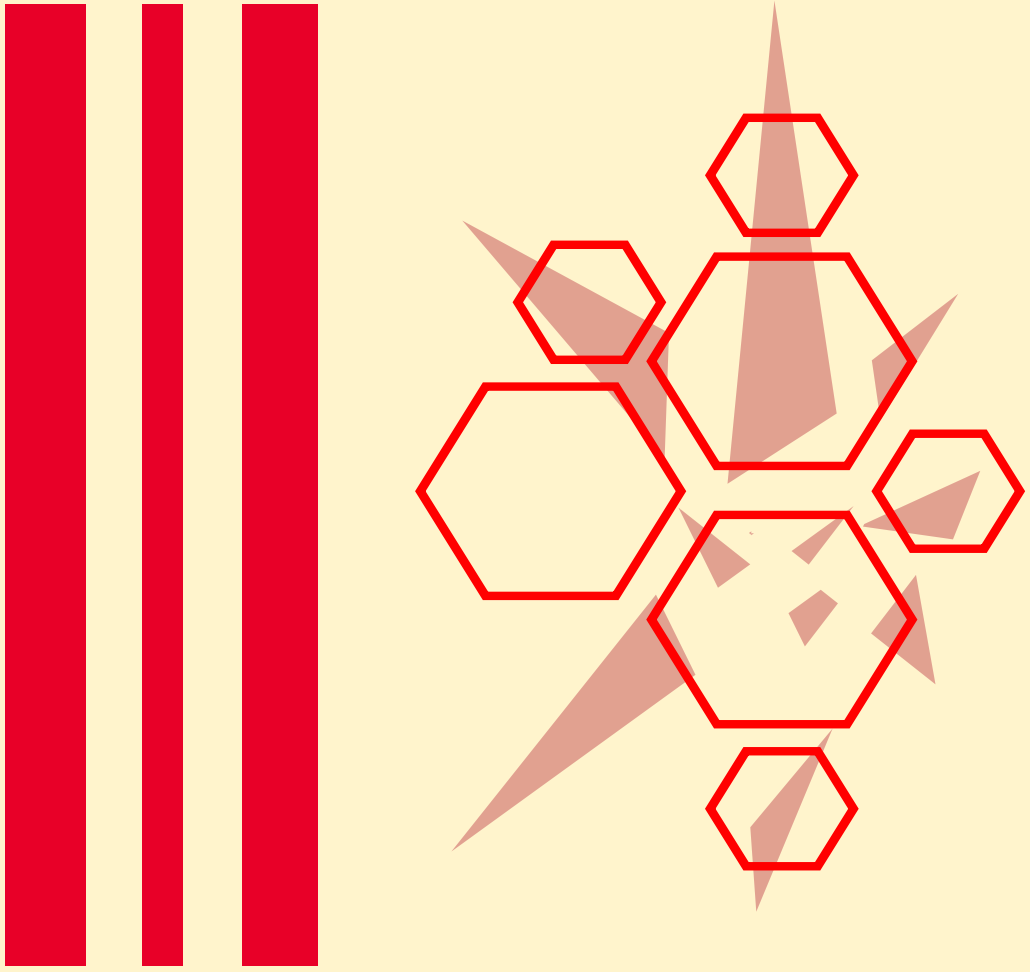

EXCEPTIONAL RUBY

by Avdi Grimm

Contents

Contents

List of listings

List of Figures

About

Exceptional Ruby

Exceptional Ruby was written using the wonderful Org-Mode package for Emacs. The PDF edition was generated using LaTeX. Source code highlighting is by Pygments, via the LaTeX "minted" package.

The cover is by RabidDesign.

This text originated as the talk "Exceptional Ruby", which I first gave in early 2011. It contains revised and extended examples, as well as a number of completely new sections.

History

- 2011-02-08 Initial
- 2011-02-09 Got started on intro and definitions.
- 2011-02-10 Up through exception safety testing.
- 2011-02-14 Beta
- 2011-02-24 Added section on dynamic exception lists for rescue. Added section on dynamic exception matchers. Colorized source listings.
- 2011-02-24 Added a section on `rescue` as a statement modifier.
- 2011-02-25 Fixed a couple of typos (Tim Knight). Made title casing more consistent. (George Anderson)

- 2011-02-26 Various typos and improvemens from George Anderson. Improved explanation of the stack unwinding process.
- 2011-02-26 Added an explanation of why `rescue` catches StandardError by default (George Anderson)
- 2011-03-06 Expanded on re-raising the current exception (George Anderson).
- 2011-03-06 Expanded the illustration of disallowing double-raise (George Anderson).
- 2011-03-06 Fixed the trap example (George Anderson).
- 2011-03-06 Made exception-in-thread example a little clearer (George Anderson).
- 2011-03-06 Lots of fixes from George Anderson.
- 2011-03-06 Renamed and expanded the handler method section.
- 2011-03-06 More fixes from George Anderson.
- 2011-03-15 Finished going through notes and corrections from George Anderson.
- 2011-03-15 Added much-expanded section on the “else” clause.
- 2011-04-26 Added section on module tags.
- 2011-04-27 Assorted PDF formatting improvements.
- 2011-04-27 Converted longer code samples into floating listings. Removed chapter/section numbers.
- 2011-04-28 Fixed figure size. Added nicer page headers.
- 2011-04-28 Added detailed commentary on exception tester implmentation.
- 2011-04-28 Added a bit more about design by contract (Dave Thomas)
- 2011-04-28 Improved exception performance section, and other tweaks (Dave Thomas)
- 2011-04-28 More edits from Dave Thomas
- 2011-04-28 Finished incorporating feedback from Dave Thomas.
- 2011-04-28 Incorporated errata from Sean Miller. Global spell check.
- 2011-04-28 Added a discussion of `Timeout::Error` (Sean Miller)

- 2011-04-28 Expanded a few explanations based on Pat Maddox' feedback.
- 2011-04-28 Incorporated feedback from Evan Light.
- 2011-04-28 Fixed a typo in the Rack example (Florent Guileux)
- 2011-04-28 Incorporated errata from Ginny Hendry. Added coverage of Cond library.
- 2011-04-28 Added mention of Nestegg gem.
- 2011-04-28 Misc tweaks.
- 2011-04-30 Fixed errata from George Anderson and Michel Demazure. Ditched the uppercase "Exception" convention.
- 2011-05-01 Numerous tweaks to the appearance of console output.
- 2011-05-01 Many small tweaks, mainly to source listings.
- 2011-05-01 Added section on "The no-raise library API"
- 2011-05-02 Added acknowledgements.
- 2011-05-02 Errata from Yale Kaul.
- 2011-05-03 Fixed a few incorrect output listings.

Acknowledgements

This book would not exist in its present form without the gracious help of a number of people.

Very special thanks go to George Anderson, who went above and beyond in providing copious, detailed proofreading notes and suggestions, constantly encouraged me, and who schooled me in the proper use of em-dashes.

Thank you to the technical reviewers who provided feedback on my early drafts: Evan Light, Jeff Casimir, Peter Cooper, Pat Maddox, Bryan Liles, and Dave Thomas.

Thanks to all of the Beta readers who bought the book in its unfinished form and thus gave me the confidence to keep working on it. Thanks especially to the Beta readers who sent me errata and other feedback: Sean Miller, Timothy Knight, Florent Guileux, Ginny Hendry, Michel Demazure, Yale Kaul.

Thank you to Jim Weirich, for sharing his failure-handling insights with me and helping clarify my thoughts on several points. Thanks to Larry Marburger, for always being around for me to bounce ideas off of. Thanks to the members of the Org-Mode mailing list for helping me with assorted LaTeX-export problems. Thanks to Robert Klemme, whose article "The Universe Between Begin and End" perfected my understanding of the `else` clause; and to Nick Sieger, for allowing me to reprint his code for printing out the Ruby exception heirarchy.

Thank you, finally, to my wife Stacey for being nothing but supportive of yet another project that keeps me in my office till all hours.

If I have missed anyone I sincerely apologize. Email me and I'll make sure you're included in future copies of the book.

Introduction

If software were a 10-year-old's bedroom, failure handling would be the tangled mass of toys and clothes crammed hastily into the closet before Mom comes to inspect. As programmers, we know that handling errors and failures is unavoidable, and on some projects we may spend half or more of our development hours chasing down tricky failure cases. Yet dealing with failure remains an afterthought in many projects, pushed off until it can no longer be ignored, and then dealt with in a piecemeal, haphazard fashion.

In most test suites, "happy path" tests predominate, and there may only be a few token failure case tests. Software books typically relegate handling failure to brief, cursory chapters—if the topic is addressed at all. And most mature codebases are riddled with the telltale signs of hastily-patched failure cases—business logic that is interrupted again and again by `nil`-checks and `begin...rescue...end` blocks.

This is a book about writing software with failure in mind. It's about how to write Ruby code that takes full advantage of the language's impressive array of features for signaling and handling unexpected conditions. And it is about approaching the problem of fallible software in a more disciplined, thoughtful way.

I have two goals for this book. I hope that after reading it, you will:

1. Have a more complete understanding of the mechanics of Ruby's failure handling facilities; and
2. Be equipped with the patterns and idioms you need in order to architect a robust failure-handling strategy for your apps and libraries.

Armed with a deep understanding of Ruby's powerful exception system and a toolbox full of failure-handling techniques, your code will be more reliable and easier to understand. It will crash less often, and when problems occur it will fail more gracefully. And you'll be a happier programmer, because you'll be spending more time on writing new features and less time chasing down unexpected exceptions.

What is a failure?

Definitions

So what exactly do we mean by “failure?” What about “error” and “exception?”

These are terms that are often used interchangeably in discussing software. In the interest of clarity, within this book I will adopt the definitions of these three terms used by Bertrand Meyer in his book “Object Oriented Software Construction”:

> An **exception** is the occurrence of an abnormal condition during the execution of a software element.
>
> A **failure** is the inability of a software element to satisfy its purpose.
>
> An **error** is the presence in the software of some element not satisfying its specification.
>
> . . . Note that failures cause exceptions. . . and are in general due to errors.

Take particular note of the definition of “error,” since it’s not the broader one typically used in software. In the following pages, “error” will be used to refer specifically to an error in implementation—in other words, a defect or bug. Of course, not all errors are simple mistakes; very often, they result from an incomplete understanding of the behavior of Ruby, third-party libraries, OS functions, or external services.

Failure as a breach of contract

So what does Meyer mean when he says “the inability of a software element to satisfy its purpose”?

Bertrand Meyer wrote the book, literally, on the notion of "Design by Contract". In this view, all methods have a "contract," either implicit or explicit, with their callers. A method can be said to have failed when it has failed to fulfill this contract.

Meyer also created a programming language, Eiffel, which embodied his ideas. Eiffel has never enjoyed widespread popularity. But its exception system—including the terminology, like "raise" and "rescue," as well as the "retry" mechanism—was a strong influence on Ruby's.

A method's contract states "given the following inputs, I promise to return certain outputs and/or cause certain side-effects". It is the callers responsibility to ensure that the method's *preconditions*—the inputs it depends on—are met. It is the method's responsibility to ensure that its *postconditions*—those outputs and side-effects—are fulfilled.

It is also the method's responsibility to maintain the *invariant* of the object it is a member of. The invariant is the set of conditions that must be met for the object to be in a consistent state. For instance, a `TrafficLight` class might have an invariant that only one lamp can be turned on at a time.

When a method's preconditions are met, but it is unable to either deliver on its promised postconditions, or to maintain the object invariant, then it is in breach of its contract; it has failed.

Reasons for failure

A method may fail to fulfill its contract for any of several reasons.

There may be a straight up mistake in the implementation of the method, such as a hash incorrectly indexed by a String instead of by Symbol.

```
h = {:count => 42}; h["count"] += 1
```

There might be a case that was unanticipated by the programmer at the time of writing.

```
def handle_message(message)
  if message.type == "success"
    handle_success(message)
  else
    handle_failure(message)
  end
end
# ...
message.type # => "stats"
handle_message(message) # Uh oh!
```

The system the program is running on may have run out of resources, such as memory space. In rare cases the system hardware may even be faulty, e.g. it may have failing RAM.

Or, some needed element external to the program and system entirely—such as a web service—might fail.

```
HTTP.get_response(url).code # => 500
```

Whatever the reason for the failure, a robust Ruby program needs to have a coherent plan in place for handling exceptional conditions. In the pages that follow I aim to equip you with the knowledge you need to assemble that plan. We'll start by taking a detailed look at how Ruby's exceptions work.

The life-cycle of an exception

> If code in one routine encounters an unexpected condition that it doesn't know how to handle, it throws an exception, essentially throwing up its hands and yelling "I don't know what to do about this—I sure hope somebody else knows how to handle it!"
>
> – *Steve McConnell*, Code Complete

In Ruby, we signal failures with exceptions. In the next few pages we'll take a whirlwind tour through the life-cycle of an exception—from inception, to its end in either a `rescue` block or the termination of the program.

Some of this may be information you already know. Feel free to skim, but stay sharp! I've peppered this section with tips and little excursions into lesser-known corners of the language. You might be surprised by some of the things you didn't know about exceptions in Ruby!

It all starts with a `raise` (or a `fail`)

Exceptions are raised with either the `raise` or `fail` methods. These two are synonyms; there is no difference in calling one or the other. Which you use is a matter of taste. In recent years `raise` has become much more common than `fail`, possibly as a side effect of the Rails codebases settling on the use of `raise`.

I traded some emails with Jim Weirich (http://onestepback.org/) in preparation for the talk which led to this book, and discovered that he has an interesting convention for how he uses `raise` and `fail`:

> I almost always use the "fail" keyword... [T]he only time I use "raise" is when I am catching an exception and re-raising it, because here I'm **not** failing, but explicitly and purposefully raising an exception.

Jim's convention makes a lot of sense to me; at the time of writing I'm considering adopting it for my own projects.

```
begin
 fail "Oops";
rescue => error
  raise if error.message != "Oops"
end
```

Listing 1: The Weirich raise/fail convention.

Calling `raise`

`raise` or `fail` can be called in a number of ways. The full method signature looks like this:

```
raise [EXCEPTION_CLASS], [MESSAGE], [BACKTRACE]
```

With no arguments, it creates a `RuntimeError` with no message.

```
raise
```

. . . is equivalent to:

```
raise RuntimeError
```

With just a string, it creates a `RuntimeError` with the given string as its message.

```
raise "Doh!"
```

. . . is equivalent to:

```
raise RuntimeError, "Doh!"
```

Given an explicit class (which must be a subclass of `Exception`[1]), it raises an object of the specified class.

```
raise ArgumentError, "Doh!"
```

A third argument can be supplied to `raise` to specify a custom backtrace for the exception. This is useful for example in assertion methods, where it's more useful for the backtrace to point to the site of the assertion than to the definition of `#assert`.

```
def assert(value)
  raise(RuntimeError, "Doh!", caller) unless value
end
assert(4 == 5)
```

Listing 2: A simple assertion method.

Evaluating the code above results in:

```
set_backtrace.rb:4: Failed (RuntimeError)
```

In this example we used `Kernel#caller` to supply the backtrace. `#caller` without any arguments will return a stack trace up to but *not* including the present line.

Overriding `raise` for fun and profit

`raise` and `fail` are not Ruby keywords. They are just methods defined on `Kernel`.

This is a significant fact, since in Ruby any method—even built-ins—can be overridden.

A simple `raise` replacement

Just as an example, we could make every exception instantly fatal by overriding `raise` to call `#exit!`:

[1] See Appendix B for more information on `Exception` and its subclasses.

```ruby
module RaiseExit
  def raise(msg_or_exc, msg=msg_or_exc, trace=caller)
    warn msg.to_s
    exit!
  end
end

module Kernel
  include RaiseExit
end
```

Listing 3: Making exceptions instantly fatal.

Note: This will only affect calls to `raise` in Ruby code. Exceptions raised in C extensions, e.g. with `rb_raise()`, will be unaffected. Note also that `fail` must be overridden separately.

What's the time? Hammertime!

For a somewhat more useful redefinition of `raise`, check out the Hammertime gem[2]. Hammertime is an error console for Ruby, inspired by the interactive error consoles in Smalltalk and some Lisp dialects. In those languages, when running code in development mode, signaling a failure causes an interactive dialog to appear asking the programmer if she would like to ignore the problem, continue processing normally, abort the current operation, or debug the source of the failure condition.

Hammertime provides a similar facility in Ruby. When an exception is raised, Hammertime presents a menu of options:

```ruby
require 'hammertime'
raise "Oh no!"
```

```
=== Stop! Hammertime. ===
    An error has occurred at example.rb:2:in
 'raise_runtime_error'
    The error is: #<RuntimeError: Oh no!>
    1. Continue (process the exception normally)
    2. Ignore (proceed without raising an exception)
    3. Permit by type (don't ask about future errors of this
 type)
```

[2] http://github.com/avdi/hammertime

```
   4. Permit by line (don't ask about future errors raised
from this point)
   5. Backtrace (show the call stack leading up to the error)
   6. Debug (start a debugger)
   7. Console (start an IRB session)
   What now?
```

`raise` internals

Calling `raise` does a little bit more than just make a new exception object and start unwinding the stack. The operation of `raise` can be broken down into four steps:

1. Call `#exception` to get the exception object.
2. Set the backtrace.
3. Set the global exception variable
4. Throw the exception object up the call stack.

Let's take a look at each of those in turn.

Step 1: Call `#exception` to get the exception.

Just by looking at the way it is called, you might think `raise` does something like the following:

```
def raise(klass, message, backtrace)
  error = klass.new(message) # Nope!
  # ...
end
```

In fact, `raise` never explicitly creates a new exception object. Instead, it uses the `#exception` method to request an exception object:

```
def raise(error_class_or_obj, message, backtrace)
  error = error_class_or_obj.exception(message)
  # ...
end
```

Ruby's `Exception` class defines `#exception` methods at both the class and instance level. When called at the class level, it is equivalent to calling `Exception.new`. Called at the instance level on an existing exception, it does one of two things:

1. When called with no arguments, it simply returns `self`.
2. When called with a message and, optionally, a stack trace, it returns a duplicate of itself. The new object is of the same class as the original, but has a new message (and maybe a new backtrace) set. Instance variables set in the original exception will also be copied to the duplicate exception.

In a sense, `#exception` behaves as Ruby's implicit "exception coercion" method. Just as Ruby implicitly calls `#to_proc` to convert an object into a Proc, or `#to_ary` to "splat" an object into an Array, Ruby calls `#exception` when it wants to get an Exception. We'll see below how this enables several different styles of re-raising Exceptions.

Implementing your own `#exception` methods

In practice, `Exception` is the only built-in class that actually implements this method. But that doesn't mean you couldn't implement your own `#exception` methods if you wanted. Consider a hierarchy of HTTP Response classes, where it might be useful to have failure-indicating responses generate their own Exceptions:

```
require 'net/http'
class Net::HTTPInternalServerError
  def exception(message="Internal server error")
    RuntimeError.new(message)
  end
end

class Net::HTTPNotFound
  def exception(message="Not Found")
    RuntimeError.new(message)
  end
end

response = Net::HTTP.get_response(
  URI.parse("http://avdi.org/notexist"))
if response.code.to_i >= 400
  raise response
end
```

Listing 4: Defining your own `exception` methods.

Note how at line 17, instead of specifying a specific exception type, we implicitly delegate the construction of an appropriate exception to the `response` object.

Step 2: `#set_backtrace`

Unless `raise` is being called to re-raise an active Exception, the next step is to attach a backtrace to the object. This is done with the rather un-Rubyish `#set_backtrace` method.

Stack traces in Ruby are simply an array of strings. The first elements in the array represent the lowest-level stack frames and the last elements are the highest. Each string is composed of three colon-delimited parts:

```
<FILE>:<LINE>:<CONTEXT>
```

E.g.:

```
prog.rb:2:in 'foo'
```

You can always grab the current stack trace with `#caller`[3]. Note, however, that in order to get a stack trace which includes the current line, you must call `#caller` passing `0` for the "start" parameter:

```
def foo
  puts "#caller: "
  puts *caller
  puts "----------"
  puts "#caller(0)"
  puts *caller(0)
end

def bar
  foo
end

bar
```

Output

```
#caller:
-:10:in `bar'
-:13
----------
#caller(0)
-:6:in `foo'
-:10:in `bar'
-:13
```

Listing 5: Accessing the stack trace.

Step 3: Set the global exception variable

The `$!` global variable always contains the currently "active" exception (if any). `raise` is responsible for setting this variable. The variable will be reset to `nil` if the exception is rescued and not re-raised.

For most day-to-day coding tasks you can go along blissfully unaware of the `$!` variable. But as we'll see in some of the coming sections, it can be quite useful when building on Ruby's exception-handling facilities

If `$!` is a little too obscure for you, you can require the `English` library and use the alias `$ERROR_INFO` instead.

Note that, like some of the other special "global" variables in Ruby, `$!` is only global in the context of the current thread. Separate threads get their

[3] http://www.ruby-doc.org/core/classes/Kernel.html#M001397

own `$!` variable, so raising an exception in one thread doesn't affect the exception state of another.

```
t1 = Thread.new do raise RuntimeError, "oops!" rescue $! end
t2 = Thread.new do raise ArgumentError, "doh!" rescue $! end
puts "main: #{$!.inspect}"
puts "t1: #{t1.value.inspect}"
puts "t2: #{t2.value.inspect}"
```

Output

```
main: nil
t1: #<RuntimeError: oops!>
t2: #<ArgumentError: doh!>
```

Listing 6: The `$!` variable in threads.

If it's not familiar to you, we'll explain the meaning of `rescue $!` in an upcoming section.

The following code demonstrates that `$!` is only set so long as an exception is active. It also demonstrates the `$ERROR_INFO` alias for `$!`:

```
require 'English'
puts $!.inspect
begin
  raise "Oops"
rescue
  puts $!.inspect
  puts $ERROR_INFO.inspect
end
puts $!.inspect
```

Output

```
nil
#<RuntimeError: Oops>
#<RuntimeError: Oops>
nil
```

Listing 7: Examining `$!`, aka `$ERROR_INFO`.

Step 4: Raise the exception up the call stack

Once the exception object has been prepared, `raise` begins to unwind the call stack. In this process Ruby works its way up the current call stack one method call or block evaluation at a time, looking for `ensure` or `rescue` statements. This process continues until either a matching `rescue` clause is found or the exception bubbles up to the top level of execution. If the latter happens, Ruby prints out the exception message and a stack trace, and terminates the program with a failure status.

`ensure`

`ensure` clauses will always be executed, whether an exception is raised or not. They are useful for cleaning up resources that might otherwise be left dangling.

Try to confine your `ensure` clauses to safe, simple operations. A secondary exception raised from an ensure clause can leave needed cleanup unperformed, as well as complicate debugging.

`ensure` with explicit return

There is a potentially unexpected edge case when using `ensure`. This case was documented by Les Hill on his blog[4].

If you explicitly `return` from a method inside an `ensure` block, the `return` will take precedence over any exception being raised, and the method will return as if no exception had been raised at all. In effect, the exception will be silently thrown away.

[4] http://blog.leshill.org/blog/2009/11/17/ensure-with-explicit-return.html

```
def foo
  begin
    raise Exception, "Very bad!"
  ensure
    return "A-OK"
  end
end

foo
```

Output

```
A-OK
```

Listing 8: Ensure with explicit return.

This is probably not what you want. My advice is to simply avoid using explicit returns inside `ensure` blocks.

Coming to the `rescue`

> In practice, the rescue clause should be a short sequence of simple instructions designed to bring the object back to a stable state and to either retry the operation or terminate with failure.
>
> – *Bertrand Meyer,* Object Oriented Software Construction

The syntax of a `rescue` clause starts with the rescue keyword, followed by one or more classes or modules to match, then a hash rocket (`=>`) and the variable name to which the matching exception will be assigned. Subsequent lines of code, up to an `end` keyword, will be executed if the `rescue` is matched. A typical example looks like this:

```
rescue IOError => e
  puts "Error while writing to file: #{e.message}"
  # ...
end
```

All elements except the `rescue` keyword itself are optional. As you may already know, a bare `rescue` with no arguments following it is not a catch-all. By default `rescue` only captures `StandardError` and derived classes.

```
rescue
  # ...
end
```

. . . is equivalent to:

```
rescue StandardError
  # ...
end
```

There is a fairly long list of built-in exception types that a bare `rescue` will *not* capture, including (but not limited to):

- `NoMemoryError`
- `LoadError`
- `NotImplementedError`
- `SignalException`
- `Interrupt`
- `ScriptError`

Why not capture the base `Exception` by default? The exceptions outside of the `StandardError` hierarchy typically represent conditions that can't reasonably be handled by a generic catch-all `rescue` clause. For instance, the `ScriptError` family of exceptions all represent conditions that probably indicate a mistake in assembling the program, such as a mistyped `require` statement. And `SignalException` is most often caused by the user signaling the process to end (e.g. by issuing a `kill` command)—and in the vast majority of cases, this should cause the program to end, not be swallowed by a catch-all `rescue`. So Ruby requires that you be explicit when rescuing these and certain other types of exception.

You can supply just a variable name to `rescue` without specifying an exception class. The syntax is a bit odd-looking:

```
rescue => error
  # ...
end
```

This is equivalent to:

```ruby
rescue StandardError => error
  # ...
end
```

The first example requires fewer keystrokes while the second is arguably more self-documenting. Whether the added specificity is worth the extra typing is a matter of personal taste.

As shown earlier, you can supply both an explicit class name and a name for the exception object:

```ruby
rescue SomeError => error
  # ...
end
```

You may also supply a list of classes or modules to match:

```ruby
rescue SomeError, SomeOtherError => error
  # ...
end
```

Order matters when stacking multiple `rescue` clauses:

```ruby
begin
  raise RuntimeError, "Specific error"
rescue StandardError => error
  puts "Generic error handler: #{error.inspect}"
rescue RuntimeError => error
  puts "Runtime error handler: #{error.inspect}"
end
```

Output

```
Generic error handler: #<RuntimeError: Specific error>
```

Listing 9: Order of `rescue` clauses matters.

`RuntimeError` is descended from `StandardError`, but the `rescue` clause for `RuntimeError` will never be reached because the preceding, more general `rescue` clause matches the `RuntimeErrors` first.

Dynamic `rescue` clauses

`rescue` clauses are similar in appearance and operation to another Ruby construct: the `case` statement. Just as with a `case` clause, you can supply a class or list of classes to be matched, followed by the code to be executed in case of a match.

```
# case
case obj
when Numeric, String, NilClass, FalseClass, TrueClass
  puts "scalar"
# ...
end

# rescue
rescue SystemCallError, IOError, SignalException
  # handle exception...
end
```

Listing 10: Comparing `case` and `rescue`.

`rescue` clauses share something else in common with `case` statements: the list of classes or modules to be matched doesn't have to be fixed. Here's an example of a method that suppresses all exceptions that match a given list of types. The list is "splatted" with the $*$ operator before being passed to `rescue`:

```
def ignore_exceptions(*exceptions)
  yield
rescue *exceptions => e
  puts "IGNORED: '#{e}'"
end

puts "Doing risky operation"
ignore_exceptions(IOError, SystemCallError) do
  open("NONEXISTENT_FILE")
end
puts "Carrying on..."
```

Output

```
Doing risky operation
IGNORED: 'No such file or directory - NONEXISTENT_FILE'
Carrying on...
```

Listing 11: Dynamic exception lists for `rescue`.

But that's not the end of the resemblance. As you may know, `case` works by calling the "threequals" (`===`) operator on each potential match. `rescue` works exactly the same way, but with an extra, somewhat arbitrary limitation:

```
# define a custom matcher that matches classes starting with "A"
starts_with_a = Object.new
def starts_with_a.===(e)
  /^A/ =~ e.name
end

begin
  raise ArgumentError, "Bad argument"
rescue starts_with_a => e
  puts "#{e} starts with a; ignored"
end
```

Output

```
#<TypeError: class or module required for rescue clause>
```

Listing 12: Limitations on exception matchers.

The sole difference between `case` matching semantics and `rescue` matching semantics is that the arguments to `rescue` *must* all be classes or modules.

But so long as we satisfy that requirement, we can define the match conditions to be anything we want. Here's an example that matches on the exception message using a regular expression:

```
def errors_with_message(pattern)
  # Generate an anonymous "matcher module" with a custom threequals
  m = Module.new
  (class << m; self; end).instance_eval do
    define_method(:===) do |e|
      pattern === e.message
    end
  end
  m
end

puts "About to raise"
begin
  raise "Timeout while reading from socket"
rescue errors_with_message(/socket/)
  puts "Ignoring socket error"
end
puts "Continuing..."
```

Output

```
About to raise
Ignoring socket error
Continuing...
```

Listing 13: A custom exception matcher.

We can generalize that to create exception matchers based on an arbitrary block predicate:

```
def errors_matching(&block)
  m = Module.new
  (class << m; self; end).instance_eval do
    define_method(:===, &block)
  end
  m
end

class RetryableError < StandardError
  attr_reader :num_tries
  def initialize(message, num_tries)
    @num_tries = num_tries
    super("#{message} (##{num_tries})")
  end
end

puts "About to raise"
begin
  raise RetryableError.new("Connection timeout", 2)
rescue errors_matching{|e| e.num_tries < 3} => e
  puts "Ignoring #{e.message}"
end
puts "Continuing..."
```

Output

```
About to raise
Ignoring Connection timeout (#2)
Continuing...
```

Listing 14: An exception matcher generator.

The `rescue` clause is powerful, and surprisingly dynamic. With dynamic type lists and custom matchers, you can be as specific as you need to be in matching exceptions to handlers.

`rescue` as a statement modifier

There is one final way to use `rescue`. In the same way that you can append an `if` or `unless` modifier to a Ruby statement, you can also append a `rescue`. A somewhat infamous example is the `rescue nil` modifier, which is sometimes used to ignore failures:

```
f = open("nonesuch.txt") rescue nil
```

When used as a modifier, `rescue` can't take any arguments except the expression to be evaluated if an exception is raised. It has the same semantics as a bare `rescue` clause at the end of a method or `begin` block: any exception deriving from `StandardError` is rescued.

The `rescue` modifier can be used as a succinct way of getting either the successful return value of a statement **or** the exception object which was raised:

```
file_or_exception = open("nonesuch.txt") rescue $!
```

Output

```
#<Errno::ENOENT: No such file or directory - nonesuch.txt>
```

If at first you don't succeed, `retry`, `retry` again

Ruby is one of the few languages that offers a `retry` facility. `retry` gives us the ability to deal with an exception by restarting from the last enclosing `begin` block.

Here's a very simple example:

```
tries = 0
begin
  tries += 1
  puts "Trying #{tries}..."
  raise "Didn't work"
rescue
  retry if tries < 3
  puts "I give up"
end
```

Output

```
Trying 1...
Trying 2...
Trying 3...
I give up
```

Listing 15: A demonstration of `retry`.

On line 2, a `tries` counter is defined. Before each attempt, the counter is incremented (line 3). When the attempts fail, they are caught, and the counter is examined (line 7). If it is under a threshold, execution restarts at line 3. Over the threshold, and the code gives up.

Using `retry` for operations you know to be unreliable can often be cleaner than the loop you would otherwise have to write. However, be very careful that your "giving up" criterion will be met eventually. It's easy to accidentally forget to increment a counter and get stuck in an infinite `retry` loop.

`raise` during exception handling

In some languages, such as C++, raising another exception while still in the progress of handling an exception is considered a bug, and will lead to the termination of the program. Ruby is more lenient in this regard; we can always raise a new Exception. What happens when we do depends partly on how we call the second `raise`.

One possibility is that we raise an entirely new Exception:

```
begin
  raise "Failure A"
rescue
  raise "Failure B"
end
```

Listing 16: Exception replacement.

In this case the original exception is **thrown away**. There is no way to retrieve it.

This can be a real headache when debugging the code, because you can work your way back to the origin of an exception only to discover that it was raised while handling a different failure—and there is no way to discover what the original exception was.

Incidentally, I've discovered several cases where the Ruby on Rails codebase does exactly this; in every case it complicated my attempts to resolve the issue that I was debugging. My advice is to never allow the original exception to be thrown away. Instead, use *Nested Exceptions*.

Nested exceptions

Nested Exceptions[5] are simply exceptions which have a slot for a reference to the originating exception, if any. You can also think of them as *wrapped exceptions*, since they effectively wrap the original exception inside a new

[5] http://c2.com/cgi/wiki?NestedException

one. Ruby's built-in Exceptions don't support nesting, but adding nesting to your own exception classes is trivial.

```
class MyError < StandardError
  attr_reader :original
  def initialize(msg, original=$!)
    super(msg)
    @original = original;
  end
end

begin
  begin
    raise "Error A"
  rescue => error
    raise MyError, "Error B"
  end
rescue => error
  puts "Current failure: #{error.inspect}"
  puts "Original failure:  #{error.original.inspect}"
end
```

Listing 17: A nested exception implementation.

We've been a little clever with the definition of `MyError`, by making the default argument to `original` be the global error variable (`$!`) (see line 3). This enables the object to "magically" discover the originating exception on initialization. If there is no currently active exception the `original` attribute will be set to `nil`.

We can also explicitly set the original Exception:

```
raise MyError.new("Error B", error)
```

John D. Hume has created a gem "Nestegg"[6] which implements nested exceptions with some nice backtrace formatting.

More ways to re-raise

Calling `raise` on the original exception simply raises the same exception object again:

[6] http://nestegg.rubyforge.org/

```
begin
  begin
    raise "Error A"
  rescue => error
    puts error.object_id.to_s(16)
    raise error
  end
rescue => error
  puts error.object_id.to_s(16)
end
```

Listing 18: Re-raising the same exception.

When we run this we can see that no new exception object is created:

Output

```
3fe7aa5efa70
3fe7aa5efa70
```

(Your `object_id` values will be different, of course.)

Calling `raise` with an existing exception and a new message raises a new, duplicate exception object with the new message set. The original exception object is unchanged.

```
begin
  begin
    raise "Error A"
  rescue => error
    puts error.inspect + ": " + error.object_id.to_s(16)
    raise error, "Error B"
  end
rescue => error
  puts error.inspect + ": " + error.object_id.to_s(16)
end
```

Listing 19: Re-raising with a new message.

Running this code confirms that the re-raised exception is a different object:

Output

```
#<RuntimeError: Error A>: 3f84556740f8
#<RuntimeError: Error B>: 3f8455674030
```

Likewise, we can add a custom backtrace to the new Exception:

```
begin
  begin
    raise "Error A"
  rescue => error
    puts error.backtrace.first
    raise error, "Error B", ["FAKE:42"]
  end
rescue => error
  puts error.backtrace.first
end
```

Output

```
-:3
FAKE:42
```

Listing 20: Re-raising with a modified backtrace.

Re-raising with no arguments simply re-raises the currently active Exception:

```
begin
  raise "Error A"
rescue => error
  raise
end
```

. . . is equivalent to:

```
begin
  raise "Error A"
rescue => error
  raise error
end
```

. . . which is equivalent to:

```
begin
  raise "Error A"
rescue
  raise $!
end
```

Disallowing double-raise

Time to have some more fun overriding `raise`. As I mentioned earlier, some languages disallow raising new errors while another exception is being handled. This is because a double-raise situation can make debugging more difficult, as well as potentially result in resources not being cleaned up.

Let's say we wanted to emulate those languages and introduce a little more discipline into our exception handling. As usual, Ruby has the flexibility to oblige us.

The following code redefines `raise` to prevent a secondary exception from being raised until the active exception has been handled:

```
module NoDoubleRaise
  def error_handled!
    $! = nil
  end

  def raise(*args)
    if $! && args.first != $!
      warn "Double raise at #{caller.first}, aborting"
      exit! false
    else
      super
    end
  end
end

class Object
  include NoDoubleRaise
end
```

Listing 21: Disallowing double-raise.

With this code in place, a secondary failure will cause the program to end.

```
begin
  raise "Initial failure"
rescue
  raise "Secondary failure"
end
```

Output

```
Double raise at (irb):25:in ‘irb_binding’, aborting
(Program exited unsuccessfully)
```

Output

```
Double raise at (irb):1248:in ‘irb_binding’, aborting
(Program exited unsuccessfully)
```

Our code must explicitly declare the current exception to be handled (by calling `error_handled!`) in order to raise a new exception.

```
begin
  raise "Initial failure"
rescue
  error_handled!
  raise "Secondary failure"
end
```

Output

```
RuntimeError: Secondary failure
      from (irb):83:in ‘raise’
      from (irb):121
```

else

`else` after a `rescue` clause is the opposite of `rescue`; where the `rescue` clause is only hit when an exception is raised, `else` is only hit when *no* exception is raised by the preceding code block.

```
def foo
  yield
rescue
  puts "Only on error"
else
  puts "Only on success"
ensure
  puts "Always executed"
end

foo{ raise "Error" }
puts "---"
foo{ "No error" }
```

Output

```
Only on error
Always executed
---
Only on success
Always executed
```

Listing 22: Using `else`.

Why would we ever use `else` instead of simply putting the else-code *after* the code which might raise an exception?

Let's say we have a logging method which, unfortunately, doesn't work very well:

```
def buggy_log(message)
  raise "Failed to log message"
end
```

Let's also posit that we sometimes want to be able to call potentially failure-prone code, ignore any failures, and log the result:

```
def ignore_errors(code)
  eval(code)
  buggy_log "Success running:\n  #{code}"
rescue Exception => error
  puts "Error running code:\n  #{error}"
end

ignore_errors("puts 'hello world'")
```

Output

```
hello world
Error running code:
  Failed to log message
```

That doesn't look right. The code succeeded, but the method reports that it failed because of the buggy logging method. Let's try again, only this time using `else` to report success:

```
def ignore_errors_with_else(code)
  eval(code)
rescue Exception => error
  puts "Error running code: #{error}"
else
  buggy_log "Success running #{code}"
end

ignore_errors_with_else("puts 'hello world'")
```

Output

```
hello world
RuntimeError: Failed to log message
      from (irb):2:in `buggy_log'
      from (irb):22:in `ignore_errors_with_else'
      from (irb):25
```

Listing 23: Code in `else` is not covered by `rescue`.

Now we have a less misleading message: the `eval` didn't fail, it was the `#buggy_log` method that failed.

What this code demonstrates is that unlike code executed in the main body of the method or `begin` block, exceptions raised in an `else` block are *not* captured by the preceding `rescue` clauses. Instead they are propagated up

to the next higher scope of execution.

Here's a quick illustration of the order of processing when there is an `else` clause:

```
begin
  puts "in body"
rescue
  puts "in rescue"
else
  puts "in else"
ensure
  puts "in ensure"
end
puts "after end"
```

Output

```
in body
in else
in ensure
after end
```

Listing 24: Order of clauses.

As you can see, processing proceeds from top to bottom. Ruby enforces this ordering of `rescue`, `else`, `ensure` clauses, so the order of processing will always match the visual order in your code.

Uncaught exceptions

What happens if an exception is never rescued? Eventually, the stack will unwind completely and Ruby will handle the un-rescued exception by printing a stack trace and terminating the program. However, before the program ends Ruby will execute various exit hooks:

```ruby
trap("EXIT") { puts "in trap" }
at_exit { puts "in at_exit" }
END { puts "in END" }

raise "Not handled"
```

```
in trap
in END
in at_exit
unhandled.rb:5: Not handled (RuntimeError)
```

Listing 25: Exit handlers.

I'm not going to go into the difference between `END` blocks, `at_exit`, and `trap` in this text. The important thing is that all of them are executed before the program terminates, and all have access to global variables like `$!`.

This fact is useful to us. Let's say we wanted to log all fatal exception-induced crashes. Let's say, further, that our code is running in a context where it is difficult or impossible to wrap the entire program in a `begin ... rescue ... end` block. (E.g. a web application server running on hosted server that we don't control). We can still "catch" Exceptions before the program exits using hooks.

Here's a simple crash logger implemented with `at_exit`:

```ruby
at_exit do
  if $!
    open('crash.log', 'a')  do |log|
      error = {
        :timestamp => Time.now,
        :message   => $!.message,
        :backtrace => $!.backtrace,
        :gems      => Gem.loaded_specs.inject({}){
          |m, (n,s)| m.merge(n => s.version)
        }
      }
      YAML.dump(error, log)
    end
  end
end
```

Listing 26: A crash logger using `at_exit`

This code is executed when the program exits. It checks to see if the exit is because of an exception by checking the `$!` variable. If so, it logs some information about the exception to a file in YAML format. Specifically, it logs a timestamp, the failure message, the backtrace, and the versions of all Rubygems loaded at the time of the crash. You can probably think of other useful information to put in a crash log.

Exceptions in threads

Un-handled exceptions raised in threads (other than the main thread) work a little differently. By default, the thread terminates and the exception is held until another thread joins it, at which point the exception is re-raised in the `join`-ing thread.

```
t = Thread.new { raise "oops" }
sleep 1 # give it plenty of time to fail
begin
  t.join
rescue => error
  puts "Child raised error: #{error.inspect}"
end
```

Output

```
Child raised error: #<RuntimeError: oops>
```

Listing 27: Exceptions in threads are held until joined.

Are Ruby exceptions slow?

Raising an exception is typically an expensive operation in any language, and Ruby is no, um, exception.

Unlike some languages, there is no penalty simply for having exception-handling code lying around dormant:

```
def fib(n)
  return n if n < 2
  vals = [0, 1]
  n.times do
    vals.push(vals[-1] + vals[-2])
  end
  return vals.last
end

def fib_with_rescue(n)
  return n if n < 2
  vals = [0, 1]
  n.times do
    vals.push(vals[-1] + vals[-2])
  end
  return vals.last
rescue IOError
  puts "Will never get here"
end

require 'benchmark'
Benchmark.bm(10) { |bm|
  bm.report("no rescue") { 1000.times { fib(100)} }
  bm.report("rescue") {
    1000.times { fib_with_rescue(100) }
  }
}
```

Output

```
                 user     system      total        real
no rescue    0.070000   0.000000   0.070000 (  0.064622)
rescue       0.060000   0.000000   0.060000 (  0.062675)
```

Listing 28: Performance with unused exception-handling code.

On the other hand, writing code that uses exceptions as part of its logic can have a significant performance cost, as in the following highly-contrived example:

```ruby
def fib(n)
  return n if n < 2
  vals = [0, 1]
  n.times do
    vals.push(vals[-1] + vals[-2])
  end
  return vals.last
end

def fib_raise(n)
  return n if n < 2
  vals = [0, 1]
  i = 0
  loop do
    vals.push(vals[-1] + vals[-2])
    i += 1
    raise if i >= n
  end
rescue RuntimeError
  return vals.last
end

require 'benchmark'
Benchmark.bm(10) { |bm|
  bm.report("fib") { 1000.times { fib(100)} }
  bm.report("fib_raise") {
    1000.times { fib_raise(100) }
  }
}
```

Output

```
                 user     system      total        real
fib          0.070000   0.000000   0.070000 (  0.070222)
fib_raise    0.130000   0.000000   0.130000 (  0.130710)
```

Listing 29: Performance with exceptions as logic.

This doesn't mean we should avoid using exceptions. It simply means we should reserve them for genuinely exceptional circumstances.

Responding to failures

Once an exception is rescued, what next? There are a lot of ways to deal with a failure. This next section covers some of the options.

Failure flags and benign values

If the failure isn't a major one, it may be sufficient to return a failure flag. In Ruby code, the most common failure flag is `nil`. A `nil` isn't very communicative to the calling code, but in some cases it may be all that's needed.

```
def save
  # ...
rescue
  nil
end
```

A related approach is to return a known benign value. As Steve McConnell puts it in Code Complete:

> The system might replace the erroneous value with a phony value that it knows to have a benign effect on the rest of the system.

This is an oft-overlooked technique that can convert an unnecessarily hard failure into a gentler fallback mode. When the system's success doesn't depend on the outcome of the method in question, using a benign value may be the right choice. Benign values are also helpful in making code more testable; when unavailable services or configuration don't cause the whole system to crash, it's easier to focus on the behavior under test.

```
begin
  response = HTTP.get_response(url)
  JSON.parse(response.body)
rescue Net::HTTPError
  {"stock_quote" => "<Unavailable>"}
end
```

Listing 30: Returning a benign value.

Reporting failure to the console

The simplest way to to report a failure to the user (or developer) is the humble `puts`.

`puts` leaves something to be desired as a failure reporting method, however. Because its output goes to `$stdout`, which is buffered by default, output may not be printed immediately, or it may come out in a surprising order:

```
puts "Uh oh, something bad happened"
$stderr.puts "Waiting..."
sleep 1
puts "Done"
```

Output

```
Waiting...
Uh oh, something bad happened
Done
```

You could address this shortcoming by calling `$stderr.puts` instead. But there's a better way: use `warn`.

```
warn "Uh oh, something bad happened"
```

`warn` is more succinct, while still using `$stderr` as its output stream. Note also that the output of `warn` can be temporarily silenced by passing the `-W0` flag to Ruby.

Warnings as errors

We all know that allowing warnings to proliferate in our programs can lead to legitimate problems being overlooked. Wouldn't it be nice if there were some easy way to track down the source of all the messages our program is spitting out?

If you've been careful to use `warn` consistently, you can use this a handy trick for temporarily turning warnings into exceptions:

```
module Kernel
  def warn(message)
    raise message
  end
end

warn "Uh oh"
```

Output

```
 -:3:in ‘warn’: Uh oh (RuntimeError)
       from -:7
```

Listing 31: Warnings as errors.

With `warn` redefined to raise exceptions, finding the source of the warnings is a matter of following the stack traces. Just remember to remove the overridden `warn` once you're finished eliminating the sources of the warnings!

An alternative to removing the overridden `warn` is to wrap the definition in a test for `$DEBUG`:

```
if $DEBUG
  module Kernel
    def warn(message)
      raise message
    end
  end
end
```

Listing 32: Optional warnings as errors.

This way `warn` will only raise an exception if you run Ruby with the `-d` flag.

Remote failure reporting

In some cases you may want to report failures to some central location. This might take any of several forms:

- A central log server
- An email to the developers
- A post to a third-party exception-reporting service such as Hoptoad[7], Exceptional[8], or New Relic RPM[9].

Collecting failure reports in a central location is an excellent way to stay on top of the health of your app. But beware: if you aren't careful, a well-intentioned effort to make your app more transparent can lead instead to worse failures. The following story illustrates such a situation.

A true story of cascading failures

I once worked on a project that was composed of many cooperating applications. This included a large number of worker processes which accepted jobs from a central queue. Sometimes, these jobs would fail.

We installed a simple failure reporter in the workers. It used our team's Gmail account to email the developers with the details of the failure. For a while, this worked well.

One day we pushed out an update which had the unintended effect of causing the workers to fail much more often. In fact, they failed so often that the repeated failure emails caused Google to throttle our email account.

From the perspective of the error reporter, the account throttling took the form of unexpected SMTP exceptions being raised. The software had not been written to handle SMTP errors, and instead of just reporting failure and grabbing another job, the workers started crashing hard. Once they had crashed, they would no longer process jobs until they had been restarted.

But that wasn't the end of it. Because other, unrelated systems also used the same Gmail account to generate notifications, they too started getting SMTP Exceptions and crashing. Failures in the workers caused a chain reaction that led to the entire distributed system breaking down.

This is a classic example of a *failure cascade*. It illustrates a common problem in software systems: because systems are typically built with most of the attention paid to the steady-state—and because failure conditions are

[7] http://hoptoadapp.com/pages/home
[8] http://www.getexceptional.com/
[9] http://newrelic.com/

often hard to simulate—failure-handling subsystems are often some of the most fragile and untested in the project. Like a decaying life raft that has not been inspected since its first installation, the failure handler itself fails when it is finally called into action.

Bulkheads

> In a ship, *bulkheads* are metal partitions that can be sealed to divide the ship into separate, watertight compartments. Once hatches are closed, the bulkhead prevents water from moving from one section to another. In this way, a single penetration of the hull does not irrevocably sink the ship. The bulkhead enforces a principle of damage containment.
>
> You can employ the same technique. By partitioning your systems, you can keep a failure in one part of the system from destroying everything.
>
> – *Michael Nygard,* Release It!

One way to avoid failure cascades is to erect *bulkheads* in your system. A bulkhead (or "barricade", depending on who is describing it) is a wall beyond which failures cannot have an effect on other parts of the system. A simple example of a bulkhead is a `rescue` block that catches all Exceptions and writes them to a log:

```
begin
  SomeExternalService.some_request
rescue Exception => error
  logger.error "Exception intercepted calling SomeExternalService"
  logger.error error.message
  logger.error error.backtrace.join("\n")
end
```

Listing 33: A simple bulkhead.

Where should you put bulkheads? In find it's a good idea to put bulkheads between your app and:

- External **services**
- External **processes**

In my experience these are a) some of the most common sources of unexpected failures; and b) areas where, at the very least, you want to collect additional information about the failure before raising an exception up the stack and losing all context surrounding the failure.

The Circuit Breaker pattern

Another way to mitigate failure cascade situations is to build *Circuit Breakers* into your apps.

Circuit Breaker is a pattern described in Micheal Nygard's book *Release It!*. It describes a mechanism that controls the operation of a software subsystem and behaves similarly to a physical circuit breaker. The mechanism has three states:

When the breaker is **Closed**, the subsystem is allowed to operate normally. However, a counter tracks the number of failures that have occurred. When the number exceeds a threshold, the breaker trips, and enters the **open** state. In this state, the subsystem is not permitted to operate.

After either a timeout elapses or (depending on the implementation) a human intervenes and resets the breaker, it enters the **half-open** state. In this state the subsystem is on probation: it can run, but a single failure will send it back into the "open" state.

By detecting unusually frequent failures, and preventing the subsystem from endlessly repeating a failing operation, the Circuit Breaker pattern eliminates a common cause of failure cascades.

If you are interested in exploring the use of Circuit Breakers in your code, Will Sargent has made a Ruby implementation of the pattern available on Github[10].

Ending the program

Some failures are sufficiently severe that the only way to handle them is to end the program. There are a few ways to accomplish that end.

Calling `exit` with a nonzero value is probably the first method that comes to mind. Something that you may not know about `exit` is that when you call it, you're actually raising an Exception. Specifically, you're raising `SystemExit`.

So this code:

[10] https://github.com/wsargent/circuit_breaker

```
warn "Uh oh"
exit 1
```

. . . is equivalent to:

```
warn "Uh oh"
raise SystemExit.new(1)
```

Now, if you want to print a failure message and then raise `SystemExit` with a failure status code, there's a more succinct way to do it: `abort`.

```
abort "Uh oh"
puts "Will never get here."
```

If you **really** want to end the program quickly—without going through any of the usual cleanup, exit hooks, and finalizer calls—you want `exit!` instead:

```
warn "Alas, cruel runtime!"
exit! false
```

This will immediately terminate the process with a failure status.

Alternatives to exceptions

Fail fast[11] is a common mantra for dealing with failure in software, and more often than not it's the best policy. Once an unexpected condition has occurred, continuing to run may do more harm than good.

There are certain situations, however, when failing fast may not yield the best results. For example, when we run an automated test suite with tools such as Test::Unit or RSpec, the test run doesn't end the moment an error occurs in one of the tests. The test framework assumes that we want to know how many failures or errors there are in total, so it keeps running and gives us a detailed report at the very end.

Here's another example. Let's say you have a script that provisions a new server instance on a cloud-computing service. In order to accomplish this the script performs numerous steps—updating configuration files, installing software, starting daemons, etc. It's a time-consuming process.

One of the steps is to download a list of SSH public keys from a remote web service. However, you've discovered that for some reason the key download doesn't always proceed successfully. One morning, after kicking off twenty iterations of the script the night before, you find that that half the server builds had been aborted in the middle because of a "keyfile download" exception. What's worse, the incomplete server instances had been automatically deleted as part of the exception cleanup code, so not only did you have to start the server builds again from scratch, but there were no log files left around to help you get to the bottom of the unpredictable fault.

In this situation, failing early clearly isn't the most useful approach. What you *really* need is a way to proceed through the steps of the provisioning process and then get a report at the end telling you what parts succeeded and what parts failed.

There are several approaches to to implementing this kind of deferred/optional failure handling; but all of them are variations on the theme of *sideband data*.

[11] http://c2.com/cgi/wiki?FailFast

Sideband data

Every method has a primary channel of communication back to its caller: its return value. When communicating failures without resorting to exceptions, we need a *side band*: a secondary channel of communication for reporting meta-information about the status and disposition of a process.

Multiple return values

The simplest possible sideband is using multiple return values. Ruby effectively supports multiple return values in the form of its "array splatting" assignment semantics. That is, an array returned from a method and assigned to a list of variables will be automatically de-structured into the respective variables:

```
def foo
  result  = 42
  success = true
  [result, success]
end
result, success = foo
puts "#{success ? 'Succeeded' : 'Failed'}; result: #{result}"
```

Output

```
Succeeded; result: 42
```

Listing 34: Multiple return values.

The downside of this approach is that anyone calling the method must know that it returns multiple values; otherwise, they'll be working with an Array when they don't expect it.

A variation on multiple return values is to return a simple structure with attributes for the result value and for the status:

```
def foo
  # ...
  OpenStruct.new(:result => 42, :status => :success)
end
```

Listing 35: Returning a structure.

The returned object is a bit more self-describing than an Array. However, the caller still has to know that the return will be in this format, and the need to append `.result` to the method just to get the "real" return value can feel a bit redundant:

```
result = foo
results << result.result
```

Using multiple return values works fine for simple cases, but it quickly breaks down when we want return the status information from several levels deep of method execution. At every level we have to be careful to preserve the sideband data from all called methods and return it to the caller. For more complex APIs we need a different approach.

Output parameters

I know, I know. If you came from C, C# or Java you're probably shaking your head. You thought programming in Ruby meant you'd never see another "output parameter".

In general that's true. And I'm not talking about output parameters in the traditional sense of a parameter that you pass in empty and which is then assigned a value by the called method. However, there's a similar pattern that sometimes makes sense to use.

When I'm writing code that executes shell commands, I like to provide a `transcript` parameter for capturing the output of the entire session:

```
require 'stringio'

def make_user_accounts(host, transcript=StringIO.new)
  transcript.puts "* Making user accounts..."
  # ... code that captures STDOUT and STDERR to the transcript
end

def install_packages(host, transcript=StringIO.new)
  transcript.puts "* Installing packages..."
  # ... code that captures STDOUT and STDERR to the transcript
end

def provision_host(host, transcript)
  make_user_accounts(host, transcript)
  install_packages(host, transcript)
end

transcript = StringIO.new
provision_host("192.168.1.123", transcript)
puts "Provisioning transcript:"
puts transcript.string
```

Listing 36: Passing in a transcript parameter.

Caller-supplied fallback strategy

We'll take a longer look at this technique a little later. But the gist of it is this: if we're not sure we want to terminate the execution of a long process by raising an exception, we can inject a failure policy into the process in much the same way that we injected a transcript above.

```
def make_user_accounts(host, failure_policy=method(:raise))
  # ...
rescue => error
  failure_policy.call(error)
end

def install_packages(host, failure_policy=method(:raise))
  # ...
  raise "Package 'foo' install failed on #{host}"
rescue => error
  failure_policy.call(error)
end

def provision_host(host, failure_policy)
  make_user_accounts(host, failure_policy)
  install_packages(host, failure_policy)
end

policy = lambda {|e| puts e.message}
provision_host("192.168.1.123", policy)
```

Output

```
Package 'foo' install failed on 192.168.1.123
```

Listing 37: Caller-supplied fallback strategy.

Global variables

In languages that don't have exceptions (such as C), one common approach to reporting failures is for a library to designate a global variable to contain the error code (if any) for the most recently executed function. Functions which wish to check for errors must do so by looking at the value of the global variable.

We can use this approach in Ruby as well. But rather than use a global variable (variable beginning with "$"), we'll use a thread-local variable. That way we'll avoid creating potential thread-safety bugs.

```
class Provisioner
  def provision
    # ...
    (Thread.current[:provisioner_errors] ||= []) <<
      "Error getting key file..."
  end
end
p = Provisioner.new
p.provision

if Array(Thread.current[:provisioner_errors]).size > 0
  # handle failures...
end
```

Listing 38: Using a thread-local variable for failure reporting.

If you're not familiar with the call to `Array()` on line 11, it intelligently coerces its argument into an Array[12]. `nil` will become an empty Array.

Note that we namespace the thread-local variable by prefixing it with `:provisioner_`. Since thread-local variables have global scope inside the thread, we must be careful to avoid using variable names which might be used by other libraries.

The advantage with the thread-local variable approach is that an error could be added to the `:provisioner_errors` array at any level of nested method calls, and it would still be found by the error-checking code. The disadvantage is that unless we are diligent about resetting the variable, there is no way of knowing for sure if its value is from the preceding call to `#provision`, or from some other prior and unrelated call.

We can mitigate the latter issue somewhat by creating a helper method to execute an arbitrary block of code and then check for failures, cleaning up the error variable when it's done:

[12] http://www.ruby-doc.org/core/classes/Kernel.html#M001437

```
def handle_provisioner_failures
  Thread.current[:provisioner_errors] = []
  yield
  if Array(Thread.current[:provisioner_errors]).size > 0
    # handle failures...
  end
ensure
  Thread.current[:provisioner_errors] = []
end

handle_provisioner_failures do
  # ...any number of Provisioner calls...
end
```

Listing 39: Thread-local variable cleanup.

Process reification

The final—and arguably the cleanest—solution is to represent the process itself as an object, and give the object an attribute for collecting status data:

```
class Provisionment
  attr_reader :problems
  def initialize
    @problems = []
  end
  def perform
    # ...
    @problems << "Failure downloading key file..."
  end
end

p = Provisionment.new
p.perform
if p.problems.size > 0
  # ... handle errors ...
end
```

Listing 40: Representing a process as an object.

This avoids the awkwardness of multiple return values, and the global

namespace pollution of using thread-local variables. The drawback is simply that it is a “heavier” solution in that it adds another class to the system.

Beyond exceptions

I’ve referred several times to the power of Ruby’s exception handling system, and it is indeed one of the most flexible and expressive exception systems to be found in any programming language. But exceptions aren’t the be-all and end-all of failure handling. Common Lisp, for instance, has long boasted a feature known as “conditions” which can do everything Ruby’s exceptions can do and a few things it can’t.

James M. Lawrence has taken on the formidable task of bringing Lisp’s condition system to Ruby with his “Cond” library[13]. The tagline of the library is “Resolve errors without unwinding the stack.”

I haven’t had a chance to dig into Cond yet, but it looks very promising. As a teaser here’s the synopsis, taken directly from the README:

```
equire 'cond/dsl'

def divide(x, y)
  restartable do
    restart :return_this_instead do |value|
      return value
    end
    raise ZeroDivisionError if y == 0
    x/y
  end
end

handling do
  handle ZeroDivisionError do
    invoke_restart :return_this_instead, 42
  end
  puts divide(10, 2)  # => 5
  puts divide(18, 3)  # => 6
  puts divide(4, 0)   # => 42
  puts divide(7, 0)   # => 42
end
```

Listing 41: The Cond gem.

[13] http://cond.rubyforge.org/

Your failure handling strategy

> Error processing is turning out to be one of the thorniest problems of modern computer science, and you can't afford to deal with it haphazardly.
>
> – *Steve McConnell*, Code Complete

This next section is something of a grab-bag. It's a collection of tips and guidelines that I and others have found to be helpful in structuring a robust failure handling strategy for apps and libraries.

Exceptions shouldn't be expected

> **Use exceptions only for exceptional situations.** [...] Exceptions are often overused. Because they distort the flow of control, they can lead to convoluted constructions that are prone to bugs. It is hardly exceptional to fail to open a file; generating an exception in this case strikes us as over-engineering.
>
> – *Brian Kernighan and Rob Pike*, The Practice of Programming

It's tempting to raise an exception every time you get an answer you don't like. But exception handling interrupts and obscures the flow of your business logic. Consider carefully whether a situation is *truly* unusual before raising an Exception.

A good example of this consideration can be found in the ActiveRecord library. `#save` doesn't raise an exception when the record is invalid, because invalid user input isn't that unusual.

Jared Carroll wrote a great blog post[14] about this point back in 2007, in which he puts it this way:

> When writing an application you expect invalid input from users. Since we expect invalid input we should NOT be handling it via exceptions because exceptions should only be used for unexpected situations.

A rule of thumb for raising

The Pragmatic Programmer suggests a good rule of thumb for determining if exceptions are being overused:

> . . . ask yourself, 'Will this code still run if I remove all the exception handlers?" If the answer is "no", then maybe exceptions are being used in non-exceptional circumstances.
>
> – *Dave Thomas and Andy Hunt,* The Pragmatic Programmer

Use `throw` for expected cases

Sometimes you really want the kind of non-local return that raising an exception can provide, but for a non-exceptional circumstance.

Ruby, always eager to please, has an answer for that: use `throw` and `catch`. Do not be confused by the naming similarity to methods used to raise and rescue exceptions in other languages; in Ruby, `throw` and `catch` are all about providing a way to quickly break out of multiple levels of method calls in *non*-exceptional circumstances.

Rack and Sinatra provide a great example of how `throw` and `catch` can be used to terminate execution early. Sinatra's `#last_modified` method looks at the HTTP headers supplied by the client and, if they indicate the client already has the most recent version of the page, immediately ends the action and returns a "Not modified" code. Any expensive processing that would have been incurred by executing the full action is avoided.

```
get '/foo' do
  last_modified some_timestamp
  # ...expensive GET logic...
end
```

[14] http://robots.thoughtbot.com/post/159807850/save-bang-your-head-active-record-will-drive-you-mad

Here's a simplified version of the `#last_modified` implementation. Note that it throws the `:halt` symbol. Rack catches this symbol, and uses the supplied response to immediately reply to the HTTP client. This works no matter how many levels deep in method calls the `throw` was invoked.

```
def last_modified(time)
  response['Last-Modified'] = time
  if request.env['HTTP_IF_MODIFIED_SINCE'] > time
    throw :halt, response
  end
end
```

Listing 42: Use of `throw`/`catch` in Rack.

What constitutes an exceptional case?

Answers to the question "what qualifies as an exception?" have filled many articles, blog posts, and book pages. It's one of the longstanding debates in software engineering.

Is an EOF a failure? How about a missing key in a hash? What about a 404 status code from a web service?

The answer to all of these is: **it depends!** Whether a particular event or condition constitutes an exceptional situation is not a question about which we can make blanket statements.

But when we raise an Exception, we force the caller to treat the condition as an exceptional case, whether the caller considers it to be one or not.

For instance, many HTTP libraries—including Ruby's `Net::HTTP`, depending on how you call it—raise Exceptions for any 400 or 500-series HTTP response. This can be frustrating, when, for instance, you are expecting a 401 (Unauthorized) challenge in order to determine what type of authentication to supply. Why should you have to trap an exception just to get at the headers in the response object?

When I'm faced with a decision like this, where I don't have enough information to make a hard-and-fast judgment, I always like to find a way to punt on the question. In this case, that means using a *caller-supplied fallback strategy*.

Caller-supplied fallback strategy

> In most cases, the caller should determine how to handle an error, not the callee.
>
> – *Brian Kernighan and Rob Pike*, The Practice of Programming

A great example of letting the caller decide how to treat a possibly-exceptional condition is the `#fetch` method. `#fetch` is a standard method defined on `Hash` and `Array` and on some of Ruby's other collection classes. It takes a key and a block.

If the collection is able to find a value corresponding to the given key, the value is returned. If it is *not* able to find a value, it yields to the provided block. This enables the caller to define the policy for missing keys.

```
h.fetch(:optional_key){ DEFAULT_VALUE }
h.fetch(:required_key) {
  raise "Required key not found!"
}
```

Listing 43: Using `fetch` to supply fallbacks.

A similar example is the `#detect` method on `Enumerable`. Since the block was already taken, `#detect` uses a parameter for its caller-specified fallback handler.

```
arr.detect(lambda{"None found"}) {|x| ... }
```

This is a terrifically useful idiom to use in your own methods. Whenever you're not sure if a certain case constitutes an exception, consider whether you can delegate the decision to the caller in some way.

```ruby
def render_user(user)
  if user.fname && user.lname
    "#{user.lname}, #{user.fname}"
  else
    yield
  end
end

# Fall back to a benign placeholder value:
render_user(u){ "UNNAMED USER" }

# Fall back to an exception:
render_user(u){ raise "User missing a name" }
```

Listing 44: Example of caller-supplied fallback strategy.

Some questions before raising

Here are five questions to ask yourself before writing code to raise an Exception. They sum up the guidelines expressed in the preceding sections, and add a few new considerations.

#1: Is the situation truly unexpected?

Do we really need to raise an exception when the user fails to answer either "y" or "n" to a question?

```ruby
puts "Confirm (y/n)"
answer = gets.chomp
raise "Huh?" unless %w[y n].include?(answer)
```

Maybe we could just loop until we get a sensible response:

```ruby
begin
  puts "Confirm (y/n)"
  answer = gets.chomp
end until %w[y n].include?(answer)
```

User input is a classic example of a case where we *expect* mistakes. Ask yourself: Is this really an unexpected case, or can you reasonably predict it will happen during normal operation?

#2: Am I prepared to end the program?

Remember that any Exception, if it goes un-handled, can potentially end the program. Or (in the case of a web application) at least end the request. Ask yourself if the failure really justifies terminating the program.

When you look at it this way, you may realize that the consequences of failure don't preclude the program from continuing. When this is the case, you may find that it makes sense to substitute some kind of benign value for the expected result.

For example, this...

```
@ug = UserGreeting.find_by_name!("winter_holidays")
```

...could become this:

```
@ug = UserGreeting.find_by_name("winter_holidays")
unless @ug
  logger.error "Someone forgot to run db:populate!"
  @ug = OpenStruct.new(:welcome => "Hello")
end
```

Listing 45: Substituting a benign value for a missing record.

It's longer, but it's less catastrophic.

#3: Can I punt the decision up the call chain?

Ask yourself: is there any way I could allow the caller to push its own decisions down into my code, and maybe provide some sensible defaults? Do I want to capture and log failure instead of aborting? Am I certain this condition represents an unexpected condition, and if not, can I delegate the decision to the caller?

#4: Am I throwing away valuable diagnostics?

```
result = some_fifteen_minute_operation()
if result.has_raisins_in_it?
  raise "I HATE RAISINS"
end
```

When you have just received the results of a long, expensive operation, that's probably not a good time to raise an exception because of a trivial formatting error. You want to preserve as much of that information as possible—if nothing else, so that you can get to the bottom of the failure more easily. See if you can enable the code to continue normally while still noting the fact that there was a problem, using some kind of sideband. Or, make sure that any exceptions raised have all of the relevant diagnostic data attached to them before they are raised.

#5: Would continuing result in a less informative exception?

Sometimes failing to raise an exception just results in things going wrong in less easy-to-diagnose ways down the road. In cases like this, it's better to raise earlier than later. This is often the case with precondition assertions: little checks at the beginning of a method that the input has the expected format.

Compare this:

```
response_code = might_return_nil()
message = codes_to_messages[response_code]
response = "System Status: " + message
# What do you mean "Can't convert nil into string"?!
```

. . . to this:

```
response_code = might_return_nil() or raise "No response code"
# ..
```

Isolate exception handling code

> An exception represents an immediate, non-local transfer of control—it's a kind of cascading `goto`. Programs that use exceptions as part of their normal processing suffer from all the readability and maintainability problems of classic spaghetti code.
>
> – *Dave Thomas and Andy Hunt*, The Pragmatic Programmer

Exception-handling code is messy, there's no getting around that. And when it interrupts the flow of business logic, it makes programs harder to understand, harder to debug, and harder to test.

Have you ever come across code that looks like this?

```
begin
  try_something
  rescue
    begin
      try_something_else
    rescue
      # handle failure
    end
  end
end
```

I've seen code that was nested many levels deep in `begin ... rescue` blocks. Wherever I've discovered code like that, I've found it to be some of the buggiest and hardest to maintain code in the project.

I consider the `begin` keyword to be a code smell in Ruby. Remember, a code smell doesn't mean it's always wrong—it's just a warning sign that there might be a better way to do things.

In this case, the "better way" is to use Ruby's *implicit begin blocks*. Every method in Ruby is implicitly a `begin` block. We can put a rescue clause at the end of the method and omit the `begin` and the extra `end`:

```
def foo
  # mainline logic goes here
rescue # ------------------
  # failure handling goes here
end
```

Listing 46: Implicit begin block.

This makes for a really elegant idiom, because it draws a neat dividing line between the mainline business logic of a method, and the failure handling part of the method.

I find that sticking to this idiom, and avoiding `begin` blocks—having just a single failure handling section in any given method—leads to cleaner, clearer, better-factored Ruby code.

Isolating exception policy with a Contingency Method

One way to refactor `begin`-heavy code towards this ideal of a single `rescue` clause per method is to make use of a *contingency method*. A contingency method cleanly separates business logic from failure handling. It is helpful in code which repeatedly has to cope with the possibility of a particular class of failure.

Let's say we have a codebase which makes numerous calls which might raise an `IOError`:

```
begin
  something_that_might_fail
rescue IOError
  # handle IOError
end

# ...

begin
  something_else_that_might_fail
rescue IOError
  # handle IOError
end
```

We can refactor the error handling out into a contingency method:

```
def with_io_error_handling
   yield
rescue
  # handle IOError
end

with_io_error_handling { something_that_might_fail }

# ...

with_io_error_handling { something_else_that_might_fail }
```

Listing 47: A contingency method.

The contingency method's sole job is to implement a failure handling policy for a given class of exceptions. Now that we've gotten rid of a `begin`, the business logic is less cluttered by failure handling code. The policy for `IOError` handling has been moved to a single location. And we can re-use this contingency method to clean up other sections of code.

Exception Safety

> Should a library attempt a recovery when something goes wrong? Not usually, but it might do a service by making sure it leaves information in as clean and harmless a state as possible.
>
> – *Brian Kernighan and Rob Pike*, The Practice of Programming

Some methods are critical. We can't afford for them to fail. For instance, in the failure cascade example above, the error-reporting methods were critical. Methods that could potentially destroy user data or violate security protocols if they fail are critical.

When it is vital that a method operate reliably, it becomes useful to have a good understanding of how that method will behave in the face of unexpected Exceptions being raised. A method's level of *exception safety* describes how it will behave in the presence of exceptions.

The three guarantees

Classically, there are three defined levels of exception safety, the "three guarantees," each defining more stringent failure-mode semantics than the last:

The weak guarantee If an exception is raised, the object will be left in a consistent state.

The strong guarantee If an exception is raised, the object will be rolled back to its beginning state.

The nothrow guarantee No exceptions will be raised from the method. If an exception is raised during the execution of the method it will be handled internally.

There's an implicit fourth level (or zero-th level) of exception safety, the level which makes no guarantee whatsoever. When critical methods make no guarantees about exception safety, the code is ripe for a cascading failure scenario.

"But wait" you might object, "this code doesn't do anything that could raise an exception". But is that really the case?

When can exceptions be raised?

Pop quiz: what parts of the following code might raise an exception?

```
size   = File.size('/home/avdi/.emacs')
kbytes = size / 1024
puts "File size is #{size}k"
```

It's a trick question, I'm afraid. The correct answer is "any of it." There are some exceptions, like `NoMemoryError` and `SignalException`, that can be raised at any point in execution. For instance, if the user hits Ctrl-C, the resulting `SignalException` could crop up *anywhere*.

Exception safety testing

So we know that for some methods, it is a good idea to have well-defined exception semantics, in the form of one of the Three Guarantees. We also know that exceptions can be raised at any point in a Ruby program.

How do we verify that these critical methods actually live up to their exception-safety promises? The answer lies in *exception safety testing*.

Exception safety testing is a technique for testing assertions about a method's semantics in the face of exceptions. It has four main elements:

1. A **test script** is a chunk of code that will exercise the method being tested.

2. One or more **assertions**—predicates returning `true` or `false`—will be made about the method.

3. The test script will be instrumented and run once in a **record phase**. In this phase, every external method call made by the code under test is recorded as a list of call points.

4. In the **playback phase**, the test script is executed once for each call point found in the record phase. Each time, a different call point is forced to raise an exception. After each execution, the assertions are checked to verify that they have not been violated as a result of the exception that was raised.

   To illustrate, let's use a simple `#swap_keys` method as our unit-under-test. It takes a hash and two keys, and swaps the values of the two keys:

```
def swap_keys(hash, x_key, y_key)
  temp = hash[x_key]
  hash[x_key] = hash[y_key]
  hash[y_key] = temp
end
```

We define a Hash to operate on, and a test script which simply calls `#swap_keys` on the Hash:

```
h = {:a => 42, :b => 23}
tester = ExceptionTester.new{ swap_keys(h, :a, :b) }
```

The `ExceptionTester` instruments the script and runs it once to record a list of callpoints.

```
def swap_keys(hash, x_key, y_key)
  temp = hash[x_key]①
  hash[x_key] = ③hash[y_key]②
  hash[y_key] = ④temp
end
```

① Hash#[]
② Hash#[]
③ Hash#[]=
④ Hash#[]=

Figure 1: Exception testing record phase

Now we define an assertion: either the keys are fully swapped or not swapped at all. This is the "strong" guarantee: completion or full rollback.

```
tester.assert{
  (h == {:a => 42, :b => 23}) ||
  (h == {:a => 23, :b => 42})
}
```

The tester runs the test script again, once for each of the four call points discovered during the record phase. Each time, it forces a different call to fail; and each time it re-checks the assertions.

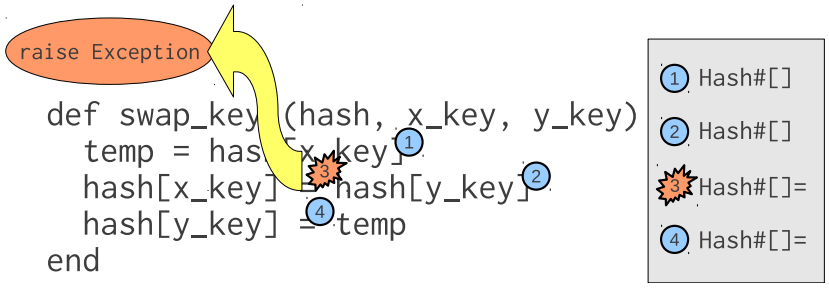

Figure 2: Exception testing playback phase

Because it must run each method under test once for every recorded call-point, exception testing is an expensive technique and probably something you should only consider for your most critical methods. For those methods though, it can give you a much-needed confidence boost that your program will function as expected when things go wrong.

Implementation of the exception-safety tester

I've created a proof-of-concept implementation of the exception tester demonstrated above. It's a little rough, but it works. See Appendix A for code and explanation.

Validity vs. consistency

The "weak" safety guarantee says that an object will be left in a consistent state even if an exception is raised. But what does "consistent" mean?

It's important to realize that consistency is not the same as validity. Here's how I define the two:

Validity Are the business rules for the data met?

Consistency Can the object operate without crashing, or exhibiting undefined behavior?

We can preserve consistency even while permitting the object to be invalid from a business perspective. For instance, we can make sure a record doesn't have any dangling references to other records that have been destroyed.

In the first chapter we discussed how an object's "invariant" is a contract that all of its public methods must adhere to. Typically an object's invariant includes rules to maintain *consistency*, but it does not necessarily make assertions about *validity*.

For example, let's say we have a LedgerEntry class which may contain one or more Credits and Debits. The invariant for a `LedgerEntry` object might include the statement "all Credit IDs must reference existing Credit records", but might not include the statement "the sum of credits minus the sum of debits must equal zero". Valid credit and debit IDs are required for normal functioning—otherwise an exception might be raised while trying to tally up the sums of debits and credits. But the requirement that the sums balance out is a business rule which we may only want to enforce before persisting the object to a database.

Be specific when eating exceptions

Here's a common antipattern:

```
begin
  # ...
rescue Exception
end
```

Listing 48: Over-broad `rescue`.

Sometimes you're forced to catch an overly broad exception type, because a library has failed to define its own exception classes and is instead raising something generic like `RuntimeError` or even `Exception`. Unfortunately, code such as the above example is prone to bugs. Often, you'll spend a long time trying to figure out why the code is not doing what it's supposed to only to discover that an exception is being raised early—and then being caught and hidden by the over-eager `rescue` clause.

If you can't match on the class of an Exception, try to at least match on the message. Here's a snippet of code that checks for a specific failure based on message and then re-raises if it doesn't find a match:

```
begin
  # ...
rescue => error
  raise unless error.message =~ /foo bar/
end
```

Listing 49: Matching exceptions by message.

Namespace your own exceptions

> A routine should present a consistent abstraction in its interface, and so should a class. The exceptions thrown are part of the routine interface, just like the specific data types are.
>
> – *Steve McConnell,* Code Complete

The way to avoid putting yourself and other programmers in the position of writing the code in the previous example is to make a habit of always namespacing the Exceptions raised in your code. This is especially important when writing libraries, but it's worth doing in apps as well.

Every library codebase should have, at the very least, a definition such as the following:

```
module MyLibrary
  class Error < StandardError; end
end
```

That gives client code something to match on when calling into the library.

Be careful, though—just because your code only raises the namespaced exceptions, doesn't mean other code that your library calls doesn't raise its own exceptions. In researching this book I heard from more than one developer who was frustrated by libraries that raised a seemingly limitless variety of exceptions—as soon as they put in a handler for `IOError`, they discovered that the library also sometimes raises a `SystemCallError`.

A way to be considerate to your library clients is to put code in the top-level API calls which catches non-namespaced Exceptions and wraps them in a

namespaced exception (using the Nested Exceptions described in a previous section) before re-raising them. In this way the client still has full access to the details of the failure, but can confidently rescue only exceptions in the library namespace.

```
# immediately re-raise namespaced exceptions, otherwise they would
# be caught by the second, more generic rescue
rescue MyLibrary::Error
  raise
rescue => error
  # This assumes MyLibrary::Error supports nesting:
  raise MyLibrary::Error.new(
    "#{error.class}: #{error.message}",
    error)
end
```

Listing 50: Library-namespaced exceptions.

Tagging exceptions with modules

There's a potential objection to the library-namespaced exceptions demonstrated above. Let's say we have a call to an external HTTP service, and we've found that the remote server is a little unreliable. To compensate for occasional connection failures, we've wrapped the call an IOError handler that retries the call:

```
begin
  response = Net::HTTP.get('example.com', '/service')
rescue IOError => error
  sleep 5
  retry
end
```

One day we replace our `Net::HTTP` calls with a new HTTP client library:

```
begin
  response = AcmeHttp.get("http://example.com/service")
rescue IOError => error
  sleep 5
  retry
end
```

Suddenly, we start getting failures. And when we look at the stack trace, we see that when a connection fails, `AcmeHttp` catches the internal `IOError` and raises a `AcmeHttp::Error` instead. Because the `rescue` block assumed `IOError`, failed requests are no longer retried. Since we didn't realize that `AcmeHttp` would raise a different error type (or we didn't remember that the retry code expects an IOError), we didn't think to change the `rescue` clause before putting the updated code into production.

Now let's switch perspectives. Look at it from the point of view of the authors of `AcmeHttp`. On the one hand, many of our library users have asked us to make all exceptions raised by `AcmeHttp` descend from a single class, in order to make it easier to set up catch-all `rescue` clauses. On the other hand, by replacing system exceptions with `AcmeHttp`-namespaced exceptions, we may violate someones Principle of Least Surprise (POLS). What to do?

As it turns out, there's a way to have our cake and eat it too. We can use *tag modules* to namespace the exceptions our library raises. A tag module is a module whose sole purpose is to act as a kind of *type tag*, which can be attached to arbitrary objects to indicate their membership in a particular group.

Let's define a tag module:

```
module AcmeHttp
  module Error; end
end
```

Now let's see how we can attach it to an exception:

```
begin
  begin
    raise IOError, "Some IO error"
  rescue Exception => error
    error.extend(AcmeHttp::Error)
    raise
  end
rescue AcmeHttp::Error => error
  puts "Rescued AcmeHttp::Error: #{error}"
end
```

Output

```
Rescued AcmeHttp::Error: Some IO error
```

Listing 51: Tagging exceptions with modules.

Notice that we are rescuing the module `AcmeHttp::Error`. Remember, `rescue` can accept any class *or* module object. Now that we have extended the original exception with the error tag module, the expression:

```
AcmeHttp::Error === error
```

evaluates to `true`, so the outer `rescue` intercepts the exception.

We can extract out the tagging code into a general-purpose helper method:

```
def tag_errors
  yield
rescue Exception => error
  error.extend(AcmeHttp::Error)
  raise
end
```

Listing 52: An exception tagging wrapper method.

Now we can use this helper in all of our top-level API methods, ensuring that any exceptions originating in the library will be tagged with the appropriate type:

```
module AcmeHttp
  def self.foo
    tag_errors do
      # ...
    end
  end
end
```

Listing 53: Tagging exceptions at the library boundary.

With tag modules, we can insure all of the exceptions which arise within our library are tagged with an identifying type, while still allowing code that expects non-library-defined exceptions to function unchanged.

The no-raise library API

In several of the preceding sections, you may have noticed an underlying theme: *minimizing exception-related surprises to library clients*. By being

specific when eating exceptions, we minimize the possibility that our library will unexpectedly swallow exceptions raised by client code inside of blocks. By namespacing or tagging exceptions, we give client code an easy way to rescue all exceptions our library might raise.

The exceptions a library may raise are just as much a part of its API as its classes and public methods. But Ruby doesn't provide any way to declare or verify which exceptions may be raised by a library. There isn't even a documentation convention for specifying the exceptions a method may raise.

As a result, exceptions are one of the most common sources of surprise and frustration with third-party libraries. I can't count the number of times I've had to fix bugs that stemmed either from not realizing that a library would raise a particular exception type, not realizing the library treated a particular case as an exceptional circumstance, or discovering that a library ate an exception that it should have passed through.

One strategy for avoiding the first two cases which we haven't discussed yet is to provide a *no-raise API*. With a no-raise API, you completely delegate the decision about how to handle failures—or even what to classify as a failure—to the client code. Your library may raise and rescue exceptions internally, but from the client perspective all outputs are in the form of either return values or callbacks; there is no need to document which exceptions may be raised because none will be.

The Typhoeus[15] library furnishes an excellent example of a no-raise library API. All of the information about a Typhoeus HTTP request—no matter what its final disposition—is encapsulated in the `Typhoeus::Response` class. Here's some code lifted from the Typhoeus README demonstrating error handling:

[15] https://github.com/dbalatero/typhoeus

```
request.on_complete do |response|
  if response.success?
    # hell yeah
  elsif response.timed_out?
    # aw hell no
    log("got a time out")
  elsif response.code == 0
    # Could not get an http response, something's wrong.
    log(response.curl_error_message)
  else
    # Received a non-successful http response.
    log("HTTP request failed: " + response.code.to_s)
  end
end
```

Listing 54: No-raise failure handling in Typhoeus

As you can see, every possible outcome, even network timeouts and connection errors, is accounted for within the response object. The client code makes the decision whether to ignore the failure, log it, or raise an exception.

Consider giving your library a no-raise API, either as its sole API or as a configurable option. You might even include custom `#exception` methods in your result objects, so that client code can easily convert the results to exceptions if it so chooses. Your library users will thank you if you do.

Three essential exception classes

The question of how to structure an exception class hierarchy is one that confronts every developer at one time or another. And there's surprisingly little guidance in the programming literature.

A few possibilities might come to mind:

- We could divide our Exceptions up by which **module** or subsystem they come from.
- We might break them down by software **layer**—e.g. separate exceptions for UI-level failures and for model-level failures.
- We could even try basing them on severity levels: separate exceptions for fatal problems and nonfatal problems, although usually that's a decision to be made where the exception is caught, not where it is raised.

Let's take a step back for a minute. Why do we create differentiated exception classes at all? Why don't we just raise `StandardError` everywhere?

There are only two reasons to raise differentiated exception classes. The first is that we wish to attach some extra information to the exception, like an error code, so we create an exception class with an extra attribute. The second and probably more common reason is that we expect to *handle the exception differently*. "Handling," in this case, may mean we take specific actions; but it also may just mean that we present the failure to the user in a particular way.

Instead of looking at our code for inspiration on how to structure an exception hierarchy, let's try working backwards. Let's look at the different ways failures are typically presented to the user.

Failures: the user's perspective

Most application failure messages break down into one of three categories. The examples below assume a web application, but the categories apply to just about any kind of application.

1. "You did something wrong. Please fix it and try again." This message tells the user that the only way to fix the problem is for them to do something differently. One form this may take is an HTTP 403 (Unauthorized) error with (hopefully) an informative message.
2. "Something went wrong inside the app. We have been notified of the issue." This tells the user that there is nothing they can do to fix the problem; it is an internal error and will have to be addressed by the web site's admin or development team.
3. "We are temporarily over capacity, please come back later." Exemplified by the legendary Twitter "fail whale," this message lets the user know that while nothing is *broken*, the system (or some resource it depends on) is temporarily over capacity and that the problem will most likely resolve itself eventually.

Three classes of exception

Working backwards from these three types of failure message, we can discern three basic classes of exception:

1. **User Error.** The user did something invalid or not allowed. Usually handled by notifying the user and giving them an opportunity to fix the problem.

2. **Logic Error.** There is an error in the system. Typically handled by notifying the system administrator and/or development team, and letting the user know that the problem is being looked into.

3. **Transient Failure.** Something is over capacity or temporarily offline. Usually handled by giving the user a hint about when to come back and try again, or, in the case of batch jobs, by arranging to re-try the failed operation a little later.

I find that if a library or app defines these three exception types, they are sufficient for 80% of the cases where an exception is warranted.

Here's an example of taking different concrete actions in a command-line app based on the three exception types above:

```
failures = 0
begin
  # ...
rescue MyLib::UserError => e
  puts e.message
  puts "Please try again"
  retry
rescue MyLib::TransientFailure => e
  failures += 1
  if failures < 3
    warn e.message
    sleep 10
    retry
  else
    abort "Too many failures"
  end
rescue MyLib::LogicError => e
  log_error(e)
  abort "Internal error! #{e.message}"
end
```

Listing 55: The three exception types.

When building a library, you may find it useful to break `LogicError` into two subclasses:

1. `InternalError`, for errors (bugs) in the library itself; and

2. `ClientError`, for incorrect use of the library.

This will help clients of your library discern errors which they can fix themselves by correcting their usage of the library from legitimate bugs in the library.

Conclusion

Robust failure handling is one of the trickiest aspects of software construction. I hope that you've learned something new from these pages, and had your imagination sparked to try new approaches to handling failures in your Ruby code. I welcome feedback; Please don't hesitate to contact me with questions, comments, suggestions, or errata. My email address is avdi@avdi.org.

If you enjoyed the content of this book, I invite you to visit my software development blog: http://avdi.org/devblog.

Thanks for reading, and happy hacking!

References

I quoted from a number of books in the preceding pages. All of these are books I keep close at hand while coding, and I recommend them to any programmer seeking to write quality code.

Note that the links below are Amazon affiliate links; if you follow them and buy a book I'll get a little bit of money. This money helps me write more eBooks and blog posts.

- Code Complete: A Practical Handbook of Software Construction, by Steve Mcconnell (Microsoft Press)
- Object-Oriented Software Construction, by Bertrand Meyer (Prentice Hall)
- Release It!: Design and Deploy Production-Ready Software, by Michael Nygard (Pragmatic Bookshelf)
- The Pragmatic Programmer: From Journeyman to Master, by Dave Thomas and Andy Hunt (Addison Wesley)
- The Practice of Programming, by Brian Kernighan and Rob Pike (Addison Wesley)

Appendix A: Exception tester implementation

Here in its entirety is the implementation of the ExceptionTester referenced in the exception safety testing section. It's pretty bare-bones, and could stand to be improved in a number of ways; but as it stands, it works as a proof of concept. It's my hope that if I don't find the time to improve it, some industrious reader will take the initiative to start building on the foundation I've laid (hint, hint).

The code below is also available on-line at: http://gist.github.com/772356

This code derives its power from the `#set_trace_func` call[16]. `#set_trace_func` enables the programmer to define a block of code to be executed at every single step of the interpreter. By using `#set_trace_func` we are able to instrument the test script and gather up a list of method calls which we can then reference in the playback phase.

First, we declare an `ExceptionTester` class, along with a `TestException` class which will be used when we inject an exception during playback.

```
require 'set'

class ExceptionTester
  class TestException < Exception
  end

  # ...
end
```

We will initialize the class with a block, which contains the code about which we want to make exception-safety guarantees.

[16] http://www.ruby-doc.org/core/classes/Kernel.html#M001430

```
def initialize(&exercise)
  @exercise = exercise
end
```

The `#assert` method takes a block containing a predicate which will be run after each playback, to determine if the code has maintained its safety guarantees.

```
def assert(&invariant)
  recording = record(&@exercise)
  recording.size.times do |n|
    playback(recording, n, &invariant)
    unless invariant.call
      raise "Assertion failed on call #{n}: #{@signature.inspect}"
    end
  end
end
```

We define a private `#record` method which will perform the "record" phase of the test.

```
private

def record(&block)
```

A recording is a `Set` of codepoint tuples. A codepoint consists of the event, filename, line number, id, and classname fields from arguments yielded by Ruby's `#set_trace_func`. We define a `recorder` lambda to record these tuples.

```
recording = Set.new
recorder = lambda do |event, file, line, id, binding, classname|
  recording.add([event, file, line, id, classname])
end
```

Next we pass the recorder to `#set_trace_func`, and initiate the recording by calling the block saved by `#initialize`. Once the block is finished we end the recording by passing `nil` to `#set_trace_func`.

```
set_trace_func(recorder)
block.call
set_trace_func(nil)
```

We are only interested in recording method calls for this implementation, so we throw away any codepoints that aren't either a Ruby method call or a C method call.

```
recording.reject!{|event| !%w[call c-call].include?(event[0])}
```

We also don't need recordings of the initial `#call` into the block, or of the second call to `#set_trace_func`.

```
# Get rid of calls outside the block
recording.delete_if{|sig|
  sig[0] == "c-call" &&
  sig[1] == __FILE__ &&
  sig[3] == :call    &&
  sig[4] == Proc
}
recording.delete_if{|sig|
  sig[0] == "c-call"           &&
  sig[1] == __FILE__           &&
  sig[3] == :set_trace_func    &&
  sig[4] == Kernel
}
```

With the recording filtered for just the codepoints we care about, now we can return it.

```
  recording
end
```

To play back a recording, we need the recording, a `fail_index` for the codepoint at which we should force an exception, and a reference to the `invariant` block to run after playback.

```
def playback(recording, fail_index, &invariant)
```

We'll start by setting up some variables.

```
recording      = recording.dup
recording_size = recording.size
call_count     = 0
```

Next we set up a `player` lambda. This will be passed to `#set_trace_func`, just like the `recorder` we defined earlier. Only this time, it will raise an exception when it reaches the selected codepoint.

```
player = lambda do |event, file, line, id, binding, classname|
  signature = [event, file, line, id, classname]
  if recording.member?(signature)
    @signature = signature
    call_count = recording_size - recording.size
    recording.delete(signature)
    if fail_index == call_count
      raise TestException
    end
  end
end
```

Once again, we use `#set_trace_func` to have our `player` be called at every step of execution. This time we have to capture any `TestExceptions` which escape the code under test. With that, our `ExceptionTester` class is complete.

```
  set_trace_func(player)
  begin
    @exercise.call
  rescue TestException
    # do nothing
  ensure
    set_trace_func(nil)
  end
end
```

To demonstrate this code, here's a simple method we can test for exception safety. It is a method to swap two keys in a hash.

```
def swap_keys(hash, x_key, y_key)
  temp = hash[x_key]
  hash[x_key] = hash[y_key]
  hash[y_key] = temp
end
```

We'll define a hash for it to operate on, and then create a new `ExceptionTester` to test the operation of `#swap_keys`.

```
h = {:a => 42, :b => 23}
tester = ExceptionTester.new{ swap_keys(h, :a, :b) }
```

Now we'll assert that no matter where an exception is raised inside `#swap_keys`, the target hash will either be fully swapped or reverted to its initial state (the Strong Guarantee).

```
tester.assert{
  # Assert the keys are either fully swapped or not swapped at all
  (h == {:a => 42, :b => 23}) ||
  (h == {:a => 23, :b => 42})
}
```

```
source/exception_tester.rb:21:in 'assert':
Assertion failed on call 4:
["c-call", "source/exception_tester.rb", 90, :[]=, Hash] (RuntimeError)
        from source/exception_tester.rb:18:in 'times'
        from source/exception_tester.rb:18:in 'assert'
        from source/exception_tester.rb:95
```

It's not the most readable output in the world, but what this tells us is that on the fourth run, when the tester artificially induced an exception from a call to Hash#[]= at line 90, the exception safety assertion we made was violated. Since the assertion was that the hash would either have its keys fully swapped or would be left in its original condition, this tells us that the `#swap_keys` method does *not* comply with the Strong Guarantee.

Appendix B: An incomplete tour of ruby's standard exceptions

Ruby provides a rich set of exception classes out of the box. Some of them are strictly for the language's own use, and you will only `rescue` them. Others may be useful to raise in your own code.

If you want a complete list of Ruby's built-in exception classes, the best way to get it is to ask your Ruby implementation. Nick Sieger has provided[17] a code snippet which does just that:

[17] http://blog.nicksieger.com/articles/2006/09/06/rubys-exception-hierarchy

```ruby
exceptions = []
tree = {}
ObjectSpace.each_object(Class) do |cls|
  next unless cls.ancestors.include? Exception
  next if exceptions.include? cls
  next if cls.superclass == SystemCallError # avoid dumping Errno's
  exceptions << cls
  cls.ancestors.delete_if {|e|
    [Object, Kernel].include? e
  }.reverse.inject(tree) {
    |memo,cls| memo[cls] ||= {}
  }
end

indent = 0
tree_printer = Proc.new do |t|
  t.keys.sort { |c1,c2| c1.name <=> c2.name }.each do |k|
    space = (' ' * indent); space ||= ''
    puts space + k.to_s
    indent += 2; tree_printer.call t[k]; indent -= 2
  end
end
tree_printer.call tree
```

Listing 56: Listing the Ruby exception hierarchy.

Here is the output in Ruby 1.8.7:

```
Exception
  NoMemoryError
  ScriptError
    LoadError
    NotImplementedError
    SyntaxError
  SignalException
    Interrupt
  StandardError
    ArgumentError
    IOError
      EOFError
    IndexError
      StopIteration
    LocalJumpError
    NameError
      NoMethodError
```

```
    RangeError
      FloatDomainError
    RegexpError
    RuntimeError
    SecurityError
    SystemCallError
    SystemStackError
    ThreadError
    TypeError
    ZeroDivisionError
  SystemExit
  fatal
```

What follows are some notes on selected exception classes.

NoMemoryError

This is raised when Ruby can't allocate any more memory for objects. If you want to simulate an out-of-memory situation without bringing your computer to its knees, and you have access to a Linux machine, you can use ulimit:

```
$ ulimit -v100000
$ ruby -e 'a = []; loop do a << "x" * 1024 end '
-e:1:in `*': failed to allocate memory (NoMemoryError)
        from -e:1
        from -e:1:in `loop'
        from -e:1
```

There is probably an OS X equivalent to the ulimit command used above, but I don't have access to an OS X machine on which to experiment.

ScriptError

Ruby uses the `ScriptError` subclasses `LoadError` and `SyntaxError` to indicate failures in loading and executing Ruby scripts. `NotImplementedError` is also included in this category.

SignalException

Signal exceptions are raised when a Ruby process is signaled by the OS. A list of common signals in UNIX-like systems can be found on Wikipedia[18]. They can be raised at **any time**—from whatever bit of code is being executed when the signal is received.

`Interrupt` is a special subclass of `SignalException` which is raised for the `SIGINT` signal. `SIGINT` is what you get when the user presses <Ctrl-C>.

If want to avoid having critical operations interrupted by unexpected SignalExceptions, you must define a trap[19]. When a signal is trapped, execution transfers to the trap block and then, once the block is finished, resumes the interrupted code where it left off.

StandardError

This class and all its subclasses will be rescued by a default `rescue` clause. Any exceptions outside the `StandardError` hierarchy must be rescued explicitly.

In general, all application and library exceptions should be descended from `StandardError`. One case where you may want to consider sub-classing something other than `StandardError` is when the exception represents a condition which you know to be fatal. In that case, you don't want any default `rescue` clauses accidentally rescuing the exception.

RuntimeError

This is the exception class we get when we call `raise` (or `fail`) without an explicit class.

```
begin
  raise "Oops"
rescue => error
  error.inspect
end
```

Output

```
    #<RuntimeError: Oops>
```

[18] http://en.wikipedia.org/wiki/Signal_(computing)#List_of_signals
[19] http://www.ruby-doc.org/core/classes/Kernel.html#M001428

In a sense `RuntimeError` is Ruby's "miscellaneous" exception. It says nothing about the failure except that there was one.

`RuntimeError` has an interesting name. In traditional compiled languages, a strong distinction is made between logic errors and runtime failures. A logic error indicates a mistake in the program, one which theoretically could have been prevented by sufficiently attentive coding or peer-review.

Runtime exceptions, by contrast, represent conditions which could not possibly be detected until the program is run. These are typically conditions which originate in the program's runtime environment: unexpected inputs, missing files, network timeouts.

Ruby has no `LogicError`, but many of `RuntimeError`'s sibling classes effectively represent logic errors: e.g. `ArgumentError`, `NoMethodError`, `LocalJumpError`.

In a sense every Ruby exception is a runtime error; as a highly dynamic language, you can't even necessarily predict if a given chunk of code will raise `NoMethodError` by inspection alone.

So I recommend simply thinking of `RuntimeError` as "miscellaneous failure" and using it in short scripts or in new code when you haven't yet solidified your exception hierarchy. For libraries and more mature applications, it's probably better to give every exception a more specific type. See "Three essential exception classes" for some suggestions on how to structure your exception hierarchy.

IOError

`IOError` indicates various unexpected IO-related conditions. It has a subclass, `EOFError`, which is raised by certain I/O calls when a file or stream hits EOF (End Of File).

Note that calls which may raise `IOError` are likely to raise `SystemCallError` subclasses as well.

ArgumentError, RangeError, TypeError, IndexError

These are all typically raised at the beginnings of methods to indicate that a method was called incorrectly.

SystemCallError

You will never see a straight `SystemCallError` raised. Instead, what you will see is one of its descendants, which are all named `Errno::<ERROR SYMBOL>`, where *<ERROR SYMBOL>* is the symbolic name for a system error code (such as `ENOENT`).

If you are interested, you can discover the actual integer error code by calling `#errno` on the object. If you want to see a list of all possible error codes on your system, look at the value of `Errno.constants`.

The actual subclasses of `SystemCallError` will vary from system to system. Here is the full list from my system (Ubuntu 10.10), as well as the code I used to generate the list:

```
table = Errno.constants.sort.map do |error_symbol|
  error_class = Errno.const_get(error_symbol)

  # Passing nil tells it to ask the system for an error message.
  exc = SystemCallError.new(nil, error_class::Errno)
  [error_symbol, exc.errno, exc.message]
end
[%w[Symbol Code Description]] + table
```

Table 1: List of System Call Errors

Symbol	Code	Description
E2BIG	7	Argument list too long
EACCES	13	Permission denied
EADDRINUSE	98	Address already in use
EADDRNOTAVAIL	99	Cannot assign requested address
EADV	68	Advertise error
EAFNOSUPPORT	97	Address family not supported by protocol
EAGAIN	11	Resource temporarily unavailable
EALREADY	114	Operation already in progress
EBADE	52	Invalid exchange
EBADF	9	Bad file descriptor
EBADFD	77	File descriptor in bad state
EBADMSG	74	Bad message
EBADR	53	Invalid request descriptor
EBADRQC	56	Invalid request code
EBADSLT	57	Invalid slot
EBFONT	59	Bad font file format
EBUSY	16	Device or resource busy
ECHILD	10	No child processes
ECHRNG	44	Channel number out of range

ECOMM	70	Communication error on send
ECONNABORTED	103	Software caused connection abort
ECONNREFUSED	111	Connection refused
ECONNRESET	104	Connection reset by peer
EDEADLK	35	Resource deadlock avoided
EDEADLOCK	35	Resource deadlock avoided
EDESTADDRREQ	89	Destination address required
EDOM	33	Numerical argument out of domain
EDOTDOT	73	RFS specific error
EDQUOT	122	Disk quota exceeded
EEXIST	17	File exists
EFAULT	14	Bad address
EFBIG	27	File too large
EHOSTDOWN	112	Host is down
EHOSTUNREACH	113	No route to host
EIDRM	43	Identifier removed
EILSEQ	84	Invalid or incomplete multibyte or wide character
EINPROGRESS	115	Operation now in progress
EINTR	4	Interrupted system call
EINVAL	22	Invalid argument
EIO	5	Input/output error
EISCONN	106	Transport endpoint is already connected
EISDIR	21	Is a directory
EISNAM	120	Is a named type file
EL2HLT	51	Level 2 halted
EL2NSYNC	45	Level 2 not synchronized
EL3HLT	46	Level 3 halted
EL3RST	47	Level 3 reset
ELIBACC	79	Can not access a needed shared library
ELIBBAD	80	Accessing a corrupted shared library
ELIBEXEC	83	Cannot exec a shared library directly
ELIBMAX	82	Attempting to link in too many shared libraries
ELIBSCN	81	.lib section in a.out corrupted
ELNRNG	48	Link number out of range
ELOOP	40	Too many levels of symbolic links
EMFILE	24	Too many open files
EMLINK	31	Too many links
EMSGSIZE	90	Message too long
EMULTIHOP	72	Multihop attempted
ENAMETOOLONG	36	File name too long
ENAVAIL	119	No XENIX semaphores available
ENETDOWN	100	Network is down
ENETRESET	102	Network dropped connection on reset
ENETUNREACH	101	Network is unreachable
ENFILE	23	Too many open files in system
ENOANO	55	No anode

ENOBUFS	105	No buffer space available
ENOCSI	50	No CSI structure available
ENODATA	61	No data available
ENODEV	19	No such device
ENOENT	2	No such file or directory
ENOEXEC	8	Exec format error
ENOLCK	37	No locks available
ENOLINK	67	Link has been severed
ENOMEM	12	Cannot allocate memory
ENOMSG	42	No message of desired type
ENONET	64	Machine is not on the network
ENOPKG	65	Package not installed
ENOPROTOOPT	92	Protocol not available
ENOSPC	28	No space left on device
ENOSR	63	Out of streams resources
ENOSTR	60	Device not a stream
ENOSYS	38	Function not implemented
ENOTBLK	15	Block device required
ENOTCONN	107	Transport endpoint is not connected
ENOTDIR	20	Not a directory
ENOTEMPTY	39	Directory not empty
ENOTNAM	118	Not a XENIX named type file
ENOTSOCK	88	Socket operation on non-socket
ENOTTY	25	Inappropriate ioctl for device
ENOTUNIQ	76	Name not unique on network
ENXIO	6	No such device or address
EOPNOTSUPP	95	Operation not supported
EOVERFLOW	75	Value too large for defined data type
EPERM	1	Operation not permitted
EPFNOSUPPORT	96	Protocol family not supported
EPIPE	32	Broken pipe
EPROTO	71	Protocol error
EPROTONOSUPPORT	93	Protocol not supported
EPROTOTYPE	91	Protocol wrong type for socket
ERANGE	34	Numerical result out of range
EREMCHG	78	Remote address changed
EREMOTE	66	Object is remote
EREMOTEIO	121	Remote I/O error
ERESTART	85	Interrupted system call should be restarted
EROFS	30	Read-only file system
ESHUTDOWN	108	Cannot send after transport endpoint shutdown
ESOCKTNOSUPPORT	94	Socket type not supported
ESPIPE	29	Illegal seek
ESRCH	3	No such process
ESRMNT	69	Srmount error
ESTALE	116	Stale NFS file handle

ESTRPIPE	86	Streams pipe error
ETIME	62	Timer expired
ETIMEDOUT	110	Connection timed out
ETOOMANYREFS	109	Too many references: cannot splice
ETXTBSY	26	Text file busy
EUCLEAN	117	Structure needs cleaning
EUNATCH	49	Protocol driver not attached
EUSERS	87	Too many users
EWOULDBLOCK	11	Resource temporarily unavailable
EXDEV	18	Invalid cross-device link
EXFULL	54	Exchange full

SecurityError

`SecurityError` indicates a violation of the code's current `$SAFE` level:

```
$SAFE = 4
puts "Blah blah blah"
```

```
-:4:in ‘write’: Insecure operation ‘write’ at level 4 (SecurityError)
        from -:4:in ‘puts’
        from -:4:in ‘main’
        from -:7
```

See the relevant chapter of the Pickaxe[20] for more on `$SAFE` levels.

Timeout::Error

`Timeour::Error` is not part of Ruby's built-in set of exceptions; it is loaded when the `timeout` library is required. But since it often crops up in Ruby programs it's worth mentioning here.

In Ruby 1.8, `Timeout::Error` derives from `Interrupt`, which means it is not a `StandardError` and won't be rescued by a default `rescue` clause. This is a common source of bugs, because it's not always obvious that a library may raise `Timeout::Error`. When it finally *is* raised there may be no explicit `rescue` in place to handle the case.

Ruby 1.9 rectifies this situation by having `Timeout::Error` derive from `RuntimeError`.

[20] http://www.ruby-doc.org/docs/ProgrammingRuby/html/taint.html