  end

  def exception
    RuntimeError.new(@value)
  end
end

raise HTMLSafeString.new("helloworld")
```

Chapter 4

Advanced Exception Objects

Exception objects hold all of the information about "what happened" during an exception.

We've seen how you can retrieve the exception object when rescuing:

```ruby
begin
  ...
rescue StandardError => e
  puts "Got the exception object: #{ e }"
end
```

Now let's dig in and see just what information these objects contain.

What Are Exception Objects?

Exception objects are instances of exception classes. It's that simple.

In the example below, we create a new exception object and raise it. If we catch it, we can see that it's the same object that we raised.

```ruby
my_exception = RuntimeError.new

begin
```

```
  raise my_exception
rescue => e
  puts(e == my_exception) # prints "true"
end
```

Not all classes can be used as exceptions. Only those that inherit from `Exception` can.

Remember our chart of the exception class hierarchy? `Exception` is right at the top:

```
Exception
 NoMemoryError
 ScriptError
   LoadError
   NotImplementedError
   SyntaxError
 SignalException
   Interrupt
 StandardError
   ArgumentError
   IOError
     EOFError

 ..etc
```

The Class

The first — and probably most important — piece of information about an exception object is it's class name. Classes like `ZeroDivisionError` and `ActiveRecord::RecordNotFound` tell you exactly what went wrong.

Because your code is unique, it can be useful to define your own exception classes to describe an exceptional condition. To do this, simply create a class that inherits from another kind of Exception.

The most common approach is to inherit from `StandardError`. As long as your exception is a kind of error, it's a good idea to follow this practice.

```ruby
class MyError < StandardError
end

raise MyError
```

Namespacing

It's a good idea to namespace your new exception classes by placing them in a module. For example, we could put our `MyError` class in a module named `MyLibrary`:

```ruby
class MyLibrary::MyError < StandardError
end
```

I haven't used namespacing in the code examples in this book to save space and to avoid confusing beginners who might think that the module syntax is a requirement.

Inheritance

You've seen how rescuing `StandardError` not only rescues exception of that class, but of all its child classes as well.

You can take advantage of this behavior to create your own easily-rescuable exception groups.

```ruby
class NetworkError < StandardError
end

class TimeoutError < NetworkError
end

class UnreachableError < NetworkError
end

begin
```
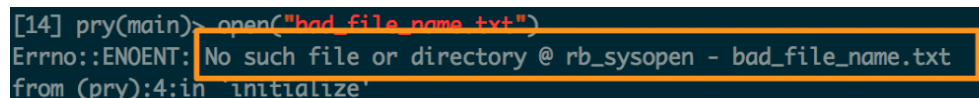
```
  network_stuff
rescue NetworkError => e
  # this also rescues TimeoutError and UnreachableError
  # because they inherit from NetworkError
end
```

The Message

When your program crashes due to an exception, that exception's message is printed out right next to the class name.

Figure 4.1: Example of an exception's message attribute

To read an exception's message, just use the `message` method:

```
e.message
```

Most exception classes allow you to set the message via the `intialize` method:

```
raise RuntimeError.new("This is the message")

# This essentially does the same thing:
raise RuntimeError, "This is the message"
```

The exception's message is a read-only attribute. It can't be changed without resorting to trickery.

The Backtrace

You know what a backtrace is, right? When your program crashes it's the thing that tells you what line of code caused the trouble.

The backtrace is stored in the exception object. The `raise` method generates it via `Kernel#caller` and stores it in the exception via `Exception#set_backtrace`.

The trace itself is just an array of strings. We can examine it like so:

```ruby
begin
  raise "FOO"
rescue => e
  e.backtrace[0..3].each do |line|
    puts line
  end
end
```

This prints out the first three lines of the backtrace. If I run the code in IRB it looks like this:

```
/irb/workspace.rb:87:in `eval'
/irb/workspace.rb:87:in `evaluate'
/irb/context.rb:380:in `evaluate'
```

You can also set the backtrace, which is useful if you ever need to "convert" one exception into another. Or perhaps you're building a template engine and want to set a backtrace which points to a line in the template instead of a line of Ruby code:

```ruby
e2 = ArgumentError.new
e2.set_backtrace(e1.backtrace)
```

Your Own Data

Adding "custom" attributes to your exception classes is as easy as adding them to any other class.

In the following example, we create a new kind of exception called `MyError`. It has an extra attribute called `thing`.

```ruby
class MyError < StandardError
  attr_reader :thing
  def initialize(msg="My default message", thing="apple")
    @thing = thing
    super(msg)
  end
end


begin
  raise MyError.new("my message", "my thing")
rescue => e
  puts e.thing # "my thing"
end
```

Causes (Nested Exceptions)

Beginning in Ruby 2.1 when an exception is rescued and another is raised, the original is available via the `cause` method.

```ruby
def fail_and_reraise
  raise NoMethodError
rescue
  raise RuntimeError
end


begin
  fail_and_reraise
rescue => e
  puts "#{ e } caused by #{ e.cause }"
end


# Prints "RuntimeError caused by NoMethodError"
```

Above, we raised a `NoMethodError` then rescued it and raised a `RuntimeError`. When we rescue the `RuntimeError` we can call `cause` to get the `NoMethodError`.

The causes mechanism only works when you raise the second exception from inside the rescue block. The following code won't work:

```ruby
# Neither of these exceptions will have causes
begin
  raise NoMethodError
rescue
end


# Since we're outside of the rescue block, there's no cause
raise RuntimeError
```

Nested Exception Objects

The `cause` method returns an exception object. That means that you can access any metadata that was part of the original exception. You even get the original backtrace.

In order to demonstrate this, I'll create a new kind of exception called `EatingError` that contains a custom attribute named `food`.

```ruby
class EatingError < StandardError
  attr_reader :food
  def initialize(food)
    @food = food
  end
end
```

Now we'll rescue our `EatingError` and raise `RuntimeError` in its place:

```ruby
def fail_and_reraise
  raise EatingError.new("soup")
rescue
  raise RuntimeError
end
```

When we rescue the `RuntimeError` we can access the original `EatingError` via `e.cause`. From there, we can fetch the value of the `food` attribute ("soup") and the first line of the backtrace:

```ruby
begin
  fail_and_reraise
rescue => e
  puts "#{ e } caused by #{ e.cause } while eating #{ e.cause.food }"
  puts e.cause.backtrace.first
end

# Prints:
# RuntimeError caused by EatingError while eating soup
# eating.rb:9:in `fail_and_reraise'
```

Multiple Levels of Nesting

An exception can have an arbitrary number of causes. The example below shows three exceptions being raised. We catch the third one and use `e.cause.cause` to retrieve the first.

```ruby
begin
  begin
    begin
      raise "First"
    rescue
      raise "Second"
    end
  rescue
    raise "Third"
  end
rescue => e
  puts e.cause.cause
  # First
end
```

Chapter 5

Extending the Exception System

So far the topics we've covered have been interesting and perhaps obscure. But they haven't been *dangerous*.

Ruby is a supremely flexible language. It lets you modify the behavior of core systems, and exceptions are no different. We can use techniques like monkey-patching to open up whole new realms of possibility when it comes to exception handling.

Extending Ruby's exception system is an interesting exercise. It can even be useful. But most of the time it's a bad idea. Still, we're all adults here so I'll leave it to you to decide what's best for your particular use-case.

Retrying Failed Exceptions

We covered the `retry` keyword in Chapter 2. If you use `retry` in your rescue block it causes the section of code that was rescued to be run again. Let's look at an example.

```ruby
begin
  retries ||= 0
  puts "try ##{ retries }"
```

```ruby
  raise "the roof"
rescue
  retry if (retries += 1) < 3
end

# ... outputs the following:
# try #0
# try #1
# try #2
```

The Problem With `retry`

While `retry` is great it does have some limitations. The main one being that the entire begin block is re-run.

For example, imagine that you're using a gem that lets you post status updates to Twitter, Facebook, and lots of other sites by using a single method call. It might look something like this:

```ruby
SocialMedia.post_to_all("Zomg! I just ate the biggest hamburger")

# ...posts to Twitter API
# ...posts to Facebook API
# ...etc
```

If one of the APIs fails to respond, the gem raises a `SocialMedia::TimeoutError` and aborts. If we were to catch this exception and retry, we'd wind up with duplicate posts because the retry would start over from the beginning.

```ruby
begin
  SocialMedia.post_to_all("Zomg! I just ate the biggest hamburger")
rescue SocialMedia::TimeoutError
  retry
end

# ...posts to Twitter API
# facebook error
```

```
# ...posts to Twitter API
# facebook error
# ...posts to Twitter API
# and so on
```

Wouldn't it be nice if we were able to make the gem only retry the failed requests? Fortunately for us, Ruby allows us to do exactly that.

Continuations to the Rescue

Continuations tend to scare people. They're not used very frequently and they look a little odd. But once you understand the basics they're really quite simple.

A continuation lets you jump to a location in your code. It's kind of like a `goto` statement in BASIC.

Let's use continuations to implement a simple loop:

```ruby
require "continuation"

# Initialize our counter
counter = 0

# Define the location to jump to
continuation = callcc { |c| c }

# Increment the counter and print it
puts(counter += 1)

# Jump to the location above
continuation.call(continuation) if counter < 3
```

When you run this, it produces the following output:

```
1
2
3
```

You may have noticed a few things:

- Continuations require a lot of ugly boilerplate.
- We use the callcc method to create a Continuation object. There's no clean OO syntax for this.
- The first time the `continuation` variable is assigned, it is set to the return value of `callcc` 's block. That's why the block has to be there.
- Each time we jump back to the saved location, the `continuation` variable is assigned whatever argument we pass the `call` method. We don't want it to change, so we do `continuation.call(continuation)`.

Reimagining `retry`

We're going to use continuations to add an `skip` method to all exceptions. The example below shows how it should work. Whenever I rescue an exception I should be able to call `skip`, which will cause the code that raised the exception to act like it never happened.

```ruby
begin
  raise "the roof"
  puts "The exception was ignored"
rescue => e
  e.skip
end

# ...outputs "The exception was ignored"
```

To do this I'm going to have to commit a few sins. `Exception` is just a class. That means I can monkey-patch it to add a `skip` method.

```ruby
class Exception
  attr_accessor :continuation
  def skip
    continuation.call
  end
end
```

Now we need to set the `continuation` attribute for every exception.

The code below is taken almost verbatim from Advi Grimm's excellent slide deck Things You Didn't know about Exceptions. I just couldn't think of a better way to implement it than this:

```ruby
require 'continuation'
module StoreContinuationOnRaise
  def raise(*args)
    callcc do |continuation|
      begin
        super
      rescue Exception => e
        e.continuation = continuation
        super(e)
      end
    end
  end
end

class Object
  include StoreContinuationOnRaise
end
```

Now I can call the `skip` method for any exception and it will be like the exception never happened.

Logging Local Variables on Raise

If you've ever wished that your exceptions contained a more complete representation of program state, you might be interested in this technique for logging local variables at the time an exception was raised.

This technique is not suitable for production. It imposes a large performance penalty on your entire app, even when no exceptions occur. Moreover, it relies on the `binding_of_caller` gem which is not actively maintained.

Introducing `binding_of_caller`

At any given moment, your program has a certain "stack." This is simply the list of currently "in-progress" methods.

In Ruby, you can examine the current stack via the `Kernel#caller` method. Here's an example:

```ruby
def a
  b()
end

def b
  c()
end

def c
  puts caller.inspect
end

a()

# Outputs:
# ["caller.rb:11:in `b'", "caller.rb:4:in `a'", "caller.rb:20:in `<main>'"]
```

A binding is a snapshot of the current execution context. In the example below, I capture the binding of a method, then use it to access the method's local variables.

```ruby
def get_binding
  a = "marco"
  b = "polo"
  return binding
end

my_binding = get_binding

puts my_binding.local_variable_get(:a) # "marco"
puts my_binding.local_variable_get(:b) # "polo"
```

The binding_of_caller gem combines these two concepts. It gives you access to the binding for any level of the current execution stack. Once you have the binding, it's possible to access local variables, instance variables and more.

In the following example, we use the binding_of_caller gem to access local variables in one method while inside another method.

```ruby
require "rubygems"
require "binding_of_caller"

def a
  fruit = "orange"
  b()
end

def b
  fruit = "apple"
  c()
end

def c
  fruit = "pear"

  # Get the binding "two levels up" and ask it for its local variable "fruit"
  puts binding.of_caller(2).local_variable_get(:fruit)
end

a() # prints "orange"
```

This is really cool. But it's also disturbing. It violates everything we've learned about separation of concerns. It's going to get worse before it gets better. Let's suppress that feeling for a bit and press on.

Replacing `raise`

One often-overlooked fact about `raise` is that it's simply a method. That means that we can replace it.

In the example below we create a new `raise` method that uses `binding_of_caller` to print out the local variables of whatever method called `raise`.

```ruby
require "rubygems"
require "binding_of_caller"

module LogLocalsOnRaise
  def raise(*args)
    b = binding.of_caller(1)
    b.eval("local_variables").each do |k|
      puts "Local variable #{ k }: #{ b.local_variable_get(k) }"
    end
    super
  end
end

class Object
  include LogLocalsOnRaise
end

def buggy
  s = "hello world"
  raise RuntimeError
end

buggy()
```

Here's what it looks like in action:

Figure 5.1: Example exception logs

Logging All Exceptions With TracePoint

TracePoint is a powerful introspection tool that has been part of Ruby since version 2.0.
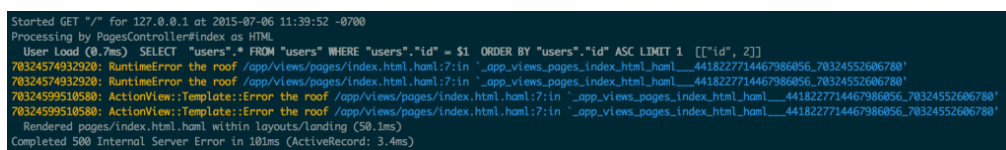
It allows you to define callbacks for a wide variety of runtime events. For example, you can be notified whenever a class is defined, whenever a method is called or whenever an exception is raised. Check out the TracePoint documentation for even more events.

Let's start by adding a TracePoint that is called whenever an exception is raised and writes a summary of it to the log.

```ruby
tracepoint = TracePoint.new(:raise) do |tp|
  # tp.raised_exeption contains the actual exception
  # object that was raised!
  logger.debug tp.raised_exception.object_id.to_s +
    ": " + tp.raised_exception.class +
    " " + tp.raised_exception.message
end

tracepoint.enable do
  # Your code goes here.
end
```

When I run this, every exception that occurs within the `enable` block is logged. It looks like this:

Figure 5.2: Logging every use of `raise` in the rendering process

Chapter 6

Third Party Tools

A number of the tools you may use integrate directly with Ruby's exception system. Let's take a look at a few of the more useful examples:

PRY

One of the nicest things about Ruby is IRB. Pry makes IRB even better. It includes several useful features for dealing with exceptions:

Full Backtraces

When exceptions happen in Pry (or IRB for that matter) you're presented with a shortened version of the backtrace. This is usually good enough, but not always.

In pry you can see the full backtrace of the most recent exception by using the `wtf -v` command. If you leave off the -v flag, you get the abbreviated backtrace.

Exception Objects

Exceptions often have interesting data attached to them. When an exception happens in IRB you can only see the class name and error message. But with

Figure 6.1: Pry's wtf -v command

Pry you have access to the full exception object.

To get the most recently raised exception, use the `_ex_` variable. Unlike Ruby's built-in `$!` variable, you don't have to be inside of a rescue block to use it.

Figure 6.2: Retrieving exception objects in PRY

Exception Formatting

If PRY's default exception formatting doesn't work for you, it can be easily customized.

To modify exception formatting, open your `~/.pryrc` and redefine the default exception handler:

```ruby
# This code was mostly taken from the default exception handler.
# You can see it here: https://github.com/pry/pry/blob/master/lib/pry.rb

Pry.config.exception_handler = proc do |output, exception, _|
  if UserError === exception && SyntaxError === exception
      output.puts "SyntaxError: #{exception.message.sub(/.*syntax error, */m, '')]
  else
    output.puts "#{exception.class}: #{exception.message}"
    output.puts "from #{exception.backtrace}"
  end
end
```

Rack Middleware

Ruby web applications, frameworks and monitoring tools often do interesting things with exceptions. For example Rails displays useful error pages when exceptions occur — instead of simply crashing. The better-errors gem provides an alternative error page. They can do this by taking advantage of a little-known feature of Rack middleware.

If you're not familiar with rack middleware, the concept is simple. It allows you to intercept HTTP requests before they get to your app, and to intercept the app's output before it goes back to the user.

Here's a simple middleware that doesn't do anything interesting.

```ruby
class MyMiddleware
  def initialize(app)
    @app = app
  end
```
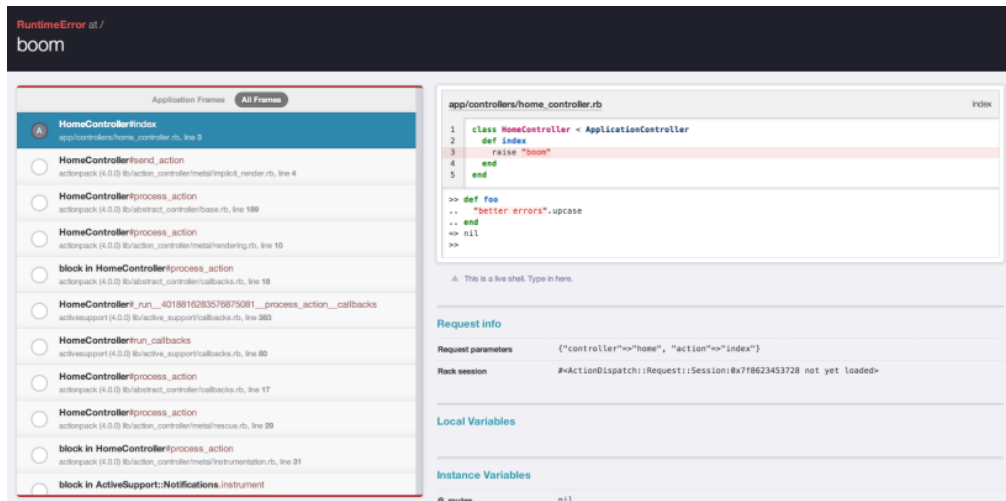
Figure 6.3: Better Errors

```ruby
  def call(env)
    @app.call(env)
  end
end
```

Rack isn't magical. It's just Ruby. When an exception occurs it travels up the middleware method chain until it's caught. That means we can add a middleware specifically to catch exceptions:
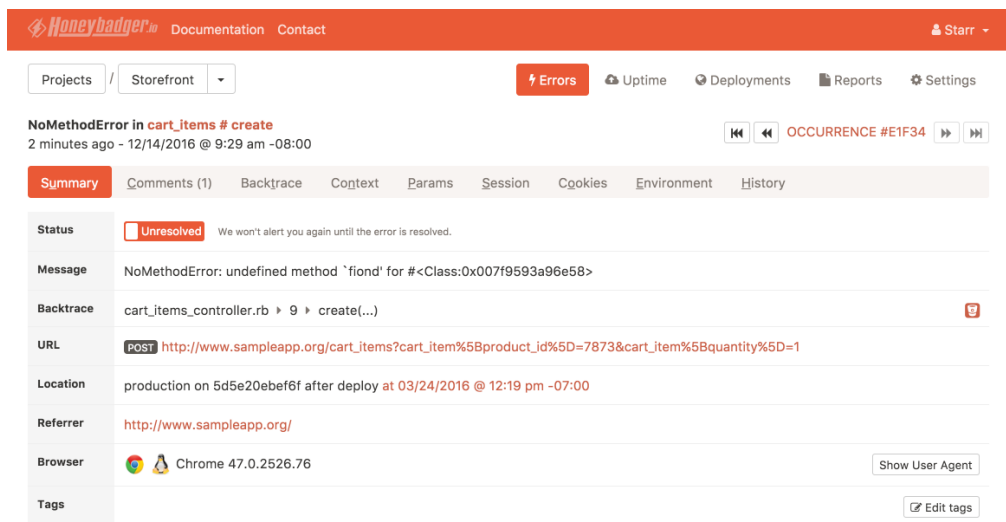
```ruby
def call(env)
  @app.call(env)
rescue StandardError => exception
  # We could render a custom error page here,
  # or log the error to a third party,
  # or send an email to the ops team
end
```

Exception Monitoring with Honeybadger

While Rails' custom exception pages are nice, they're not useful in production. It would be confusing to your users — not to mention a security risk — to display debug data for production errors.

That's why you need a tool like Honeybadger.

Honeybadger watches your web app and reports any exceptions that occur to its servers. You'll be notified that there's a problem and be given all the details of the exception that you need to fix it.

Figure 6.4: Honeybadger Exception Details

Using Honeybadger you can see the error that occurred, the backtrace, and any causes associated with the error. Once you add the gem to your app, you'll never need to worry about missing an error in your production app again. Give it a try at www.honeybadger.io and let us know what you think.